Programação concorrente - Cozinha de Fast Food

Victor Costa

10 de maio de 2021

1 Introdução

Este artigo tem por objetivo simular a concorrência correta de uma cozinha de um restaurante de *fast-food*. Em um restaurante de *fast-food* típico as tarefas que existem giram em torno de anotar e entregar pedidos, preparar comida, limpar o ambiente e reestocar ingredientes. Para o propósito educacional desse problema o foco será na entrega de pedidos e preparação de hambúrgueres e porções de batatas fritas. Para a recepção de pedidos assume-se que os clientes conseguem pedir sozinhos. As outras tarefas são consideradas magicamente feitas.

Para a resolução desse problema será utilizada a biblioteca *pthread* e a linguagem de programação C++ versão de 2011 (`-std=c++11`). Essa biblioteca fornece funções para gerenciamento de *threads* e a sincronização delas. Para resolver este problema optou-se por usar *locks* e variáveis de condição.

2 Formalização do problema

O objetivo é simular um restaurante. Este restaurante é de *fast-food* e nele se vende hambúrgueres e porções de batata frita apenas. Os clientes fazem pedidos através de um totem que disponibiliza os pedidos para os funcionários do restaurante automaticamente. Após receber um pedido, funcionários deverão operar as máquinas e estações para atender ao pedido do primeiro cliente da fila. As máquinas e estações que existem são:

- Fritadeira: frita batatas sozinha. Para funcionar precisa de um preparo feito por algum funcionário. Ela automaticamente frita e disponibiliza uma quantidade de batatas fritas sem sal.
- Estação de salgagem de fritas: após fritar, as batatas precisam de sal e de serem colocadas em caixas. Esta estação necessita de toda a atenção do funcionário Uma fritada das fritadeiras permite que se façam mais de uma porção de batatas fritas.
- Chapa: o funcionário frita as carnes dos hambúrgueres aqui. Este equipamento exige toda a atenção do chapeiro, porém no fim frita várias carnes de uma vez.
- Estação de montagem de hambúrgueres: após fritar as carnes, precisa-se montar o hambúrguer com todos os outros ingredientes. Nesta estação o funcionário mantém toda sua concentração e no fim do processo gera apenas um hambúrguer pronto.
- Balcão de entrega: aqui o funcionário gasta um tempo montando o pedido do primeiro na fila e entrega a ele. Exige toda a atenção do funcionário.

Os valores dos parâmetros de tempo, quantidade de comida gerada por tarefa e quantidade de funcionários por estação não são relevantes para a resolução do problema, então foram todos deixados como constantes configuráveis no programa. Todas as estações têm limite de funcionários operando que deve ser observado na resolução do problema. Todos os funcionários só conseguem realizar uma tarefa por vez.

3 Descrição do algoritmo da solução do problema

3.1 Principais desafios

Este problema está cheio de condições de corrida, mas elas essencialmente consistem de 2 tipos: limitar quantidade de trabalhadores e garantir a validade das quantidades de comida produzidas. O primeiro tipo aparece quando existem múltiplos funcionários tentando acessar a mesma máquina e é resolvido com uma variável de controle que exige acesso exclusivo. O segundo tipo ocorre no momento em que uma estação produz um recurso que outra quer utilizar e para resolver a variável que cuida da quantidade do recurso só pode ser acessada exclusivamente.

Existem também problemas de possíveis *deadlocks* no momento da aquisição de múltiplos recursos. Para resolver isso, todas as funções foram conferidas na ordem que eram chamadas para remover ciclos de requisitos e em momentos de requisição de múltiplos *locks*, eles sempre foram adquiridos em uma ordem única para o programa.

Além desses desafios, deve-se deixar as *threads* dormindo em momentos que não têm tarefas para realizar. Ao retirar esperas ocupadas evita-se *starvation* de linhas de execução e otimiza-se o uso do processador. Para isso fez-se uso extensivo de variáveis de condição.

3.2 Resolvendo condições de corrida

3.2.1 Retirada segura de recursos

A primeira condição de corrida se refere a retirada de um recurso para uso em outra tarefa. Nela asseguramos que o valor do recurso é consistente, evitando itens fantasma e sumiço de itens. Para não ter que compartilhar o *lock* com outros módulos, optou-se for fazer uma função que testa e retira. O algoritmo 1 implementa a solução desse problema.

Algoritmo 1: Retirada segura de recursos da estação de montagem de hambúrgueres.

```
1  bool AssemblyStation::getBurgers(int n)
2  {
3    bool canDo = false;
4    pthread_mutex_lock(&mutex);
5    if (burgers >= n)
6    {
7      burgers -= n;
8      statusDisplayer->updateAssemblyStationBurgers(burgers);
9      canDo = true;
10   }
11   pthread_mutex_unlock(&mutex);
12
13   if (!canDo)
14     return false;
15
16   return true;
17 }
```

Nas linhas 5 a 10 do algoritmo 1 está localizada a região crítica, onde se testa a possibilidade de remoção da quantidade recebida. O acesso exclusivo é garantido pelas linhas 4 e 11, por conta dos *locks* que garantem que só uma linha de execução conseguem obter a trava. No caso de possível a retirada, ela ocorre dentro dessa mesma região. De forma a aumentar a concorrência, faz-se uso de uma variável local booleana chamada `canDo` que guarda se foi possível ou não retirar. Fora da região crítica, das linhas 13 a 16 é informado à quem chamou a função se foi possível ou não extrair o valor.

A linha 8 do algoritmo 1 é uma chamada a um método de uma classe responsável por mostrar o estado do restaurante de forma organizada após cada alteração. Este módulo é completamente destacável do programa e não é relevante para a resolução do problema.

3.2.2 Limitação de funcionários e adição segura de recursos

Para limitar os funcionários, faz-se um algoritmo muito parecido. A maior mudança é a condição de uso e os efeitos colaterais. A condição nas linhas 5 e 6 do algoritmo 2 limita a quantidade de traba-

lhadores pela variável de controle `workers` e usa a função `Griddle::getBurgerMeats`, que funciona de forma similar ao algoritmo 1, para testar se tem recursos para realizar a tarefa. No caso em que ambas as condições sejam verdadeiras, ajusta-se o valor de `workers` para garantir que não sejam excedidos o número de trabalhadores nessa estação e também ajusta-se a quantidade de recursos retirada. O ajuste da quantidade retirada ocorre dentro da função `Griddle::getBurgerMeats` para não ter a necessidade de lidar com *locks* de outro módulo.

Existem estações de trabalho que não precisam verificar a existência de recursos, como a fritadeira e a grelha, e nesse caso apenas retira-se da condição a função que pede por recursos.

No momento que o funcionário terminar sua tarefa (após o `sleep` da linha 16), adquire-se o *lock* referente aos trabalhadores e recursos gerados da própria estação (linha 17). Tendo esta trava, o algoritmo atualiza a variável de controle `workers` para indicar que o funcionário deixou a estação de trabalho. Ao mesmo tempo é incrementada a variável dos recursos produzidos pela estação (linha 19), adicionando os recursos de forma segura.

Ao fim de toda tarefa os trabalhadores são acordados, pois uma estação ficou livre para funcionários realizarem tarefas nela. Isso será mais aprofundado na seção 3.4.

Algoritmo 2: Limitação de funcionários e adição segura de recursos na estação de montagem de hambúrgueres.

```
1  bool AssemblyStation::makeBurgers()
2  {
3    bool canDo = false;
4    pthread_mutex_lock(&mutex);
5    if (workers < maxWorkers
6      && Griddle::getBurgerMeats(burgersPerBatch))
7    {
8      statusDisplayer->updateAssemblyStationWorkers(++workers);
9      canDo = true;
10   }
11   pthread_mutex_unlock(&mutex);
12
13   if (!canDo)
14     return false;
15
16   sleep(assemblingTime);
17   pthread_mutex_lock(&mutex);
18   statusDisplayer->updateAssemblyStationWorkers(--workers);
19   burgers += burgersPerBatch;
20   statusDisplayer->updateAssemblyStationBurgers(burgers);
21   pthread_mutex_unlock(&mutex);
22
23   Worker::broadcastAvailableTasks();
24   return true;
25 }
```

3.2.3 Caso especial: fritadeira

As fritadeiras fogem um pouco desse padrão por funcionarem de forma independente. O algoritmo 3 tem uma função parecida com a do algoritmo 2, a maior diferença é a forma de controlar a quantidade de trabalhadores e que ela só solicita o funcionamento das fritadeiras ao invés de incrementar na quantidade de recursos gerados. Nessa estação de trabalho optou-se por usar uma trava para o controle das batatas sem sal e uma trava para o controle das `threads` de forma a promover uma maior concorrência.

Na linha 6, ao invés de saber quantos trabalhadores têm e o máximo deles, o módulo da fritadeira opta por saber quantas fritadeiras estão disponíveis. Essa mudança de arquitetura facilita o controle das *threads* que representam as fritadeiras e realizam seu funcionamento automático. A outra parte da condição serve para limitar a quantidade de fritas sem sal produzidas e o motivo desse limite se encontra na seção 3.4. Após confirmar a possibilidade de usar a fritadeira, marca-se na variável `available` que uma a menos se encontra disponível.

Na linha 22, após o tempo de preparo na linha 20 passar, modifica-se uma variável de controle `requested` para demonstrar que uma fritadeira pode iniciar o trabalho. Logo depois na linha 24 é dado um sinal na variável de condição das fritadeiras para que uma comece a fritar.

Algoritmo 3: Preparação das fritadeiras.

```
1  bool DeepFriers::setupDeepFrier()
2  {
3    bool canDo = false;
4    pthread_mutex_lock(&mutex);
5    pthread_mutex_lock(&mutexUnsaltedFries);
6    if (available > 0
7      && unsaltedFries
8        + (1 + maxFriers - available) * friesPerBatch <= 240)
9    {
10     canDo = true;
11     available--;
12     statusDisplayer->updateDeepFrierAvailable(available);
13   }
14   pthread_mutex_unlock(&mutex);
15   pthread_mutex_unlock(&mutexUnsaltedFries);
16
17   if (!canDo)
18     return false;
19
20   sleep(setupTime);
21   pthread_mutex_lock(&mutex);
22   requested++;
23   pthread_mutex_unlock(&mutex);
24   pthread_cond_signal(&condition);
25
26   return true;
27 }
```

O algoritmo 4 simula o funcionamento independente das fritadeiras. No início, elas demonstram que estão prontas para uso incrementando `available` (linha 16). Logo após, na linha 27, elas verificam se houve algum pedido de funcionamento (fim do preparo visto no algoritmo 3). Caso não tenham, na linha 28, elas esperam até receberem o sinal de que foi requisitado. Para a eventualidade que uma *thread* acorde e outra já tenha respondido ao pedido, essa uma confere se ainda existem pedidos. Assim que elas começam a fritar, na linha 29, elas marcam que atenderam o pedido.

Depois de atender ao pedido de funcionamento, a fritadeira gasta um tempo na linha 36 fritando as batatas. Quando ela terminar de fritar, ela realiza uma adição segura, de forma similar às linhas 17 a 21 no algoritmo 2.

Algoritmo 4: *Threads* das fritadeiras.

```
1  void *DeepFriers::DeepFrier(void *args)
2  {
3    int id = *(int *)args;
4    delete (int *)args; // Prevent memory leaks
5
6    std::ostringstream out;
7    out << "Deep frier of id " << id << " instantiated" << std::endl;
8    if (loggingEnabled)
9      logFile << out.str();
10   out.str("");
11
12   while (runThreads)
13   {
14     pthread_mutex_lock(&mutex);
```

```
15      /* Show that the deep frier is free */
16      if (++available == 1)
17      Worker::broadcastAvailableTasks();
18
19      statusDisplayer->updateDeepFrierAvailable(available);
20
21      out << "DeepFrier[" << id << "]: Estou pronto." << std::endl;
22      if (loggingEnabled)
23        logFile << out.str();
24      out.str("");
25
26      /* Wait for a request */
27      while (requested <= 0 && runThreads)
28        pthread_cond_wait(&condition, &mutex);
29      requested--;
30      pthread_mutex_unlock(&mutex);
31
32      if (!runThreads)
33        break;
34
35      /* Fry */
36      sleep(fryTime);
37
38      /* Add unsalted fries */
39      pthread_mutex_lock(&mutex);
40      unsaltedFries += friesPerBatch;
41      out << "DeepFrier[" << id << "]: Fritei. Agora temos "
42        << unsaltedFries << " batatas." << std::endl;
43      if (loggingEnabled)
44        logFile << out.str();
45      out.str("");
46      if (unsaltedFries >= SaltingStation::friesPerPortion)
47        Worker::broadcastAvailableTasks();
48
49      statusDisplayer->updateDeepFrierUnsaltedFries(unsaltedFries);
50      pthread_mutex_unlock(&mutex);
51    }
52    return nullptr;
53  }
```

Todas as chamadas ao `statusDisplayer` servem para atualizar uma classe responsável por mostrar o estado do programa de uma forma mais geral e não é relevante para a solução do problema. Nas *threads* desse programa existem também escritas a um arquivo de *log* referenciado pela variável `logFile` que são opcionais e servem apenas para ajudar na resolução de problemas ou para ver o programa de outra forma. Também pode-se notar chamadas a uma função de acordar trabalhadores, ela notifica os funcionários de que existe uma tarefa disponível. Essa funcionalidade é melhor discutida na seção 3.4.

3.3 Evitando *deadlocks*

Não existem muitos problemas com *deadlocks* no programa, porém eles podem vir escondidos graças a modularização do código. Dentro de um mesmo módulo com vários *locks* há só os exemplos dos algoritmos 3 e 5. Nesses algoritmo, o *deadlock* é resolvido garantindo que os *locks* são sempre adquiridos e liberados na mesma ordem.

Quando a questão é obter *locks* entre módulos, a forma como fluem os recursos resolvem esse problema. Na figura 1 é possível ver quais módulos fazem chamadas a quais módulos requisitando recursos e por conta dessa ordem unidirecional, não existem possibilidades de *deadlocks*.

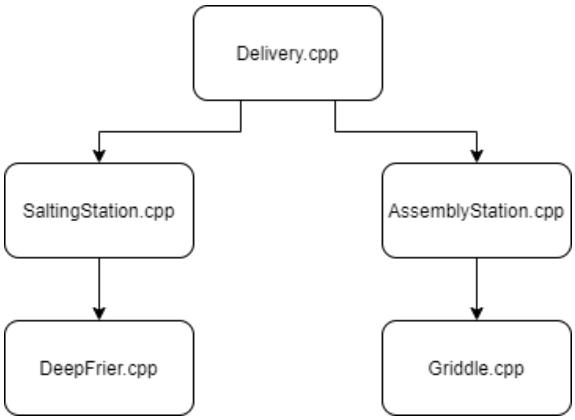

Figura 1: Desenho demonstrando como cada módulo usa os *locks* de outros módulos

Algoritmo 5: Entrega de pedido no módulo de *Delivery*.

```
1  bool Delivery::deliverOrder()
2  {
3    bool canDo = false;
4    order_t front;
5
6    pthread_mutex_lock(&mutex);
7    pthread_mutex_lock(&SaltingStation::mutex);
8    pthread_mutex_lock(&AssemblyStation::mutex);
9    if (ordersQueue.size() > 0)
10   {
11
12     front = ordersQueue.front();
13
14     if (workers < maxWorkers
15       && SaltingStation::fries >= front.fries
16       && AssemblyStation::burgers >= front.burgers)
17     {
18       ordersQueue.pop();
19       ++workers;
20       AssemblyStation::burgers -= front.burgers;
21       SaltingStation::fries -= front.fries;
22       totalBurgers -= front.burgers;
23       totalFries -= front.fries;
24       statusDisplayer
25         ->updateDeliveryWorkers(workers);
26       statusDisplayer
27         ->updateAssemblyStationBurgers(AssemblyStation::burgers);
28       statusDisplayer
29         ->updateSaltingFries(SaltingStation::fries);
30       statusDisplayer
31         ->updateDeliveryOrdered(totalFries, totalBurgers);
32       statusDisplayer
33         ->updateDeliveryFirstOrder(ordersQueue);
34       canDo = true;
35     }
36   }
37   pthread_mutex_unlock(&mutex);
38   pthread_mutex_unlock(&SaltingStation::mutex);
39   pthread_mutex_unlock(&AssemblyStation::mutex);
40
41   if (!canDo)
```

```
42      return false;
43
44    sleep(assemblingTime);
45
46    // Allows one customer to order more.
47    pthread_cond_signal(&waitForOrderDelivered);
48
49    pthread_mutex_lock(&mutex);
50    if (--workers == maxWorkers - 1)
51      Worker::broadcastAvailableTasks();
52    statusDisplayer->updateDeliveryWorkers(workers);
53    pthread_mutex_unlock(&mutex);
54
55    return true;
56  }
```

3.4 Evitando esperas ocupadas

Para evitar esperas ocupadas foram usadas variáveis de condição em todas as *threads*. Existem apenas 3 tipos de *threads* nesse programa: a fritadeira, o funcionário e o freguês. O código referente a *thread* da fritadeira se encontra no algoritmo 4 na seção 3.2.3 junto da explicação de como dorme e acorda.

O funcionário fica em busca de tarefas até que não exista absolutamente nenhuma no momento para executar. Na linha 20 do algoritmo 6 ele descobre o que precisa ser feito primeiro e nas linhas 21 a 36 ele tenta em ordem cada uma das prioridades. Caso não seja possível entregar pedidos, executar um passo na linha de produção de batatas fritas ou executar um passo na linha de hambúrgueres, a *thread* do usuário dorme como descrito na linha 38 e 46. O mecanismo para acordar é bem simples, criou-se uma função `Worker::broadcastAvailableTasks()` que faz um *broadcast* a todos os funcionários dormindo de que há a possibilidade de realizar uma tarefa. Essencialmente sempre que uma tarefa acaba de uma forma que abra espaço para trabalhadores realizarem alguma tarefa que antes não era possível, essa função é chamada e acorda todos os funcionários. A partir daí é uma corrida para fazer tarefas. Dessa forma é evitado que algum funcionário fique incansavelmente tentando fazer cada uma das tarefas, tomando tempo de processamento de outras *threads* úteis.

Algoritmo 6: O código executado pela *thread* de funcionário.

```
1   void *Worker::Worker(void *args)
2   {
3     std::vector<Delivery::priority_type_t> priorities;
4     bool doneSomething;
5
6     std::ostringstream out;
7     int id = *(int *)args;
8     delete (int *)args;
9
10    out << "Worker of id " << id << " instantiated" << std::endl;
11    if (loggingEnabled)
12      logFile << out.str();
13    out.str("");
14
15    while (runThreads)
16    {
17      doneSomething = false;
18      while (!doneSomething)
19      {
20        priorities = Delivery::getPriority();
21        for (auto priority : priorities)
22        {
23          switch (priority)
```

```
24          {
25            // Tenta realizar tarefas seguindo ordem de prioridade
26            // [...]
27          }
28
29          if (doneSomething)
30          {
31            if (loggingEnabled)
32              logFile << out.str();
33            out.str("");
34            break;
35          }
36        }
37        pthread_mutex_lock(&mutex);
38        if (!doneSomething)
39        {
40          out << "Worker[" << id
41            << "]: Sem tarefas para fazer, vou esperar por uma."
42            << std::endl;
43          if (loggingEnabled)
44            logFile << out.str();
45          out.str("");
46          pthread_cond_wait(&waitForTask, &mutex);
47        }
48        pthread_mutex_unlock(&mutex);
49      }
50    }
51    return nullptr;
52  }
```

Para os fregueses fica um algoritmo bem mais simples. Pode-se ver no algoritmo 7 que consiste de um produtor que dorme: nas linhas de 16 a 24 o freguês pensa num pedido; na linha 26 ele faz o pedido inserindo numa fila e; na linha 37 ele espera pela entrega do seu pedido. Quando ocorre a entrega de um pedido, sempre acorda-se um freguês. Dessa forma na fila têm-se sempre a quantidade de fregueses em pedidos.

Algoritmo 7: Código executado pela *thread* de freguês.

```
1   void *Delivery::Customer(void *args)
2   {
3     order_t order;
4     std::ostringstream out;
5     int id = *(int *)args;
6     delete (int *)args;
7
8     out << "Customer of id " << id << " instantiated" << std::endl;
9     if (loggingEnabled)
10      logFile << out.str();
11    out.str("");
12
13    while (runThreads)
14    {
15      /* Think of an order */
16      order.burgers = 0;
17      order.fries = 0;
18      while (order.burgers <= 0 && order.fries <= 0)
19      {
20        order.burgers = random() % 6;
21        order.fries = std::max(
22          order.burgers + ((int)random() % 5) - 2, 0
23        );
```

```
24      }
25
26      placeOrder(order);
27      out << "Customer[" << id << "]: Pedi " << order.fries
28        << " fritas e " << order.burgers << " hamburgueres."
29        << std::endl;
30      if (loggingEnabled)
31        logFile << out.str();
32      out.str("");
33
34      // The customer waits for a delivery to be made
35      // before ordering more.
36      pthread_mutex_lock(&orderMakingMutex);
37      pthread_cond_wait(&waitForOrderDelivered, &orderMakingMutex);
38      pthread_mutex_unlock(&orderMakingMutex);
39    }
40
41    return nullptr;
42  }
```

Por fim o último mecanismo adotado foi limitar a quantidade de produtos gerados pelas primeiras etapas de produção, a fritadeira e a grelha. As tarefas realmente importantes de salgar as batatas e de montar hambúrgueres criam um gargalo por conta de sua produção mais lenta. Quando se combina isso com um excesso de funcionários, haverá sempre alguém fritando hambúrgueres e fritando batatas, podendo acumular uma quantidade enorme de carnes e batatas. Isso no fim das contas cria uma espécie de espera ocupada em que os funcionários ficam fazendo tarefas sem valor para o momento. Sendo assim foram adicionados mecanismos que limitam a quantidade de fritas sem sal e de carnes de hambúrguer como vistos nas linha 7 e 8 do algoritmo 3.

3.5 Sobre *starvation*

Este programa está sujeito a *starvations*, sendo alguns casos parte da simulação e outros não relevantes. No caso podem ocorrer quando existem funcionários demais, porém depende muito da sorte, já que teria de ter um grupo de funcionários que ficam se alternando nas tarefas, enquanto os resto não consegue adquirir os *locks* para realizar uma tarefa antes desse pertencentes ao grupo. Esse comportamento é esperado e pode ajudar a mostrar ineficiências na cozinha quando muitos trabalhadores ficam a espera por tarefas. O outro caso é quando se leva em consideração os fregueses, sendo possível que sempre a mesma *thread* seja acordada para fazer outro pedido e a outra fique dormindo. Esse comportamento não é problemático, pois eles estão lá apenas para criar uma espécie de fila infinita de pedidos e fregueses.

Já para as *threads* de fritadeiras, a necessidade de fritas sem sal para a estação de salgagem evitam que elas não possam trabalhar por conta dos trabalhadores. Entre as fritadeiras existe sim a possibilidade de *starvation*, porém não importa muito, pois as outras continuam gerando fritas e pode servir como um indicativo de que há muitas fritadeiras.

3.6 Exibindo o estado da simulação

Por fim o último elemento a explicar é como mostrar o estado do sistema de uma forma geral e menos poluída. Para resolver esse problema foi criada uma classe `StatusDisplayer` cuja função é fazer isso. Para não ter que reorganizar o programa para gerenciar o estado em um único módulo, optou-se por fazer com que cada mudança nas variáveis interessantes chamasse essa classe atualizando o valor. Sempre que atualiza é escrito na tela o novo estado. Para garantir que sejam registradas **todas** as mudanças de estado, foi criado um *lock* na própria classe que protege todas as mudanças, forçando que sempre haverá um print correto para cada mudança.

4 Conclusão

Este trabalho foi mais grande que o esperado, porém foi possível botar muito bem em prática os mecanismos aprendidos na matéria de programação concorrente. Após criar o programa foi notado

que poderia ter sido mais fácil implementar certas partes com semáforos e também praticar melhor o reuso com herança, métodos e funções. Apesar desses pontos de melhoria, o código desenvolvido simula corretamente uma cozinha de *fast-food* seguindo as especificações propostas na seção 2. Além disso esse programa fornece um algoritmo que aproveita bem a concorrência das *threads*, sem causar *deadlocks* e evitando *starvations* ao máximo.

Referências

[1] Lawrence Livermore National Laboratory Blaise Barney. *POSIX Threads Programming*. URL: `https://hpc-tutorials.llnl.gov/posix/`.

[2] *C and C++ reference*. URL: `https://en.cppreference.com/w/`.

[3] Victor André Gris Costa. *victoragcosta/ConcurrentProgramming-FastFoodKitchen*. URL: `https://github.com/victoragcosta/ConcurrentProgramming-FastFoodKitchen`.