Gabriela Lemos Lúcidi Pinhão

Sistemas Distribuídos – Lista 2

1) As duas abordagem são troca de mensagens e memória compartilhada. A primeira é feita pelo sistema operacional através de *system calls* e o espaço de memória do kernel é utilizado para a comunicação entre dois processos. A segunda trata-se de uma região de memória fora do kernel, porém definida pelo sistema operacional, que pode ser usada por um ou mais processos.
   As vantagens da troca de mensagens é que o sistema operacional gerencia a comunicação mas, em compensação, a memória e os recursos do kernel são utilizados para isso. Já a memória compartilhada faz apenas uma *system call* para estabelecer a comunicação entre processos, utilizando menos recursos do sistema operacional, mas a coordenação e sincronização de mensagens é feita pelos próprios processos, o que pode ser uma grande fonte de problemas.

2) Read bloqueante e write não bloqueante – Neste caso a leitura interrompe a escrita a qualquer momento e quando a leitura começa, não existe escrita. O conteúdo da pipe não vai mudar durante a leitura, mas pode ocorrer a leitura de conteúdo incompleto ou vazio.
   Read não bloqueante e write bloqueante - Neste caso a escrita interrompe a leitura a qualquer momento e quando a escrita começa, não há leitura. Isso evita a leitura de conteúdo incompleto ou vazio, mas pode ocorrer que durante o processamento novos conteúdos sejam escritos, o que pode gerar conflito de informações.
   Read e write bloqueantes – Desta maneira não há falhas. A escrita só começa quando a leitura/processamento terminar e vice-versa.
   Read e write não bloqueantes – Neste caso todos os problemas citados acima podem ocorrer.

3) Um sistema multi-threaded é mais leve do que um sistema multi-processed pelo fato de permitir a programação paralela tendo apenas uma CPU. Além disso, quando existe mais de uma CPU, tem melhor aproveitamento delas. Uma possível desvantagem é o aumento da complexidade do programa que gerencia os processos.

4) Pode ser que sim, mas não pelo fato do sistema ser multi-threaded, mas por conseguir executar processos (com threads ou não) em paralelo, um em cada CPU. O uso de user-level threads não faz diferença para a CPU, ela não enxerga as threads de um processo. O usuário é quem é responsável por implementá-las e a gerência é feita por uma biblioteca acoplada ao programa, não sendo possível fazer duas threads de um mesmo processo serem executadas em cores diferentes. Pode acontecer também de um sistema com mais de uma CPU não lidar bem com o uso de user-level threads e ter desempenho ainda pior do que um sistema com uma só CPU.

5) É a situação em que o resultado depende da sequência de eventos. É muito comum em softwares e hardwares quando há paralelismo, pois a ordem em que as linhas de código executam podem influenciar o resultado final. É

importante ressaltar que a condição de corrida só é indesejável quando ela causa um resultado inesperado, como por exemplo:

```
retirada (conta, valor) {
        saldo = getSaldo(conta);
        saldo = saldo – valor;
        putSaldo(conta, saldo);
        retorna saldo;
}
```

Supondo que Maria esteja fazendo um saque de 100 reais de sua conta num caixa automático e sua mãe esteja em outro também fazendo um saque da mesma conta. Se essa função for executada em uma thread disparada por Maria e logo após a linha:

```
saldo = saldo – valor;
```

houver uma troca de contexto para outra thread executando a mesma função disparada pela mãe, quando a primeira thread voltar a executar ao fim da segunda o saldo não estará atualizado e retornará 900 reais, independente de quanto dinheiro a mãe sacar.

6) Se na linha:

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

houver uma troca de contexto, ambas as threads ficarão presas nos loops e acontecerá um deadlock.

7)
```
struct lock {
   bool turn[2] = [0,1];
}

void acquire(lock) {
   while(lock->turn[other_thread]);
}

void release(lock) {
   swap(lock->turn[this_thread], lock->turn[other_thread]);
}
```

8) Caso os semáforos s1 e s2 estejam abertos, será imprimido o padrão “ababab...” para cada thread da fila.
Se o semáforo s2 estiver inicialmente fechado, a primeira thread da fila conseguirá imprimir “a”, mas jamais conseguirá executar o resto do código, pois ficará esperando o semáforo s2 ser liberado e ele não será. A partir daí

nenhuma outra thread conseguirá imprimir “a”, pois o semáforo s1 estará fechado, ocorrendo deadlock.
Se o semáforo s1 estiver fechado, independente do s2 estar ou não, nada será impresso pois a primeira thread da fila, assim como todas as outras, ficarão esperando o semáforo s1 estar aberto, ocorrendo deadlock.

9) Poderia acontecer um deadlock caso não houvesse nenhum buffer vazio. O semáforo mutex estaria fechado, impossibilitando que o consumidor liberasse um buffer.

10) Ambas permitem a entrada de uma thread em espera na região crítica, a diferença é que no monitor, se não existem threads em espera, o sinal é perdido. Já os semáforos têm memória e sempre incrementam um contador.

11) Para a semântica Hoare, se uma thread chamar a função raiz(), ela será bloqueada e colocada em estado waiting se o número for menor que zero. Porém, quando as próximas threads chamarem a função inc() o número suficiente de vezes para que o número fique positivo, no momento que isso acontecer haverá uma troca de contexto para a função anteriormente bloqueada, que poderá terminar de executar a função raiz(). Dessa forma, garante-se que o número continuará positivo ao executar-se a função raiz().

    Na semântica Mesa, se uma thread chamar a função raiz(), ela será bloqueada e colocada em estado waiting se o número for menor que zero. Porém, quando as próximas threads chamarem a função inc() o número suficiente de vezes para que o número fique positivo, no momento que isso acontecer, a thread anteriormente bloqueada será colocada em estado ready, mas não entrará em execução até que a atual thread em execução termine de executar suas funções. Com isso, pode acontecer que quando a thread anteriormente bloqueada continue a execução da função raiz(), o número não seja mais positivo.