Sincronização

Ricardo de la Rocha Ladeira

Introdução

Na aula passada iniciou-se o estudo sobre *threads*, envolvendo conceitos, exemplos de aplicações, modelos *multithreading*, abordagens de *threading* implícita e uso de bibliotecas como `pthread` e OpenMP.

Novamente, aproveitando a base já obtida pelas aulas passadas, dar-se-á sequência no estudo sobre processos e *threads*, trabalhando ainda mais com `pthread` e mecanismos de sincronização.

Como visto na aula sobre Comunicação entre Processos, um *processo cooperativo* é aquele que pode afetar ou ser afetado por outros processos em execução no sistema. Os processos cooperativos podem compartilhar diretamente código e dados ou ter permissão para compartilhar dados apenas por meio de arquivos ou mensagens. O primeiro caso é obtido com o uso de *threads*. No entanto, o acesso concorrente aos dados compartilhados pode resultar em incoerência de dados. Nesta aula, vamos discutir sobre mecanismos para garantir a execução ordenada de processos cooperativos que compartilham um espaço de endereços lógico, de forma a manter a consistência dos dados.

Grande parte deste texto é baseada no capítulo 6 (*Sincronização de processos*) de [1], cuja leitura já foi solicitada na aula anterior.

Sincronização de Processos e *Threads*

Um sistema normalmente conta com várias *threads* executando tarefas de processos concorrentemente ou em paralelo. Enquanto isso, o SO atualiza continuamente várias estruturas de dados para dar suporte às *threads*.

Threads comumente compartilham dados, o que precisa acontecer de forma controlada. O exercício 5 da aula passada discutiu exatamente este ponto sobre o código do arquivo `pthread-exercicio-5-aula-4.c`. Neste código, a tarefa de cada *thread* é simplesmente atualizar uma variável compartilhada incrementando o seu valor. Porém,

vimos que há momentos em que a execução do código entrega respostas distintas devido à falta de controle de acesso à variável compartilhada por parte de duas ou mais *threads*. Isso pode ser exemplificado da seguinte forma:

- a *thread* 0 acessa a variável compartilhada e incrementa o seu valor (de *0* para *1*)
- a *thread* 1 acessa a variável compartilhada ao mesmo tempo que a *thread* 0, e incrementa o valor da variável compartilhada (de *0* para *1*)
- a *thread* 2 acessa a variável compartilhada ao mesmo tempo que a *thread* 1, e incrementa o valor da variável compartilhada (de *0* para *1*)

Caso os resultados não estejam sendo diferentes, sugere-se aumentar o número de iterações, por exemplo, para 10.000. Neste caso, respostas diferentes de 40.000 indicam que condições de corrida ocorreram e assim produziram resultados inesperados.

Em uma linguagem típica de máquina, a instrução de incremento poderia ser implementada assim:

```
registrador = contador
registrador = registrador + 1
contador = registrador
```

No caso acima de acesso simultâneo por três *threads*, poderia ocorrer o seguinte:

```
T0 executa registrador = contador               registrador = 0
T0 executa registrador = registrador + 1        registrador = 1
T1 executa registrador = contador               registrador = 0
T0 executa contador = registrador               contador = 0
T1 executa registrador = registrador + 1        registrador = 1
T2 executa registrador = contador               registrador = 0
T1 executa contador = registrador               contador = 0
T2 executa registrador = registrador + 1        registrador = 1
T2 executa contador = registrador               contador = 1
```

Idealmente, gostaríamos que as instruções ocorressem de forma sequencial:

- a *thread* 0 acessa a variável compartilhada e incrementa o seu valor (de *0* para *1*)
- a *thread* 1 acessa a variável compartilhada e incrementa o seu valor (de *1* para *2*)
- a *thread* 2 acessa a variável compartilhada e incrementa o seu valor (de *2* para *3*)

Como a situação de acesso simultâneo pode acontecer em diversas ocasiões, o valor final obtido pode ser muito diferente do valor esperado.

Condição de Corrida

Diz-se que existe uma ***condição de corrida*** **(*race condition*)** quando o acesso aos dados compartilhados não é controlado, possivelmente resultando em valores de dados corrompidos. No exercício abordado, a condição de corrida sobre a variável compartilhada gerou resultados diferentes em algumas execuções.

Exemplos de condições de corrida:

- **Reserva de passagens aéreas**. Vários usuários podem reservar os mesmos assentos em um voo. Sem um mecanismo de sincronização, dois ou mais usuários podem reservar o mesmo assento ao mesmo tempo.
- **Transferência bancária**. Diversos clientes podem transferir valores entre contas ao mesmo tempo. Sem sincronização, as operações concorrentes podem resultar em saldos incorretos ou até mesmo duplicação de transferências.
- **Aplicativo de mensagens instantâneas**. Vários usuários podem enviar mensagens para o mesmo destinatário ao mesmo tempo. As mensagens podem ser exibidas fora de ordem ou partes de mensagens diferentes podem ser intercaladas.
- **Sistema de Arquivos**. Dois ou mais processos podem criar, escrever ou apagar arquivos simultaneamente. Com isso, a estrutura do sistema de arquivos pode ficar inconsistente, com operações incoerentes, arquivos corrompidos ou desatualizados. Um exemplo está disponível neste vídeo.
- **Jogos *online***. Em jogos *multiplayer online*, vários jogadores podem tentar realizar ações que afetam o mundo do jogo, como coletar moedas ou aplicar dano a inimigos. Sem sincronização, as ações de jogadores diferentes podem causar resultados conflitantes ou situações ilógicas no mundo do jogo.
- **Edição colaborativa em documentos**. Vários editores tentam modificar um documento simultaneamente. Neste caso, as edições concorrentes podem ser perdidas ou conflitantes, tornando difícil manter uma versão coerente do documento.

Todos esses exemplos ilustram situações em que a falta de sincronização adequada pode levar a resultados indesejados, conflitantes ou incorretos. A sincronização é fundamental para garantir que as operações ocorram de maneira controlada e previsível, evitando conflitos e garantindo a integridade dos dados.

A sincronização de processos e *threads* envolve o uso de ferramentas que controlem o acesso a dados compartilhados para evitar condições de corrida. Essas ferramentas devem ser utilizadas com cuidado, pois seu uso incorreto pode resultar em baixo desempenho do sistema, inclusive em *deadlock*[1].

Seção Crítica

Uma *seção crítica* é uma parte do código onde um ou mais processos (também vale para *threads*) acessam ou manipulam recursos compartilhados, como variáveis, estruturas de dados ou dispositivos de E/S. Essa seção precisa ser tratada com cuidado para evitar condições de corrida e resultados imprecisos. É necessário garantir que, dentro de uma seção crítica, apenas um processo por vez tenha acesso aos recursos compartilhados, o que é feito utilizando técnicas de sincronização.

Uma solução para o problema da seção crítica precisa satisfazer três requisitos:

1. **Exclusão mútua**. Se um processo estiver executando em sua seção crítica, nenhum outro processo poderá estar executando em suas seções críticas. Isso impede situações de condição de corrida, pois dois ou mais processos poderiam modificar uma variável compartilhada ao mesmo tempo, o que levaria a resultados imprevisíveis. Além disso, se eles estiverem lendo e escrevendo em um recurso compartilhado simultaneamente, os dados podem ficar inconsistentes, pois as operações podem ocorrer fora da ordem esperada.
2. **Progresso**. Se nenhum processo estiver executando em sua seção crítica e alguns processos quiserem entrar em suas seções críticas, então somente os processos que não estão executando em suas seções restantes poderão participar da decisão sobre qual entrará em sua seção crítica em seguida, e essa seleção não pode ser adiada indefinidamente.
3. **Espera limitada**. Existe um limite no número de vezes que outros processos podem entrar em suas seções críticas depois que um processo fez uma solicitação para entrar em sua seção crítica e antes que a solicitação seja concedida.

O código que implementa um sistema operacional (código do *kernel*) está sujeito a várias condições de corrida possíveis. Exemplos:

---

[1] *Deadlock* (*impasse*) é a situação que gera paralisação completa no sistema porque duas ou mais *threads* ou processos ficam bloqueadas indefinidamente, cada uma esperando que a outra libere um recurso necessário para continuar sua execução.

- Gerenciamento de arquivos e diretórios.
- Alocação de recursos.
- Tratamento de interrupção.

Há duas abordagens para lidar com a seção crítica em sistemas operacionais: *kernels preemptivos* e *kernels não preemptivos*. Um *kernel preemptivo* permite que um processo seja interrompido enquanto estiver sendo executado no modo *kernel*; já um *não preemptivo* não permite que um processo executando no modo *kernel* seja interrompido.

Um *kernel* não preemptivo está livre de condições de corrida nas estruturas de dados do *kernel*, pois somente um processo está ativo no *kernel* de cada vez. Não se pode dizer o mesmo sobre os *kernels* preemptivos, que precisam ser projetados cuidadosamente para garantir que os dados de *kernel* compartilhados sejam livres de condições de corrida.

No entanto, um *kernel* preemptivo pode ser mais responsivo, pois há menos risco de que um processo no modo *kernel* seja executado por um período longo antes de liberar o processador para os processos em espera (que podem *morrer de fome*). Além disso, um *kernel* preemptivo é mais adequado para programação em tempo real, pois permitirá que um processo em tempo real preempcione um processo atualmente em execução no *kernel*.

Suporte de Hardware para Sincronização

Nesta seção serão abordadas três operações de hardware que fornecem suporte para resolver o Problema da Seção Crítica: Barreiras de Memória, Instruções de Hardware e Variáveis Atômicas

Barreiras de Memória

As *Barreiras de Memória* (*Memory Barriers ou Memory Fences*) são instruções que controlam a reordenação de instruções de leitura e escrita de memória pelo processador. As barreiras estão presentes em algumas arquiteturas de computador para garantir a consistência e a ordem dos acessos à memória em sistemas multiprocessados.

Em ambientes multiprocessados, os processadores muitas vezes reordenam as instruções para otimizar o desempenho, mas isso pode fazer com que a ordem de leitura e escrita na memória não corresponda à ordem em que essas operações foram programadas pelo

software. *Memory Barriers* resolvem esse problema, garantindo que as operações de memória ocorram na sequência correta.

Um exemplo ocorre em processadores Intel, tais como Core 2 Duo, Atom, Core Duo, Pentium 4 e P6, que permitem que algumas operações sejam reordenadas. Esses processadores fornecem instruções de barreira de memória para carregamento e armazenamento (`MFENCE`), apenas de carregamento (*load fence*, `LFENCE`) e apenas de armazenamento (*store fence*, `SFENCE`). As instruções `LFENCE` e `MFENCE` estão disponíveis desde o processador Pentium 4, e a instrução `SFENCE` desde o Pentium III [2]. Já os processadores ARM dispõem da instrução DMB (*Data Memory Barrier*) para prevenir o reordenamento de instruções de *load* e *store* [3].

Um exemplo para a arquitetura ARM [3]:

```
LDR X0, [X1]  // Deve ser visto pelo sistema de memória antes da operação
              // STR abaixo.
DMB ISHLD     // Operação que aguarda apenas a conclusão dos loads (LDR) e
              // apenas para o domínio compartilhável interno
ADD X2, #1    // Deve ser executado antes ou depois do sistema de memória
              // ver LDR.
STR X3, [X4]  // Deve ser visto pelo sistema de memória depois do LDR
              // acima.
```

O pseudocódigo abaixo considera a existência de duas *threads* e ajuda a ilustrar o uso de barreiras de memória:

```
// Thread 1                          // Thread 2
(...)                                (...)
while (!flag)                        x = 100;
     memory_barrier();               memory_barrier();
print(x);                            flag = true;
(...)                                (...)
```

O código da *thread* 1 garante que o valor de `flag` será carregado antes que o valor de `x`. O código da *thread* 2 garante que a atribuição de `x` acontecerá antes da atribuição de `flag`.

Barreiras de Memória são consideradas operações de nível muito baixo e normalmente são usadas apenas por desenvolvedores de *kernel* ao escrever código especializado que garante a exclusão mútua.

Observação: *Barreira de Memória* (*Memory Barrier*) e *Barreira de Sincronização* (*Synchronization Barrier*) **não** são sinônimos.

Instruções de Hardware

Muitas arquiteturas modernas fornecem instruções de hardware especiais que permitem testar e modificar o conteúdo de uma palavra (`test_and_set()`) ou testar e trocar o conteúdo de duas palavras atomicamente (`compare_and_swap()`) – isto é, como uma unidade ininterrupta. Essas instruções podem resolver o Problema da Seção Crítica. Em vez de discutir uma instrução específica para uma máquina específica, abstraímos os detalhes e descrevemos as instruções `test_and_set()` e `compare_and_swap()`.

O importante na função `test_and_set()` é que ela é executada atomicamente. Assim, se duas instruções `test_and_set()` forem executadas simultaneamente (cada uma em um núcleo diferente), elas serão executadas sequencialmente em alguma ordem arbitrária. Se a máquina suportar a instrução `test_and_set()`, podemos implementar a exclusão mútua declarando uma variável booleana `lock`, com o valor `false`.

A instrução `compare_and_swap()`, assim como a instrução `test_and_set()`, opera em duas palavras atomicamente, mas usa um mecanismo diferente baseado na troca do conteúdo de duas palavras. A característica importante desta instrução é que ela é executada atomicamente. Assim, se duas instruções `compare_and_swap()` forem executadas simultaneamente (cada uma em um núcleo diferente), elas serão executadas sequencialmente em alguma ordem arbitrária.

Nas arquiteturas Intel x86, a instrução em linguagem assembly `cmpxchg` é usada para implementar a instrução `compare_and_swap()`. Para impor a execução atômica, o prefixo `lock` é usado para bloquear o barramento enquanto o operando de destino está sendo atualizado. A forma geral desta instrução é

```
lock cmpxchg <operando de destino>, <operando de origem>
```

Veja mais detalhes sobre instruções de hardware em [1].

Variáveis Atômicas

Normalmente, a instrução `compare_and_swap()` não é usada diretamente para fornecer exclusão mútua. Em vez disso, é usado como um bloco de construção básico para construir outras ferramentas que resolvam o problema da seção crítica. Uma dessas ferramentas é uma *variável atômica*, que fornece operações atômicas em tipos de dados básicos, como inteiros e booleanos. Como vimos, incrementar ou decrementar um valor inteiro pode produzir uma condição de corrida. Variáveis atômicas podem ser utilizadas para garantir a exclusão mútua em situações onde pode haver uma corrida de dados em uma única variável enquanto ela está sendo atualizada, como quando um contador é incrementado.

A maioria dos sistemas que suportam variáveis atômicas fornecem tipos de dados atômicos especiais, bem como funções para acessar e manipular variáveis atômicas. Essas funções geralmente são implementadas usando operações `compare_and_swap()`. Como exemplo, o seguinte incrementa o inteiro atômico `sequence`:

```
increment(&sequence);
```

E a função `increment` é implementada como segue:

```
void increment(atomic_int *v) {
      int temp;
      do {
            temp = *v;
      } while (temp != compare_and_swap(v, temp, temp+1));
}
```

As variáveis atômicas são comumente usadas em sistemas operacionais e em aplicativos simultâneos, embora seu uso geralmente seja limitado a atualizações únicas de dados compartilhados, como contadores e geradores de sequência. Embora isso pareça simples, o uso de variável atômica em conjunto com operações atômicas é suficiente para resolver o problema do exercício 5 da aula passada, já citado no início deste material. A nova solução está disponível no arquivo `pthread-exercicio-5-aula-4-solucao-atomica.c`.

Suporte de Software para Sincronização

As soluções baseadas em hardware apresentadas são complexas. Em vez de utilizá-las diretamente, é mais comum utilizar ferramentas de software de alto nível desenvolvidas por projetistas, tais como *mutexes*, *semáforos* e *monitores*.

Mutex

Mutex (abreviação de ***mut****ual* ***ex****clusion*) é um bloqueio ou uma trava (*lock*) usada para proteger seções críticas. O bloqueio *mutex* é considerado o mecanismo mais simples para sincronização em Programação Concorrente. A ação do *mutex* é, de fato, simples: o processo obtém a trava antes de entrar na seção crítica e libera a trava quando sai da seção crítica.

Uma analogia seria o acesso a uma sala para fazer um exame de ressonância magnética. Quando uma pessoa acessa a sala, a porta é trancada até que ela conclua o exame. Quando a porta é destrancada, a pessoa sai da sala e uma nova pessoa pode acessá-la. Além do indivíduo que realiza o exame, outras pessoas (*threads*) desejam acessar a máquina de ressonância que está na sala (recurso a ser compartilhado entre as pessoas), mas não conseguem porque há uma trava (a porta é trancada durante o exame). Esse mecanismo garante exclusão mútua ao acesso à sala de ressonância magnética.

A função `acquire()` adquire a trava e a função `release()` libera a trava do *mutex*. Um bloqueio *mutex* possui uma variável booleana cujo valor indica se o bloqueio está disponível ou não. Se o bloqueio estiver disponível, uma chamada para `acquire()` é bem-sucedida e o bloqueio é considerado indisponível. Um processo que tenta adquirir um bloqueio indisponível é bloqueado e, assim, precisa esperar até que o bloqueio seja liberado.

As funções `acquire()` e `release()` **precisam ser atômicas**. Portanto, *mutexes* podem ser implementados com CAS (*compare and swap*).

As funções `acquire()` e `release()` também podem ser chamadas de `lock()` e `unlock()`, respectivamente. A biblioteca `pthread`, por exemplo, usa as funções `pthread_mutex_lock` e `pthread_mutex_unlock` para essas tarefas [4].

Retornando ao exemplo de código em que uma variável compartilhada é incrementada por quatro *threads*, pode-se utilizar *mutex* para resolver o problema de sincronização. Para isso, basta realizar três ações:

1. Criar um *mutex*, uma estrutura do tipo `pthread_mutex_t`. Ele pode receber o valor `PTHREAD_MUTEX_INITIALIZER`, usado para inicializar o mutex com seus atributos padrão, tornando-o pronto para uso.
2. Inserir uma trava com `pthread_mutex_lock()` na função de incremento, antes de atualizar a variável; e
3. Inserir `pthread_mutex_unlock()` depois que a variável é atualizada, liberar o *mutex* e garantir que outras *threads* possam acessar a variável compartilhada de maneira segura, evitando conflitos e condições de corrida.

O código que demonstra a solução para o problema da variável compartilhada está disponível no arquivo pthread-exercicio-5-aula-4-solucao-mutex.c.

Semáforo

Semáforo é um mecanismo de sincronização criado por Edsger Dijkstra, publicado em seu artigo *Co-operating sequential processes* [6], em 1968. Esse mecanismo é representado por uma variável inteira que após sua inicialização só pode ser acessada pelas funções **`wait()` e `signal()`**[2]. Ele é um pouco mais sofisticado que o *mutex*. Na verdade, se o semáforo for *contador*, seu valor pode variar por um domínio menos restrito. Se o semáforo for *binário*, seu valor só pode ser *0* ou *1*, e seu comportamento se assemelha ao do bloqueio *mutex*.

Seja *S* um semáforo, as funções `wait()` e `signal()` podem ser da seguinte forma:

```
wait(S) {
      while (S <= 0); // Espera ocupada
      S--;
}

signal(S) {
      S++;
```

[2] É comum encontrar na literatura e em exemplos a função `wait()` sendo chamada de `P()` e a função `signal()` sendo chamada de `V()`. Isso se deve ao fato de Edsger Dijkstra ser holandês, assim, P significa *proberen* (tentar), e V significa *verhogen* (aumentar).

```
}
```

Essa é uma implementação simples em que ocorre a espera ocupada[3]. Há outras implementações possíveis em que um processo pode, em vez de ficar em espera ocupada (o que consome recursos), suspender-se. A operação de suspensão coloca um processo em uma fila de espera. Em seguida, o controle é transferido para o escalonador da CPU, que seleciona outro processo para execução. Nessa abordagem, o semáforo não é apenas um valor inteiro, mas uma estrutura (`struct` em C) que contém um valor inteiro e uma lista de processos em espera.

Todas as modificações do semáforo ocorrem nas operações `wait()` e `signal()`, que devem ser atômicas. Ou seja, quando um processo modifica o valor do semáforo, nenhum outro processo pode modificar simultaneamente o mesmo valor do semáforo. Além disso, no caso de `wait(S)`, o teste do valor inteiro de `S` ($S \leq 0$), bem como sua possível modificação (`S--`), deve ser executado sem interrupção.

Diversas aplicações computacionais fazem uso de semáforos para gerenciar recursos. Exemplos:

- Servidor Apache
- MySQL
- VirtualBox [7]
- Chromium [8]

Suponha que exista um aplicativo de edição de documentos de texto colaborativo, onde dez usuários podem compartilhar o acesso ao documento, mas somente três podem editá-lo simultaneamente. Cada usuário é representado por uma *thread*, e o documento é o recurso compartilhado.

Nesse cenário:

- **Número de *Threads***: dez.
- **Recurso Compartilhado**: um documento de texto.

[3] Também conhecido como *busy waiting*, é uma técnica de programação em que um processo ou *thread* fica em um *loop* enquanto espera por uma determinada condição ser atendida. Durante esse período, o processo verifica repetidamente se a condição desejada foi cumprida. Essa abordagem não é considerada eficiente, especialmente em situações em que a espera pode ser prolongada.

- **Semáforo Contador**: necessário para controlar o acesso simultâneo às seções de edição do documento. Precisa ser iniciado com um valor de *3*, permitindo que três *threads* editem seções diferentes do documento ao mesmo tempo.

Um exemplo de aplicação das operações do semáforo neste caso poderia ser o seguinte:

- *Thread* 1 (usuário 1) deseja editar uma parte do documento. Ela executa a operação `wait()` no semáforo, decrementando-o para *2* (*3 - 1*).
- *Thread* 2 (usuário 2) também deseja editar o documento. Ela executa a operação `wait()` no semáforo, decrementando-o para *1* (*2 - 1*).
- *Thread* 3 (usuário 3) também inicia sua edição no documento. Ela executa a operação `wait()` no semáforo, decremento-o para *0* (*1 - 1*). Isso significa que não há mais recursos disponíveis, ou seja, neste cenário não é permitido que mais usuários editem o documento.
- *Thread* 4 (usuário 4) também quer editar o documento, mas o semáforo está em *0*. Ela é bloqueada e espera até que um dos três usuários que estão editando o documento concluam suas ações, ou seja, aguarda até que o recurso seja liberado.
- *Thread* 1 (usuário 1) conclui sua edição. Ela, portanto, executa a operação `signal()` no semáforo, incrementando-o para *1* (*0 + 1*). Isso libera um recurso para outra *thread*.
- *Thread* 4 (usuário 4) pode começar sua edição. Ela executa a operação `wait()` no semáforo, decrementando-o para *0* (*1 - 1*). Novamente, no interesse de novos usuários, será necessário aguardar pela liberação de recursos.
- (...)

Dessa forma, o semáforo ajuda a controlar quantas *threads* podem editar o documento colaborativo simultaneamente, garantindo que a edição seja feita de maneira ordenada e evitando conflitos.

Voltando ao exemplo de aula, o problema de incremento da variável compartilhada usando semáforo foi resolvido com o auxílio da biblioteca `semaphore.h` e está disponível no arquivo `pthread-exercicio-5-aula-4-solucao-semaforo.c`. Note que foi utilizado um semáforo binário, e não um semáforo contador. Isso acontece porque se mais de uma *thread* acessar a variável ao mesmo tempo, poderão ocorrer acessos simultâneos a ela, ou seja, haverá condição de corrida. Assim, é necessário assegurar que no máximo uma thread por vez acesse o recurso.

Como sugestão, altere o código para que ele contenha um semáforo contador no lugar de um semáforo binário. Que alteração(ões) é(são) necessária(s)? Execute o código até que a resposta obtida seja diferente de `400`. Novamente, caso os resultados não estejam sendo diferentes, sugere-se aumentar o número de iterações, por exemplo, para 10.000. Neste caso, respostas diferentes de 40.000 indicam que condições de corrida ocorreram e assim produziram resultados inesperados.

A biblioteca `semaphore.h`

A biblioteca `semaphore.h` é parte do padrão POSIX e fornece a definição de um tipo e um conjunto de funções para trabalhar com semáforos. A seguir, descreve-se o tipo e as principais funções dessa biblioteca [5].

Tipo: `sem_t` é o tipo criado para representar um semáforo.
Funções:

- `sem_init`: inicializa um semáforo com um valor específico.
- `sem_wait`: aguarda até que o semáforo esteja disponível; quando estiver, o decrementa.
- `sem_post`: incrementa o semáforo, liberando-o para outros processos ou *threads*. É a operação `signal` descrita anteriormente.
- `sem_destroy`: destrói o semáforo liberando a memória alocada para ele.

Mais informações estão disponíveis na documentação de `semaphore.h`.

Monitor

Tanto *mutexes* quanto semáforos possuem um problema: dependem do uso correto do programador para garantir que seções críticas não sejam acessadas indevidamente. Por exemplo, o uso invertido das funções `wait()` e `signal()` em um semáforo pode resultar em condições de corrida e resultados não desejados. O monitor é uma solução de sincronização de alto nível que lida melhor com essa questão.

O monitor é um tipo abstrato de dados que encapsula dados com um conjunto de funções para operar nesses dados que são independentes da implementação do monitor. O acesso aos recursos compartilhados é gerenciado por entradas de monitor, que garantem a exclusão mútua. Apenas um processo ou *thread* pode estar dentro do monitor a qualquer

momento, assim o programador não precisa codificar explicitamente essa restrição de sincronização.

O monitor consiste, portanto, de três partes [9]:

1. Dados compartilhados de uma instância do monitor.
2. Funções que implementam operações do monitor.
3. Código de inicialização.

Devido ao formato em que essa construção é proposta, pode-se dizer que o monitor se assemelha à ideia de orientação a objetos. A Figura Monitor.png ilustra esse aspecto mostrando as partes do monitor.

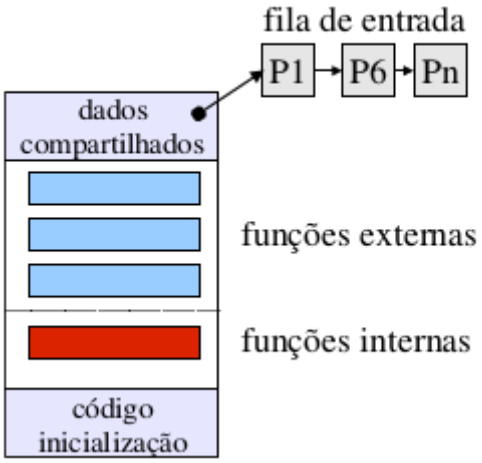

Figura Monitor.png [9]. Disponível em: https://drive.google.com/file/d/1ZuI3v7H9BGZoTHpI11vyE_po40RNVsU_/view?usp=drive_link.

Exemplos de uso de Monitores:

- **Bancos de dados**: Monitores são usados para controlar o acesso a bancos de dados, garantindo que várias transações não interfiram umas nas outras.
- **Linguagens de programação**: Java utiliza monitores para a sincronização de *threads*, garantindo acesso seguro a recursos compartilhados [10].
- **Controle de impressoras**: monitores podem ser usados para controlar a impressão concorrente em impressoras compartilhadas.

Pode-se dizer que uma desvantagem ao utilizar monitores é o fato de que eles devem ser implementados como parte da linguagem de programação. Isso significa que o compilador precisa gerar código para eles, o que, na prática, gera um custo adicional de ter que saber quais recursos do sistema operacional estão disponíveis para controlar o acesso a seções críticas em processos simultâneos [11].

Monitores em C

Internamente, monitores podem ser implementados tanto utilizando *mutexes* quanto semáforos. Há linguagens que fornecem uma construção nativa que implementa monitores, tais como Java[4] e Python. Infelizmente, esse não é o caso da linguagem C, que exige a criação manual de monitores através de primitivas de sincronização. Isso significa que, ao programar em C, a criação e o gerenciamento de monitores são tarefas mais complexas e propensas a erros, se comparadas às linguagens que oferecem suporte nativo.

Observe o código do arquivo `monitor-exemplo-1.c`. Este código em C implementa um monitor simples para sincronização de *threads* usando *mutexes* e variáveis de condição. A ideia do código é apresentar a criação do monitor como um tipo abstrato de dados, o que é feito em C através da construção de uma estrutura (`struct`). A estrutura `Monitor` encapsula um *mutex*, uma variável de condição e um contador para controlar a entrada e saída das *threads* no monitor. A função `iniciarMonitor` inicializa os elementos do monitor, enquanto `encerrarMonitor` os encerra, destruindo *mutexes* e variáveis de condição. A função `entrarMonitor` bloqueia o *mutex*, permitindo que uma *thread* entre no monitor e aguarde se houver outras *threads* presentes. Quando uma *thread* sai do monitor usando `sairMonitor`, ela sinaliza outras *threads* que podem estar esperando na variável de condição para entrar. Dentro da função `main`, um monitor é criado e inicializado. Uma *thread* entra no monitor, imprime uma mensagem e sai dele. Após isso o monitor é encerrado.

A partir do código fornecido, criou-se o arquivo `monitor-exemplo-2.c`, este sim com mais *threads* para demonstrar a concorrência e a sincronização em funcionamento em um exemplo hipotético em que diversos usuários (*threads*) compartilham um mesmo recurso (neste caso, um banheiro). No código, cada *thread* simula a ação de entrar no banheiro, utilizar por um tempo definido (meio segundo, representado pela função `usleep()`), e então sair do banheiro, garantindo que apenas uma *thread* possa utilizá-lo por vez, evitando condições de corrida e garantindo a exclusão mútua. A estrutura de monitor implementada usando *mutex* e variável de condição garante a sincronização adequada entre as *threads* para controlar o acesso ao recurso compartilhado.

[4] Verifique exemplos de uso de monitores em Java no Apêndice I.

Agora, voltando ao problema em que quatro *threads* incrementam uma variável compartilhada, nota-se que ele pode ser resolvido com um monitor, como mostra o arquivo pthread-exercicio-5-aula-4-solucao-monitor.c.

O Jantar dos(as) Filósofos(as)

O problema do Jantar dos(as) Filósofos(as) é um problema clássico de sincronização em que há, em uma mesa de jantar circular, cinco filósofos(as). Cada filósofo(a) possui um prato e ao lado de cada prato há um talher. Para que um(a) filósofo(a) possa comer, é necessário utilizar dois talheres, o que não será possível se pelo menos um dos filósofos ao lado não estiver comendo. A Figura abaixo ilustra a situação descrita:

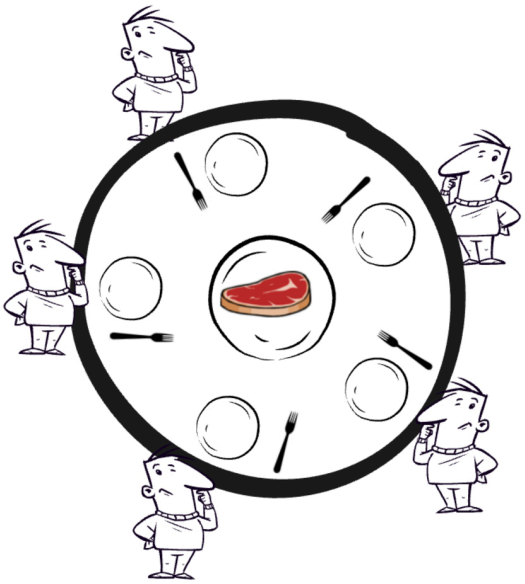

Figura Jantar dos filósofos. Disponível em: https://deinfo.uepg.br/~alunoso/2021/SO/Filosofos_Java/. Acesso em: 29 ago. 2023.

O filósofo deve devolver os talheres imediatamente após comer, momento em que ele vai filosofar. É necessário impedir que os filósofos adjacentes peguem um talher ao mesmo tempo, evitando assim a ocorrência de *deadlock*.

O problema pode ser resolvido utilizando as estratégias vistas em aula. Veja algumas soluções nos arquivos

- jantar-mutex.c
- jantar-semaforo.c
- jantar-monitor.c

Memórias Transacionais

Tópico a ser abordado futuramente.

Resumo

Esta aula abordou a sincronização entre processos e *threads*. Foram explorados os conceitos de condição de corrida e seção crítica, bem como estratégias para mitigar a ocorrência de condições de corrida, tipicamente divididas em soluções de hardware e software.

Entre as principais operações de hardware para sincronização estão as barreiras, as instruções de hardware e as variáveis atômicas. *Mutexes*, semáforos e monitores foram as soluções baseadas em software detalhadas no texto desta aula para sincronização, o que foi apresentado também através de exemplos utilizando a linguagem C.

Exercícios

1) Pesquise e resuma o que é a *Solução de Peterson* (ou *Algoritmo de Peterson*) e como ela está relacionada com o Problema da Seção Crítica.

2) (FAURGS – 2018 – TJ-RS – Administrador de Banco de Dados) A expressão *condição de corrida* (*race condition*) refere-se a situações de processamento que podem levar os processos a fornecer resultados errôneos da computação. Essas condições de corrida ocorrem

a) quando os processos rodam em máquinas cujas diferenças de velocidades de processamento forem maiores do que cos(45°) (aproximadamente 70%).
b) apenas quando 3 (três) ou mais processos estão envolvidos no acesso a variáveis compartilhadas.
c) sempre que são utilizados semáforos para proteger as seções críticas dos processos.
d) quando os processos manipulam variáveis compartilhadas de forma concorrente.
e) apenas, quando os processos estão em *deadlock*, causado pelo mecanismo de proteção das seções críticas.

3) (POSCOMP 2008, Questão 52) Analise as seguintes afirmativas.

I. Condições de corrida podem ocorrer se múltiplas *threads* fazem leituras de um dado compartilhado, mesmo que nenhuma realize escritas.

II. O uso de *mutex* para a exclusão mútua em seções críticas garante que não haja condição de corrida, porém pode ocasionar *deadlocks* se não for corretamente empregado.

III. Monitores são baseados em um tipo abstrato de dados e um controle de acesso aos dados. Apenas funções do monitor acessam os dados e apenas uma *thread* ou processo pode executar funções de um monitor por vez.

IV. Semáforos têm duas operações, `P()` e `V()`, sendo que apenas a operação `P()` pode bloquear um processo ou *thread*.

A análise permite concluir que

a) apenas as afirmativas I, II e III são verdadeiras.
b) apenas as afirmativas I, III e IV são verdadeiras.
c) apenas as afirmativas II e IV são verdadeiras.
d) apenas as afirmativas II, III e IV são verdadeiras.
e) nenhuma das afirmativas é verdadeira.

4) (Marinha – 2013 – Primeiro Tenente – Informática) Uma coleção de rotinas, de variáveis e de estruturas de dados, agrupados em um tipo especial de módulo ou pacote que permite que, em um dado momento, somente um processo ativo execute um de seus procedimentos, implementando, dessa forma, a exclusão mútua, é denominada

a) semáforo.
b) *mutexes*.
c) instrução TEST AND SET LOCK.
d) monitor.
e) BUSY WAITING.

5) (**EXTRA**) Suponha que uma empresa de Segurança da Informação guarde seus principais segredos em uma sala-cofre. A empresa dispõe de quatro guardas armados que são as únicas pessoas autorizadas a entrar e inspecionar a sala. Para isso, precisam abrir e fechar a sala-cofre. Um mesmo guarda pode abrir e depois fechar a sala, mas devido ao peso da porta, pode acontecer de um guarda abrir e outro guarda fechá-la. Obviamente, um guarda não pode abrir uma sala que já está aberta, assim como uma sala com a porta fechada não pode ser fechada. Utilizando monitor, crie um código em C para monitorar infinitamente a abertura e o fechamento da porta da sala-cofre. Você deve indicar qual *thread* está abrindo/fechando a porta.

Dica: utilize `pthread_cond_broadcast` para informar as *threads* que uma determinada condição foi cumprida, permitindo que elas continuem a execução.

Gabarito

1) A solução de Peterson é um algoritmo de sincronização que resolve o Problema da Seção Crítica em um ambiente simples onde existem apenas dois processos (P_0 e P_1) que compartilham recursos. O algoritmo cumpre os três requisitos para solucionar o problema: exclusão mútua, progresso e espera limitada.

   A ideia do algoritmo é usar variáveis de turno (*turn*) e de controle (*flags*) para coordenar a entrada dos processos em suas seções críticas. As variáveis de turno determinam qual processo tem a prioridade para entrar em sua seção crítica, enquanto as *flags* indicam o desejo de um processo de entrar em sua seção crítica.

   O algoritmo de Peterson funciona da seguinte maneira:
   1. Cada processo define sua *flag* para indicar seu desejo de entrar na seção crítica.
   2. Cada processo atribui o turno ao outro processo, indicando que é a vez do outro processo entrar em sua seção crítica.
   3. O processo verifica se é a sua vez e se o outro processo deseja entrar na sua seção crítica (*flag*).
   4. Se ambas as condições forem verdadeiras, ocorre uma espera ativa até que a vez do processo chegue ou a flag do outro processo seja retirada.
   5. Quando um processo sai de sua seção crítica, ele redefine sua flag e passa o turno para o outro processo.

   A solução de Peterson não é escalável para um número maior de processos e não é apropriada para arquiteturas modernas, pois muitas operações são implementadas sem atomicidade, além de poderem ser reordenadas pelo processador. Outro aspecto que influencia na inapropriação desse algoritmo é o uso de *caches*, que podem resultar em inconsistências ao acessar diferentes cópias de variáveis.
2) d
3) d
4) d
5) Resposta possível no arquivo `exercicio-extra-monitor.c`.

Referências

[1] SILBERSCHATZ, A., Galvin, P. B., & Gagne, G. (2018). Operating System Concepts (10ª ed.). Wiley.

[2] INTEL. Intel(R) 64 and IA-32 Architectures Software Developer's Manual. Documentation changes. 2023. Disponível em: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html. Acesso em: 22 ago. 2023.

[3] ARM DEVELOPER. Learn the architecture - ARMv8-A memory systems. Barriers. Disponível em: https://developer.arm.com/documentation/100941/0101/Barriers. Acesso em: 22 ago. 2023.

[4] LINUX MAN PAGES. pthreads(7) - Linux manual page. Disponível em: https://man7.org/linux/man-pages/man7/pthreads.7.html. Acesso em: 23 ago. 2023.

[5] THE OPEN GROUP. (2018). The Open Group Base Specifications, Issue 7, 2018 edition. semaphore.h - The Open Group Base Specifications Issue 7, 2018 edition. Disponível em: https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/semaphore.h.html. Acesso em: 24 ago. 2023.

[6] DIJKSTRA, E. W. Co-operating sequential processes. In Programming languages : NATO Advanced Study Institute : lectures given at a three weeks Summer School held in Villard-le-Lans, 1966 / ed. by F. Genuys (pp. 43-112). Academic Press Inc. 1968. Disponível em: https://pure.tue.nl/ws/files/4279816/344354178746665.pdf. Acesso em: 25 ago. 2023.

[7] VIRTUALBOX. Código fonte do arquivo ntsem.c. Disponível em: https://www.virtualbox.org/svn/vbox/trunk/src/libs/xpcom18a4/nsprpub/pr/src/md/windows/ntsem.c. Acesso em: 25 ago. 2023.

[8] CHROMIUM. Código fonte do arquivo ntsem.c. Disponível em: https://chromium.googlesource.com/chromium/deps/nss/+/702e3efe060266601e5e4d619388235ce47c7b22/nspr/pr/src/md/windows/ntsem.c. Acesso em: 25 ago. 2023.

[9] PILLA, M. L. Material didático. Concorrência. Versão 2011-03-29.

[10] ORACLE. The Java™ Tutorials. Intrinsic Locks and Synchronization. Disponível em: https://docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html. Acesso em: 28 ago. 2023.

[11] MCCLANAHAN, P. Operating System: The Basics. San Joaquin Delta College, Eng Libretexts. 2020. Disponível em: https://eng.libretexts.org/Courses/Delta_College/Operating_System%3A_The_Basics. Acesso em: 28 ago. 2023.

[12] EUGEN GITHUB. https://github.com/eugenp/. Acesso em: 30 ago. 2023.

Apêndice I

Monitores em Java

Exemplo de código que resolve o problema de incremento de variável compartilhada em Java:

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class MonitorExemplo1 {
    private static final int NUM_THREADS = 4;
    private static final int NUM_ITERATIONS = 100;

    static class Monitor {
        private Lock mutex = new ReentrantLock();
        private int variavelCompartilhada = 0;
    }

    static void incrementaVariavelCompartilhada(Monitor monitor) {
        monitor.variavelCompartilhada++;
    }

    static class IncrementaThread implements Runnable {
        private Monitor monitor;

        IncrementaThread(Monitor monitor) {
            this.monitor = monitor;
        }

        @Override
        public void run() {
            for (int i = 0; i < NUM_ITERATIONS; i++) {
                monitor.mutex.lock();
                try {
                    incrementaVariavelCompartilhada(monitor);
                } finally {
                    monitor.mutex.unlock();
                }
            }
        }
```

```
    }

    public static void main(String[] args) throws InterruptedException {
        Thread[] threads = new Thread[NUM_THREADS];
        Monitor meuMonitor = new Monitor();

        for (int i = 0; i < NUM_THREADS; i++)
            threads[i] = new Thread(new IncrementaThread(meuMonitor));

        for (int i = 0; i < NUM_THREADS; i++)
            threads[i].start();

        for (int i = 0; i < NUM_THREADS; i++)
            threads[i].join();

        System.out.println("Valor final da variável compartilhada: " +
meuMonitor.variavelCompartilhada);
    }
}
```

Compile o código com:

> `javac MonitorExemplo1.java`

Execute o código com:

> `java MonitorExemplo1`

Exemplo de código que simula o uso de monitor para sincronizar o acesso a uma impressora compartilhada. Um processo para impressões é criado e possui duas *threads*, com cada uma sendo responsável por imprimir um dos textos. No entanto, há controle de concorrência garantindo que a impressora só imprimirá um dos textos por vez.

```
import java.lang.management.ManagementFactory;
import java.lang.management.ThreadMXBean;

class ImpressoraCompartilhada {
    synchronized public void imprimir(String processo, String texto) {
        System.out.println(processo);
        System.out.println("*** Início da impressão ***");
        for (int i = 0; i < texto.length(); i++) {
            System.out.print(texto.charAt(i));
            try {
                Thread.sleep(200);
```

```
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
        System.out.println("*** Fim da impressão ***");
    }
}

class Trabalho extends Thread {
    private ImpressoraCompartilhada impressora;
    private String processo;
    private String texto;

    public Trabalho(ImpressoraCompartilhada impressora, String processo,
String texto) {
        this.impressora = impressora;
        this.processo = processo;
        this.texto = texto;
    }

    public void run() {
        impressora.imprimir(processo, texto);
    }
}

public class MonitorExemplo2 {
    public static void main(String[] args) {
        ImpressoraCompartilhada impressora = new ImpressoraCompartilhada();

        ThreadMXBean threadMXBean = ManagementFactory.getThreadMXBean();
        long processoId =
threadMXBean.getThreadCpuTime(Thread.currentThread().getId()) / 1000000;

        long threadId1 = Thread.currentThread().getId();
        long threadId2 = threadId1 + 1;
        String processo1 = "O trabalho " + processoId + "_" + threadId1 + "
está imprimindo o texto:";
        String processo2 = "O trabalho " + processoId + "_" + threadId2 + "
está imprimindo o texto:";
        String texto1 = "Blumenau, 14 de maio de 2023.\nQuerido diário, hoje
eu estou cansado.\n";
```

```
        String texto2 = "PROVA DE PROGRAMAÇÃO DE ALTO
DESEMPENHO\nNOME:\nDATA:\n...\n";

        Trabalho t1 = new Trabalho(impressora, processo1, texto1);
        Trabalho t2 = new Trabalho(impressora, processo2, texto2);
        t1.start();
        t2.start();

        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}
```

Compile o código com:

> `javac MonitorExemplo2.java`

Execute o código com:

> `java MonitorExemplo2`

Exemplo de solução para o problema do Jantar dos(as) Filósofos(as), adaptado de [12]:

Arquivo `Filosofo.java`:

```
public class Filosofo implements Runnable {

    private final Object esqTalher;
    private final Object dirTalher;

    Filosofo(Object esq, Object dir) {
        this.esqTalher = esq;
        this.dirTalher = dir;
    }

    private void fazAlgumaCoisa(String action) throws InterruptedException
{
            System.out.println(Thread.currentThread().getName() + " " +
action);
        Thread.sleep(((int) (Math.random() * 100)));
    }
```

```
    @Override
    public void run() {
        try {
            while (true) {
                     fazAlgumaCoisa(System.nanoTime() + ": Pensando."); //
thinking
                synchronized (esqTalher) {
                     fazAlgumaCoisa(System.nanoTime() + ": Pega o talher da
esquerda.");
                    synchronized (dirTalher) {
                        fazAlgumaCoisa(System.nanoTime() + ": Pega o talher
da direita - Come!"); // eating
                                fazAlgumaCoisa(System.nanoTime() + ": Larga o
talher da direita.");
                    }
                    fazAlgumaCoisa(System.nanoTime() + ": Larga o talher da
esquerda. Volta a pensar.");
                }
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}
```

Arquivo `Jantar.java`:

```
public class Jantar {

    public static void main(String[] a) throws Exception {

        Filosofo[] filosofos = new Filosofo[5];
        Object[] talheres = new Object[filosofos.length];

        for (int i = 0; i < talheres.length; i++) {
            talheres[i] = new Object();
        }

        for (int i = 0; i < filosofos.length; i++) {

            Object esqTalher = talheres[i];
```

```
            Object dirTalher = talheres[(i + 1) % talheres.length];

            if (i == filosofos.length - 1) {
                // O último Filósofo pega o talher da direita primeiro
                filosofos[i] = new Filosofo(dirTalher, esqTalher);
            } else {
                filosofos[i] = new Filosofo(esqTalher, dirTalher);
            }

            Thread t = new Thread(filosofos[i], "Filosofo " + (i + 1));
            t.start();
        }
    }
}
```