Sistemas Distribuídos

COS470 - Lista 2

Pedro Maciel Xavier
116023847

22 de julho de 2021

Questão 1

Resposta

A comunicação entra processos pode se dar através do uso de memória compartilhada ou de algum mecanismo de troca de mensagens. Dentro destas duas categorias existe ainda um monte de técnicas diferenetes.

Em uma situação do uso de memória compartilhada, uma região da memória poderá ser acessada pelos processos que a compartilham através das operações usuais de leitura e escrita em seus endereços. Antigamente, isso costumava ser feito utilizando arquivos. Alguns sistemas operacionais implementavam um tipo especial de arquivo para este propósito.

A troca de mensagens pode ser dar de maneira ainda mais diversa, desde a simples troca de sinais, passando pelo uso de *traps* do sistema operacional, considerando os *Pipes* e até mesmo os *Sockets*. O que estes métodos apresentam em comum é o papel do sistema operacional como intermediário na comunicação.

Utilizar memória compartilhada entre processos, em geral, torna a execução mais rápida do que a maioria dos meios de troca de mensagens, uma vez que chamadas ao sistema operacional não são necessárias. Esta abordagem, no entanto, incorre em problemas de escalabilidade além de não ser aplicável a um contexto de redes. Uma vantagem da troca de mensagens é a possibilidade de se construir programas sobre este paradigma que operam remotamente. A depender da técnica utilizada, pode-se encara maior ou menor dificuldade de implementação. Se a exigência não for tremenda, o simples uso de sinais pode ser suficiente para estabelecer a comnuicação desejada.

Questão 2

Resposta

	P2 (`read`) bloqueante	P2 (read) não-bloqueante
P1 (`write`) bloqueante	Quando ambos são bloqueantes, a comunicação se dá de maneira sincronizada, intercalando chamadas `read` e `write`. P1 sempre aguardará que o conteúdo do *Pipe* seja lido antes de escrever. P2 sempre irá esperar até que alguma informação seja escrita no *Pipe* antes de realizar a leitura.	Quando somente a leitura é não-bloqueante, pode ser que P2 leia repetidas vezes antes de uma troca de contexto, levando à situação em que o *Pipe* encontra-se vazio. Nesse caso, a depender da implementação, a chamada `read` terá valor de retorno indicativo da falha na leitura e, portanto, é de responsabilidade do programador lidar com este cenário.
P1 (`write`) não-bloqueante	A escrita não-bloqueante permite que a operação se repita sucessivas vezes antes de uma troca de contexto. Com isso, a capacidade máxima do *Pipe*, que depende da implementação, pode ser atingida. Isso permite que a atomicidade da escrita seja violada e parte da mensagem seja entregue de maneira fragmentada.	Quando ambas leitura e escrita se dão de maneira não-bloqueante, teremos que enfrentar os cenários descritos nos dois casos.

Questão 3

Resposta

Criar um sistema *multi-threaded* é imediatamente vantajoso do ponto de vista dos recursos necessários, quando comparado a um sistema *multi-processed*. Neste último, é necessário criar um novo espaço de memória com todas as informações relativas ao processo para cada linha de execução necessária. Múltiplas *threads* dentro de um mesmo processo vão utilizar o mesmo espaço de endereçamento. Isso nos leva a uma outra vantagem do uso de *threads* em vez de processos: a comunicação. *Threads* podem simplesmente se comunicar através da memória que compartilham, enquanto processos distintos dependem de outros mecanismos como Sinais, *Sockets* e *Pipes*.

Questão 4

Resposta

Em geral, *user-level threads* não dispõem dos recursos de paralelismo que múltiplos núcleos de processamento podem oferecer. O Sistema Operacional não é capaz de fazer distinção entre as *threads* e, portanto, deve designá-las ao mesmo núcleo de maneira sequencial, mesmo que alternada.

No entanto, se considerarmos um computador onde são executados diversos processos simultâneamente, o fato de ser multi-processado pode garantir uma menor disputa entre o processo do sistema em questão e os demais, fazendo com que ele permaneça por mais tempo em execução.

Questão 5

Resposta

Uma *condição de corrida* é a situação onde o comportamento de um programa pode ser influenciado pela execução concorrente de outros programas com os quais compartilha recursos. É muito comum que *condições de corrida* causem incosistência nos dados do programa, o que pode gerar decisões equivocadas guiadas por condicionais, *deadlocks*, perda definitiva de informações em bancos de dados dentre outros problemas.

```
1 int n;
2 ...
3 void f() {
4     while (1) n = n + 1;
5 }
6
```

No código em questão, um simples exemplo ilustrativo, vamos supor que a função `f` é executada por diferentes *threads* e que o incremento é realizado de maneira fragmentada, não-atômica. Pode ser que ocorra uma interrupção no momento em que uma delas acabou de acessar do valor da variável `n`. Esse valor foi, portanto, salvo em um registrador. Então, outra *thread* passa a incrementar repetidamente o valor da variável. Quando retorna a *thread* em questão, ela utiliza o valor inconsistente do registrador para atualizar a variável, desconsiderando uma série de alterações anteriores.

Questão 6

Resposta

Se uma das *threads* fizer uma chamada à função `acquire` num momento próximo à troca da linha de execução, ela pode acabar declarando interesse em entrar na região crítica, ou seja, executando a instrução

```
lock->interested[this_thread] = 1
```

e, em seguida, saindo da CPU. Uma outra *thread* que em sua vez de executar faça uma chamada à função `acquire` estará cometendo um erro terrível: irá declarar interesse em adentrar a região crítica e logo entrará em espera, pois já existe outra *thread* interessada. Terminado o seu tempo de execução, a *thread* anterior, ao retornar à CPU, deparar-se-à com o interesse da outra *thread*. Resultado: ambas para sempre esperando que a

outra não esteja mais interessada.

Questão 7

Resposta

Podemos contruir locks da seguinte forma:

```
1  void swap(bool *a, *b) {
2      bool temp = *a;
3      *a = *b;
4      *b = temp;
5  }
6
7  struct lock {
8      bool x = false;
9  }
10
11 void acquire(lock) {
12     bool x = true;
13     while (x) swap(&lock.x, &x);
14 }
15
16 void release(lock) {
17     lock->x = false;
18 }
```

Quando uma *thread* realizar a chamada `acquire`, ela vai executar as linhas 12 e 13, substituindo o valor armazenado na estrutura `lock` pelo valor de `x`, que pertence à *thread*. Como este valor era inicialmente `false`, a troca permite que a *thread* saia da espera e adentre a região crítica. Outra *thread* que tome a mesma atitude irá encontrar o valor de *Lock* verdadeiro e não poderá seguir adiante. Quando a *thread* original fizer uma chamada à função `release`, o valor da *Lock* permitirá que outra *thread* passe o valor falso para sua variável de condição do loop.

Mesmo que ocorra uma troca de contexto após a chamada ao `swap` e antes da verificação de condição do loop, o valor da *Lock* não poderá ser lido como falso por mais de uma *thread*, evitando assim uma condição de corrida.

Questão 8

Resposta

No código apresentado, podemos chamar R_1 a região crítica encapsulada pelo semáforo s_1 e R_2 aquela determinada pelo semáforo s_2. Como os semáforos são binários teremos, no máximo, uma *thread* dentro de uma determinada região. Percebe-se que $R_2 \subset R_1$, ou seja, para executar código em R_2, uma *thread* necessariamente deve estar em R_1. Como o acesso ao código em R_1 é sempre exclusivo à uma única linha de execução, o código de R_2 também será. Independente do número de *threads*, a saída poderá ser descrita pela expressão regular `(ab)*`.

Questão 9

Resposta

Realizar a chamada `wait(mutex)` antes da chamada `wait(buffer)` pode levar a um cenário onde um produtor fica na região de exclusão mútua, solitário, aguardando a liberação de um *buffer*. Os consumidores não conseguirão entrar na região crítica, tampouco sair. Dessa maneira, nenhum *buffer* será liberado e o sistema permanecerá congelado indeterminadamente.

Questão 10

Resposta

A chamada `signal` associada ao semáforo serve para incrementar o valor do semáforo permitindo que, caso o valor se torne positivo, alguma *thread* que tenha realizado a chamada `wait` possa enfim decrementar o valor do mesmo.

A semântica da chamada no caso dos monitores, por sua vez, é responsável por alterar o estado das *threads* que chamaram *wait* para *ready*, a depender da variável de condição associada.

Questão 11

Resposta