Capítulo 2

Sistemas concorrentes

Os sistemas concorrentes têm como origem a existência de vários processos ou threads executados de forma colaborativa para uma mesma tarefa, sendo que os sistemas nos quais estão inseridos podem conter um único processador ou múltiplos processadores. No caso de um único processador, os processos se revezam conforme os parâmetros de escalonamento estabelecidos pelo sistema operacional. E, no caso de múltiplos processadores, as tarefas podem ser executadas em vários processadores, com cada tarefa sendo executada em um processador diferente. Evidentemente, em sistemas concorrentes, deve ocorrer o compartilhamento de recursos. Por isso, os processos concorrentes precisam ser sincronizados para garantir a sua correta interpretação e evitar problemas na aplicação.

O objetivo deste capítulo é apresentar os sistemas concorrentes e demonstrar como funciona o compartilhamento de recursos, o uso de processos e threads, além da implementação de controles para as regiões críticas por mecanismos de exclusão mútua, *mutex* e semáforo.

1 Os sistemas concorrentes e a memória compartilhada

Dentro do ambiente de sistemas distribuídos, os aplicativos fornecem recursos que são compartilhados pelos usuários, e dessa forma pode ocorrer acesso concomitante ao recurso compartilhado. A estratégia de aceitar um pedido de cliente por vez limita o funcionamento do sistema, e, portanto, a permissão para que vários pedidos de clientes sejam processados concorrentemente se torna a melhor estratégia. Para que se tenha sucesso nessa operação, é necessário que o processo em um sistema distribuído seja responsável por garantir a operação correta em um ambiente concorrente, obedecendo aos critérios de compartilhamento. O gerenciamento e a operação do uso concorrente podem ser realizados utilizando-se semáforos e locks, que estão disponíveis nos sistemas operacionais. Caso esses critérios não sejam seguidos, poderá ocorrer inconsistência nos dados (COULOURIS; DOLLIMORE; KINDBERG, 2007).

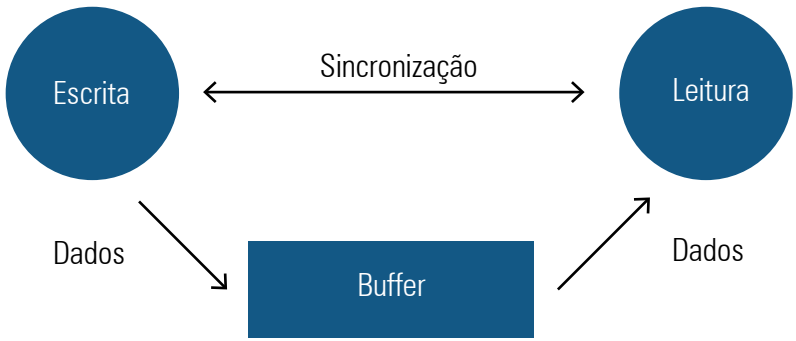
Tanenbaum e Van Steen (2008) informam que as aplicações concorrentes podem usar diversos métodos de comunicação. Um dos exemplos é o uso de memória compartilhada entre os processos ou através da troca de mensagens entre processos em execução.

A memória compartilhada é um mecanismo de comunicação utilizado através de um buffer, que é compartilhado entre os processos para as operações de escrita e leitura, e na operação de escrita o processo grava dados no buffer somente quando este estiver vazio. Já na

operação de leitura, um processo lê dados no buffer quando existe algo. A sincronização é o mecanismo que coloca os processos para aguardar a sua execução, evitando conflitos (MACHADO; MAIA, 2017).

A figura 1 ilustra esse comportamento. No lado esquerdo, temos o processo *escrita*, e no lado direito o processo *leitura*. Entre os processos escrita e leitura há uma seta bidirecional que realiza o sincronismo entre a escrita e a leitura. Abaixo dos dois processos, há a representação do buffer recebendo a entrada de dados do processo escrita e da saída dos dados para o processo leitura.

Figura 1 – Comunicação entre processos concorrentes



Fonte: adaptado de Machado e Maia (2017).

Os processos concorrentes devem se comunicar ao realizar qualquer atividade no conteúdo da memória compartilhada através de estratégias bem definidas. Entre essas estratégias, destacam-se exclusão mútua, semáforo e gerenciamento (LYNCH, 1996). Para alcançar esse objetivo, as linguagens de programação devem prover suporte para a concorrência. Um dos exemplos é a linguagem de programação Python, que utiliza o modelo de memória compartilhada com o bloqueio de acesso realizado por monitores.

NA PRÁTICA

O website do Vale Lab 4, da Universidade da Califórnia, disponibiliza um artigo intitulado “Sharing memory between processes”, tratando do compartilhamento de memória entre processos. Neste site há uma explicação teórica sobre o assunto e também uma prática para a criação de segmentos de memória para serem compartilhados entre processos em linguagem C.

2 Uso de processos e threads

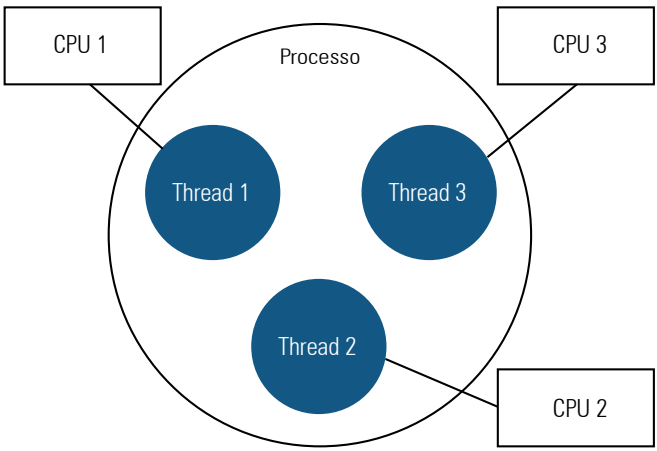
Com a evolução dos sistemas, a forma tradicional de um processo, executado em um único fluxo, tornou-se insuficiente para o compartilhamento de recursos entre atividades relacionadas. Uma nova solução foi proposta aprimorando-se a noção de processo e associando múltiplas atividades (threads), para que assim fosse possível ocorrer a distribuição do processamento e a tarefa não precisasse mais ser sequencial.

Na figura 2, é possível visualizar a execução sequencial de um processo, ficando o processador, a memória e os demais recursos computacionais alocados de forma dedicada até o final da tarefa. A figura 2 é descrita com três itens: processo, execução e término. O passo 1 é o início do processo, com a alocação dos recursos; já o passo 2 é a execução do processo, ficando todos os recursos alocados até que o processo seja finalizado; e, finalmente, no passo 3 ocorre o término da execução do processo.

Figura 2 – Fluxo linear

Já as threads são criadas dentro de processos, e os processos possuem no mínimo uma linha de execução, conforme é ilustrado na figura 3. Na imagem, temos um único processo com três threads, sendo cada thread associada a um processador e ocorrendo o processamento simultâneo.

Figura 3 – Threads



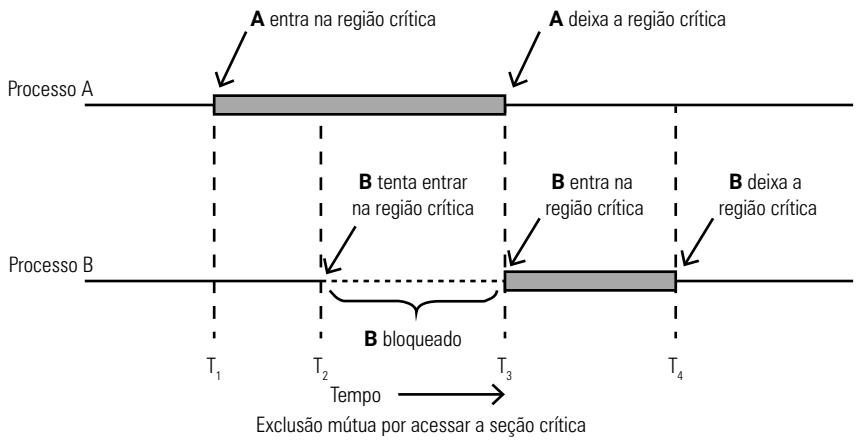
Para elucidar o conceito de processos e threads, vejamos esta analogia feita por Coulouris, Dollimore e Kindberg (2007):

> Um ambiente de execução consiste em um jarro tampado com ar e o alimento dentro dele. Inicialmente há uma mosca – uma thread – no jarro. Essa mosca, assim como suas descendentes, pode produzir outras moscas e matá-las. Qualquer mosca pode consumir qualquer recurso (ar ou alimento) no jarro. As moscas podem ser programadas para entrarem em uma fila, de maneira ordenada, para consumir recursos. Se não tiverem essa disciplina, elas podem se chocar umas com as outras dentro do jarro – isto é, colidir e produzir resultados imprevisíveis ao tentarem consumir os mesmos recursos de maneira não controlada. As moscas podem se comunicar com (enviar mensagens para) moscas de outros jarros, mas nenhuma pode escapar do jarro e nenhuma mosca de fora pode entrar nele. (COULOURIS; DOLLIMORE; KINDBERG, 2007, p. 206)

3 Implementação de mutex e semáforo

Para evitar que dois processos ou threads tenham acesso simultâneo a um recurso compartilhado, problema conhecido na computação como condição de corrida (race condition), é necessário que as regiões críticas sejam protegidas por mecanismos de exclusão mútua. Esse mecanismo impede que mais de um processo leia ou escreva em memória ou qualquer outro componente compartilhado. Para impedir esse tipo de problema é necessário conhecermos as técnicas mutex (mutual exclusion, ou exclusão mútua) e semáforo. A figura 4 apresenta o mecanismo que garante que um processo está sendo executado exclusivamente, sendo outros processos excluídos caso tentem acessar a mesma área. Ainda na figura 4, é demonstrado que o processo A é executado nos instantes T_1, T_2 e T_3. No instante T_2, um segundo processo – o processo B – tenta acessar a mesma região do processo A, mas é bloqueado. A área só é liberada ao processo B após a execução por completo do processo A, no instante T_3.

Figura 4 – Exclusão mútua no acesso de região crítica



Fonte: Yu (2010).

O semáforo é uma variável presente nas linguagens de programação na qual podem ser aplicadas duas operações: incremento e decremento, trabalhando com mecanismos de sinalização para informar se se utilizam ou liberam recursos. Já os mutexes são uma versão simplificada dos semáforos, com apenas dois valores possíveis, sendo que o primeiro valor possível serve para adquirir (bloquear) um recurso, e o segundo valor possível, para liberar (desbloquear).

Um objeto semáforo pode ser criado conforme o código a seguir, no qual "counter" representa o número máximo de threads permitidas para acesso simultâneo.

```
s=Semaphore(counter)
```

Para verificar quantos núcleos de processadores existem no seu computador, execute o comando a seguir.

```
from multiprocessing import cpu_count
cpu_count()
```

Na execução desse comando na IDLE do Python, a saída foram 4 CPUs. No próximo exemplo, para não utilizarmos todo o recurso computacional, usaremos 2 CPUs e, posteriormente, 4.

IMPORTANTE

Vale lembrar que a saída do comando a seguir pode diferir do seu computador, porém é importante para os próximos exemplos a utilização de 2 CPUs e 4 CPUs para a correta assimilação do conteúdo.

>>>> from multiprocessing import cpu_count

>>>> cpu_count()

4

>>>>

No programa a seguir, levando em consideração que o contador de semáforo é 2 (CPUs), as threads 1 e 2 são executadas paralelamente três vezes, e, após o término da sua execução, as outras threads, 3 e 4, serão iniciadas, executadas três vezes e finalizadas conforme demonstra a parte (a) da figura 5. Na parte (b) da figura, o contador de semáforo foi modificado para 4 (CPUs), e sendo assim as threads 1, 2, 3 e 4 são executadas paralelamente três vezes, em um intervalo de tempo menor se comparado com as 2 CPUs, e mesmo assim preservando os dados compartilhados.

IMPORTANTE

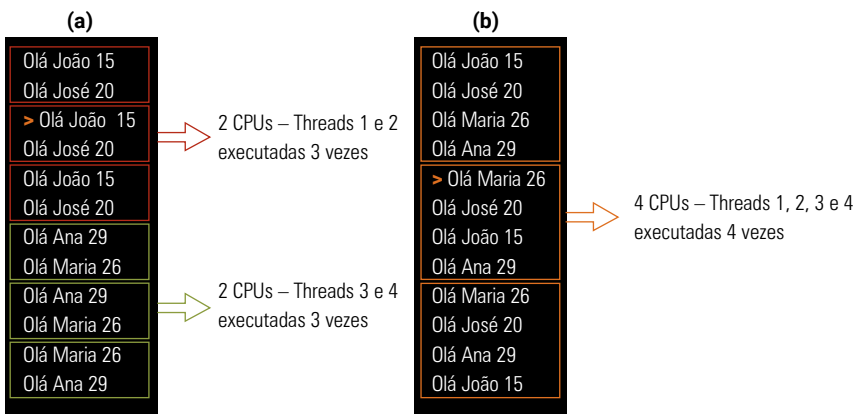
Existem dois modos para a execução do programa descrito, através do IDLE, de forma interativa, ou de alguma IDE de sua preferência. Caso opte por executar pelo IDLE, é necessário executar a opção do "Run Module" de forma interativa ou, então, criar um script através da IDE e executar diretamente via prompt.

```
from threading import *
import time

s=Semaphore(2)

def example(nome,idade):
  for i in range(3):
      s.acquire()
      print("Olá",nome, idade)
      time.sleep(2)
      s.release()

tread1=Thread(target=example, args=("João",15))
tread2=Thread(target=example, args=("José",20))
tread3=Thread(target=example, args=("Maria",26))
tread4=Thread(target=example, args=("Ana",29))
```

(*cont.*)

```
tread1.start()
tread2.start()
tread3.start()
tread4.start()
```

Figura 5 – Execução de semáforos com 2 CPUs (a) e 4 CPUs (b)



PARA SABER MAIS

Para se aprofundar no assunto mutexes e semáforos, recomendamos a leitura do artigo "Mutexes and semaphores demystified", da BarrGroup Software Experts. Neste artigo, são explicadas as principais diferenças e exemplos práticos de seu uso. Para isso, acesse a página oficial da BarrGroup.

Considerações finais

Neste capítulo, foram apresentados os sistemas concorrentes e a memória compartilhada, com a sua utilização de forma colaborativa com o compartilhamento de recursos. Vimos que, para que esse compartilhamento ocorra de forma satisfatória, é necessário o sincronismo

entre os processos, para evitar assim problemas de utilização do mesmo recurso e, consequentemente, inconsistência na aplicação.

Também abordamos que o gerenciamento e a operação no uso concorrente podem ser realizados utilizando semáforos e locks e implementando controles para as regiões críticas por mecanismos de exclusão mútua, mutex e semáforo.

Para encerrar, vimos um exemplo simples desse controle através de uma implementação utilizando linguagem de programação Python.

Referências

COULOURIS, George; DOLLIMORE, Jean; KINDBERG, Tim. **Sistemas distribuídos**: conceitos e projeto. 4. ed. Porto Alegre: Bookman, 2007.

LYNCH, Nancy. **Distributed algorithms**. Nova York: Morgan Kaufmann Publishers, 1996.

MACHADO, Francis Berenger; MAIA, Luiz Paulo. **Arquitetura de sistemas operacionais**. Rio de Janeiro: LTC, 2017.

TANENBAUM, Andrew S.; VAN STEEN, Maarten. **Sistemas distribuídos**: princípios e paradigmas. São Paulo: Prentice Hall, 2008.

YU, Tong Lai. **CSE 460 Operating Systems**. 2010. Disponível em: http://cse.csusb.edu/tongyu/courses/cs460/notes/interprocess.php. Acesso em: 20 out. 2020.