Olá chat, tenho uma proposta para esse chat atual. Tenho que estudar para uma prova de Sistemas Operacionais da minha faculdade. Essa prova se baseia em 39 afirmações que podem estar certas ou erradas. Meu papel é descobrir quais estão erradas e justificar o porquê de estarem erradas. Irei te mandar as 39 afirmações e te direi quais estão erradas com a sentença (falsa) antes da afirmação. Após isso, quero que aleatoriamente me mande uma das afirmações que eu te enviei na lista abaixo e eu terei que dizer se essa afirmação está certa ou errada, caso estiver errada eu terei que justificar. No final você irá corrigir a minha resposta, se eu disser que é correta, mas é falsa você irá me explicar o porquê e se eu disser que é falsa, mas é correta você também me irá explicar o porquê, além de também explicar o porquê as afirmações que eu acertei estão corretas ou erradas. Em geral, você irá explicar a minha resposta em qualquer ocasião. Após as 39 afirmações você me passará um relatório de quantas acertei e errei.

Aqui estão as 39 afirmações:

1. Um processo pode estar em 3 estados diferentes: rodando, bloqueado e pronto.

2. (falsa) Na multi-programação cada programa tem o monopólio da CPU até seu término, mas o Sistema Operacional pode decidir a ordem do escalonamento de alto nível do processo (decidir qual a ordem em que eles serão executados).

3. (falsa) No Minix existem apenas 3 maneiras dos processos se comunicarem, todas síncronas: arquivos, pipes e mensagens.

4. Os processos em Minix são organizados de maneira hierárquica, com o processo INIT no topo da árvore.

5. Pipes são tratados como arquivos no MINIX.

6. (falsa) Na versão que utilizamos do Minix o SO é dividido em 4 camadas, sendo que as duas inferiores compartilham o espaço de endereçamento e nas outras os processos rodam com memorias independentes.

7. Em Minix, chamadas de sistemas não são realmente “chamadas”. A biblioteca do sistema transforma a chamada de procedimentos no envio de mensagens sı́ncronas.

8. Todos os processos em Minix, inclusive os do Kernel (System Task e Clock Task) funcionam de maneira sı́ncrona, com um loop de recebimento de mensagens.

9. (falsa) O funcionamento do Minix com envio de mensagens torna o sistema menos seguro.

10. A única maneira de se criar um processo em Minix é pela chamada de sistema `fork()`.

11. (falsa) As duas maneiras de se criar um processo em Minix são a chamada de sistema `fork()` e a chamada de sistema `execve()`.

12. (falsa) A rotina `malloc()`, deve utilizar diretamente a chamada de sistema `brk()`. Desta maneira, a rotina `free()` libera memória apenas quando a regiã o liberada está no limite da área de dados do processo.

13. (falsa) A chamada `open()` é necessária apenas para verificar as permissões do arquivo.

14. O Minix tem apenas dois tipos de arquivo: de bloco e de caractere. O primeiro oferece acesso aleatório pela manipulação do ponteiro de leitura. O segundo provê apenas acesso sequencial.

15. Para CRIARMOS um pipe no Minix precisamos utilizar não somente a chamada `pipe()` mas também a chamada `close()`.

16. Quando queremos redirecionar entrada e saı́da, utilizamos a chamada `dup()`.

17. A chamada `setuid()` é muito útil para a segurança do sistema Minix, uma vez que permite que um processo chamado por um usuário rode com privilégios de superusuário.

18. Tabelas de processo são essenciais para o multiprocessamento. Elas guardam todas as informações de um processo e permitem o restauro de seu estado quando ele voltar a executar.

19. (falsa) No Minix, assim como no UNIX , apenas o Kernel é responsável pela manutenção da tabela de processos. Chamadas ao System Task permitem que os programas obtenham os dados que precisam para tomar decisões de escalonamento.

20. (falsa) Interrupções sã o comunicações assíncronas do hardware para o Kernel do sistema. Quando ocorrem, o hardware consulta uma tabela de endereços. Estes endereços se referem a procedimentos do System Task. Basta então que o processador de um goto a estes procedimentos, escritos diretamente na linguagem C.

21. Em muitos sistemas operacionais, processos que trabalham em conjunto compartilham alguma memória em comum. O uso compartilhado dessa memória cria “condições de corrida”, o que implica que o uso deste recurso precisa ser coordenado. As regiões do programa que acessam a memória comum sã o chamadas de “regiões criticas”.

22. (falsa) São 3 as condições para uma boa soluçã o para o problema da exclusã o mútua:

i. Só um processo deve entrar na região critica de cada vez.

ii. Não deve ser feita nenhuma hipótese sobre a velocidade relativa dos processos.

iii. Nenhum processo executando fora de sua região crı́tica deve bloquear outro processo.

23. Existe uma equivalência entre semáforos, monitores e mensagens. Qualquer um destes esquemas pode ser implementado usando o outro.

24. São 3 os nı́veis de escalonamento de processos:

i. alto nı́vel onde se decide quais processos entram na disputa por recursos;

ii. nı́vel médio, usado para balanceamento de carga;

iii. baixo nı́vel, onde se decide quão dos processos prontos deve ter o controle da CPU.

25. (falsa) Um processo em uma fila de menor prioridade sempre demorará mais para rodar que um processo numa fila de maior prioridade.

26. O código abaixo resolve o problema dos filósofos comilões:

#define N 5

Philosopher(i) {

int I;

think();

take\_chopstick(i);

take\_chopstick((i+1) % N);

eat();

put\_chopstick(i);

put\_chopstick(i+1);

}

27. Em sistemas não preemptivos, o relógio é desligado e o processo decide quando deve ceder a CPU. Isso pode gerar o travamento do sistema.

28. No sistema de multi-level feeback queues, existem várias filas de prioridade e cada processo é alocado a uma desde o inı́cio de sua execução. Desta maneira a alocação desta prioridade é essencial para o bom desempenho do sistema.

29. (falsa) No EP1, para que os comandos rodados com `rode()` ou `rodeveja()` recebessem argumentos, é necessário criar um vetor de strings com os argumentos que seria passado para a chamada de sistema `execve()`.

30. (falsa) Para implementar semá foros usando monitores, basta criar um monitor chamado “semaforo\_mon”, com dois procedimentos, P() e V(). A exclusã o mútua do monitor por si só já garante o funcionamento, sem necessidade de mais código.

31. (falsa) Em sistemas de memória real não é possı́vel executar um programa maior do que a memória, daı́ a importância da memória virtual

32. (falsa) Em sistemas de memória real, a estratégia de colocação “first fit” é baseada no princı́pio da localidade.

33. Em sistemas de memória real as estratégias “best fit” e “worst fit” envolvem heurı́sticas que visam diminuir a fragmentação externa

34. Em sistemas de memória real com partições fixas um programa é compilado para rodar em apenas uma partição, isso pode gerar ociosidade no sistema, mesmo com programas habilitados a rodar.

35. Em sistemas de memória particionada, um processo sem memória disponı́vel fica aguardando fora das filas de escalonamento primá rio (ou seja, seu estado NÃO é “pronto"}.

36. (falsa) No minix o grande desafio de implementação é impedir que muitas mensagens se acumulem e causando um overflow do buffer de mensagens.

37. As mensagens do Minix são sı́ncronas.

38. O Estado abaixo é seguro de acordo com o algoritmo do banqueiro: (chat, nessa afirmação me de um exemplo de Estado de sistema operacional em uma tabela para eu avaliar se é seguro ou não)

39. (falsa) A solução abaixo resolve o problema da exclusã o mútua:

p1dentro = FALSE;

p2dentro = FALSE;

Processo 1

while (TRUE){

while (p2dentro == TRUE){};/\*espera\*/

p1dentro = TRUE;

…..<regiã o crı́tica>….

p1dentro = 0; …..

<resto do código>….

}

Processo 2

while (TRUE){

while (p1dentro == TRUE){};/\*espera\*/

p2dentro = TRUE;

…..<regiã o crı́tica>….

p2dentro = 0;

…..<resto do código>….

}