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 as mensagens mas, em compensação, a memória e os recursos do kernel são utilizada 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 de mensagens é feita pelos próprios processos, o que pode ser uma grande fonte de problemas.

1. 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.

1. Um sistema multi-threaded é bem mais leve do que um sistema multi-processed pelo fato de que, no primeiro, um único processo gasta os mesmos recursos que no segundo, porém podendo ter várias linhas de execução. Além disso, sistema multi- threaded permite a programação paralela já que sua abstração permite duas threads executando ao mesmo tempo sem necessariamente ter mais de uma CPU. Uma possível desvantagem é o aumento da complexidade do programa que gerencia os processos.
2. 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.
3. 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 está fazendo um saque de 100 reais de sua conta num caixa automático e sua mãe está 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.

1. Se na linha:

lock->interested[this\_thread] = 1

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

1. VER