Lista 2 - Sistemas Distribuídos

Thiago Guimarães - DRE: 118053123

2021.1

1

As 2 principais abordagens são Memória Compartilhada e Troca de Mensagens. A primeira consiste em fornecer aos processos uma região de memória comum, fora do kernel space. Na segunda, há um canal de comunicação explícito que recebe chamadas send/receive, sendo uma fila de mensagens em kernel space.

1.1 Memória Compartilhada

1.1.1 Vantagens:

Como temos um único espaço de memória compartilhado entre dois processos, só é necessária a alocação deste espaço uma única vez, ou seja, só é necessária uma system call para preparar a região de memória compartilhada. Além disso, um programa vê esta memória exatamente da mesma forma que vê uma memória normal. Outra vantagem é que ler e escrever na memória compartilhada é mais rápido em comparação com outros mecanismos IPC.

1.1.2 Desvantagens:

Uma desvantagem é o fato de que passa a ser necessária uma coordenação dos processos para que não ocorram problemas relacionados a escrita e leitura neste espaço de memória compartilhada.

1.2 Troca de Mensagens

1.2.1 Vantagens:

De vantagem, temos que não é necessário coordenar os processos, dado que a fila de mensagens funciona em kernel space e o próprio SO faz essa coordenação entre os processos.

1.2.2 Desvantagens:

Uma desvantagem é que sempre que uma mensagem precisar ser enviada/lida, é necessária a realização de system calls.

2

2.1 Read não-bloqueante e write bloqueante:

No read não bloqueante, a chamada read(pipe, m) tentará ler o receiving end do pipe. Se o receiving end estiver vazio, ou seja, ainda não tiver ocorrido nenhuma escrita ou todo o conteúdo do pipe já tiver sido consumido, o valor do parâmetro m poderá ser setado para null. O write bloqueante implica apenas que, para que ocorra o write, é necessário que haja espaço no pipe para tal, logo, a chamada write ficará em espera até que esta condição seja alcançada, não interferindo no parâmetro m.

2.2 Read bloqueante e write não-bloqueante:

No read bloqueante, a leitura do pipe é realizada somente quando há conteúdo. O processo fazendo read fica esperando até que está condição seja alcançada, assim, o parâmetro m nunca será setado para null, mas sim com o valor no início da fila do pipe. O write não-bloqueante implica que o processo que faz write não ficará esperando que o pipe tenha espaço para escrita para escrever. Assim, quando o write tentar escrever no pipe cheio, esta informação será perdida, de forma a não se refletir futuramente no parâmetro m.

2.3 Read bloqueante e write bloqueante:

Ocorrerá o comportamento esperado. A escrita acontecerá somente quando houver espaço para escrita no pipe e a leitura somente quando houver algo para ser lido. Assim, o parâmetro m nunca será null e não haverá perda de informação.

2.4 Read não-bloqueante e write não-bloqueante:

Todos os cenários anteriores podem acontecer. O parâmetro m poderá ser null (read não bloqueante) e poderá ocorrer a perda de informação (write não bloqueante).

3

Uma das vantagens de um sistema multi-threaded é que um espaço de memória comum é utilizado, diferentemente do multi-processed, onde cada processo possui seu próprio espaço de memória. Outra vantagem é que para aumentar o poder computacional, os sistemas multi-threaded podem criar N threads dentro de um mesmo processo, quando nos sistemas multi-processed é necessário adicionar CPUs para aumentá-lo. Além disso, também há a vantagem de que threads são muito mais leves que processos, dado que o Kernel mantém bem menos estado. Uma possível desvantagem é o fato do desenvolvimento do sistema tornar-se potencialmente mais complexo.

4

Como a maioria das respostas na computação, depende. O sistema, por ser multiprocessado, consegue executar múltiplos processos em paralelo. Assim, mesmo que nosso sistema baseado em user-level threads seja visto como 1 único processo pelo SO, o fato do computador ser multiprocessado pode diminuir a chance do processo ser desescalonado, aumentando o desempenho . Contudo, vale ressaltar que esta possibilidade de melhora de desempenho não é consequência do fato do sistema ser baseado em user level threads. User level threads não são visíveis ao SO, ou seja, o SO não conseguiria rodar 2 user level threads em paralelo, dado que, para o SO, o sistema multi-threaded em questão é constituído por apenas 1 única kernel level thread. Porém, é válido ressaltar que, se tivéssemos as CPU's 100% disponíveis para esse único sistema, o desempenho não aumentaria, pois as threads de nível de usuário não executam paralelamente em vários núcleos, logo não tiram vantagem de uma CPU com múltiplos cores.

5

Uma condição de corrida é uma falha num programa em que o resultado do programa depende da sequência ou sincronia de eventos, podendo ser ou não um bug no sistema. Um exemplo pode ser dado a partir da função:

```
retirada(conta, valor) {
    saldo = get_saldo(conta);
    saldo = saldo - valor;
    put_saldo(conta, saldo);
    retorna saldo;
}
```

Nesta função, se a conta tivesse inicialmente 100 reais e duas pessoas executassem-na paralelamente, retirando cada uma 20 reais, seria possível que as duas, ao realizar a chamada get_saldo(conta), carregariam o valor 100 reais. Ao final da execução das duas funções, ficaríamos portanto com 80 reais na conta, quando na verdade deveríamos ficar com apenas 60 reais na conta.

6

Pode ocorrer uma troca de contexto quando o valor de interested da thread é definido para 1 ao chegar no loop na sequência. Assim, as duas threads ficarão no loop, esperando a outra perder o interesse, sendo então um deadlock.

7

```
struct lock {
    bool held = false;
}

void acquire(lock) {
    bool *b = true;
    while (*b)
        swap(&lock.held, *b);
}

void release(lock) {
    &lock.held = false;
}
```

8

8.1 Assumindo que os dois semáforos começam abertos:

"abababababab..." será o padrão impresso, não ocorrendo deadlock, pois todas as threads que tentarem executar algo serão embarreiradas pelo semáforo s1.

8.2 Assumindo que o semáforo s1 começa aberto e o s2 começa fechado:

"a" será o padrão impresso, apenas uma vez. A primeira thread executará até o wait(s2), aonde ficará pra sempre esperando, enquanto as outras threads ficarão em wait(s1), sendo então um deadlock.

8.3 Assumindo que o semáforo s1 começa fechado e o s2 começa fechado ou aberto:

Aqui, não haverá padrão impresso, pois todas as threads ficarão em wait(s1), configurando deadlock.

9

```
producer {
        while(1) {
                Produce new resource;
                wait(mutex); // lock buffer list
```

```
                    wait(empty); // wait for empty buffer
                    Add resource to an empty buffer;
                    signal(mutex); // unlock buffer list
                    signal(full);  // note a full buffer
    }
}
```

Dado isso, o que pode acontecer é o seguinte: quando o semáforo contador de empty tiver valor = 0, o(s) produtores ficarão em espera. Contudo, como essa chamada de wait(empty) agora faz parte da região crítica - entre o wait(mutex) e o signal(mutex) - quando um consumidor tentar consumir um recurso e aumentar o valor do semáforo empty, este não conseguirá adquirir o semáforo mutex. Assim, ocorrerá um deadlock - 1 produtor ficará esperando por um valor maior do que 0 em empty enquanto os consumidores ficarão esperando por mutex. Os outros produtores ficarão também esperando por mutex para acessar a região crítica.

10

A chamada Signal associada a semáforos incrementa um contador, abrindo um semáforo, enquanto em monitores a chamada Signal envia um sinal para uma thread aguardando-o em wait - se nenhuma thread estiver em wait, este Signal não acarreta em nada.

11

Na semântica Hoare, o comportamento do monitor implementado será o esperado, ou seja, quando um signal(positive) for chamado, haverá a troca de contexto imediata, iniciando a execução da thread em wait. A Condition, nos Hoare Monitors, é garantida de ser verdadeira nesta thread que passou a executar. Assim, na função double raiz(), quando o wait(positive) retornar, é garantido que o valor da double a será positivo. Isto se dá pois, na função inc(), quando há a chamada de signal(positive), o contexto é trocado imediatamente para a thread que estiver no wait(positive). Já na semântica Mesa, a implementação apresentada apresentará problemas. Nesta semântica, a chamada signal() coloca a thread em wait() em estado de ready, porém, não muda imediatamente de contexto como a Hoare, continuando a execução da thread atual. Assim, no caso do Monitor apresentado, quando a uma thread estiver em wait(positive), assim que ela sair deste estado de wait, não há certeza de que o número não é negativo. Um exemplo que pode acontecer é: temos uma thread em wait(positive) e outra que, começando com valor de a = 0, faz inc() - gerando o signal positive - porém continuando sua execução e executando dec() duas vez na sequência, antes da thread escalonar para a thread que estava no wait, em estado de ready. Assim, a thread que estava em wait tentará retornar sqrt(-1), um valor que não é positivo, exemplificando bem como, na semântica Mesa, a Condition não necessariamente é verdadeira quando a nova thread entrar em execução.