**MINIX**

9. (2018/2022) O funcionamento do Minix com envio de mensagens torna o sistema menos seguro. **Falso**, torna mais seguro. Se há a vulnerabilidade em um dos processos, só aquela área é afetada. Um kernel monolítico tornaria o sistema menos seguro.

6. (2022) 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. Falso. Na versão 3.2.1a, o SO se divide em 3 camadas (user, system, kernel).

36. (2022) No Minix o grande desafio de implementação o é impedir que muitas mensagens se acumulem e causando um overflow do buffer de mensagens. **Falso.** O Minix não lida diretamente com um buffer de mensagens que possa sofrer overflow, pois a entrega de mensagens é confiável e ordenada entre os processos.

37. (2022) As mensagens do Minix são síncronas.

**MULTI-PROGRAMAÇÃO**

2. (2018/2022) Na multiprogramação 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). **Falso.** Não há monopólio da CPU até o término. Isso dependerá do tipo de escalonamento. No geral, um programa roda por um tempo determinado e então é bloqueado e substituído por outro, e assim sucessivamente.

**PROCESSOS**

1. (2018/2022) Um processo pode estar em 4 estados diferentes: rodando, pronto, bloqueado e suspenso**. Falso.** Dependendo do sistema operacional, há 3 ou 5 estados. Não são todos os sistemas que possuem o estado “suspenso”, e os que possuem, têm suspenso bloqueado e suspenso pronto.

**EP**

30. (2018/2022) No EP1 para que os comandos rodados com rode() ou rodeveja() recebessem argumentos, é necessário criar um vetor de strings com os argumentos que seriam passados para a chamada de sistema execve().

**EP**

30. (2018/2022) No EP1 para que os comandos rodados com rode() ou rodeveja() recebessem argumentos, é necessário criar um vetor de strings com os argumentos que seriam passados para a chamada de sistema execve()**.**

**ARQUITETURA DE COMPUTADORES**

**PROCESSOS**

1. (2018/2022) Um processo pode estar em 4 estados diferentes: rodando, pronto, bloqueado e suspenso**. Falso.** Dependendo do sistema operacional, há 3 ou 5 estados. Não são todos os sistemas que possuem o estado “suspenso”, e os que possuem, têm suspenso bloqueado e suspenso pronto.

**MULTI-PROGRAMAÇÃO**

2. (2018/2022) Na multiprogramação 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). **Falso.** Não há monopólio da CPU até o término. Isso dependerá do tipo de escalonamento. No geral, um programa roda por um tempo determinado e então é bloqueado e substituído por outro, e assim sucessivamente.

**MINIX**

9. (2018/2022) O funcionamento do Minix com envio de mensagens torna o sistema menos seguro. **Falso**, torna mais seguro. Se há a vulnerabilidade em um dos processos, só aquela área é afetada. Um kernel monolítico tornaria o sistema menos seguro.

36. (2022) No Minix o grande desafio de implementação o é impedir que muitas mensagens se acumulem e causando um overflow do buffer de mensagens. **Falso.** O Minix não lida diretamente com um buffer de mensagens que possa sofrer overflow, pois a entrega de mensagens é confiável e ordenada entre os processos.

37. (2022) As mensagens do Minix são síncronas.

**PROCESSOS MINIX**

3. (2018/2022) No Minix existem apenas 3 maneiras dos processos se comunicarem, todas síncronas: arquivos, pipes e mensagens. **Falso.** Há quatro tipos, o quarto sendo os sinais, que são assíncronos.

4. (2018/2022) Os processos em Minix são organizados de maneira hierárquica, com o processo INIT no topo da árvore. Essa é polêmica. Marcar como **falsa** se estiver faltando sentença pra marcar. Ela é verdadeira para processos de usuário, mas, por exemplo, daemons e drivers não são filhos do INIT.

8. (2018) Todos os processos em Minix, inclusive os do Kernel (System Task e Clock Task) funcionam de maneira síncrona, com um loop de recebimento de mensagens.

**ARQUIVOS MINIX**

5. (2018/2022) Pipes são tratados como arquivos no MINIX.

14. (2018/2022 (aparece como verdadeira)) 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. **Falso.** Também existem arquivos especiais e arquivos comuns (ASCII ou binários, por exemplo), além de diretórios, que são arquivos do sistema.

**ESTRUTURA DO MINIX**

6. (2018/2022) 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 memórias independentes. **Falso.** Só a primeira compartilha espaço de endereçamento. (primeira é a de baixo).

**CHAMADAS DO SISTEMA DO MINIX**

7. (2018/2022) Em Minix, chamadas de sistemas não são realmente “chamadas”. A biblioteca do sistema transforma a chamada de procedimentos no envio de mensagens síncronas.

10. (2018/2022) A única maneira de se criar um processo em Minix é pela chamada de sistema fork().

11. (2018/2022) As duas maneiras de se criar um processo em Minix são as chamadas de sistema fork() e execve() **Falso**. Execve() não cria processo, apenas executa.

12. (2018/2022 (aparece como errada)) A chamada malloc deve utilizar diretamente a chamada de sistema brk(). Desta maneira, a rotina free() libera memória apenas quando a região liberada está no limite da área de dados do processo. Falsa. A malloc() não utiliza exclusivamente a chamada brk() para alocar memória, e a free() não libera memória apenas quando atinge o limite da área de dados.

13. (2018) A chamada open() é necessária apenas para verificar as permissões do arquivo. **Falso**. A chamada open() é necessária para criar o file descriptor, necessário para outras operações de arquivo.

15. (2018/2022) Para criarmos um pipe precisamos usar não somente a chamada pipe mas a chamada close.

16. (2018/2022) Quando queremos redirecionar entrada e saída, usamos a chamada dup().

17. (2018) O comando chroot() é muito útil para usuários uma vez que permite mudar o diretório corrente de um processo, simplificando referências a arquivos no mesmo diretório. **Falso.** Quem faz isso é chdir().

18. (2018/2022) 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 super usuário.

**TABELA DE PROCESSOS**

19. (2018/2022) 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.

20. (2018) No Minix, assim como no UNIX , 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. Essa também é polêmica, marcar só se estiver **faltando**. A tabela de processos pode ser distribuída.

19. (2022) 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. **Falso.** a manutenção da tabela de processos não é exclusivamente responsabilidade do Kernel, a tabela de processos pode ser distribuída entre o Kernel e o System Task.

**EXCLUSÃO MÚTUA**

22. (2018/2022) 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 são chamadas de “regiões críticas”.

23. (2018/2022) São 3 as condições para uma boa solução para o problema da exclusão mútua. **Falso**, são 4:

> Só um processo deve entrar na região crítica de cada vez

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

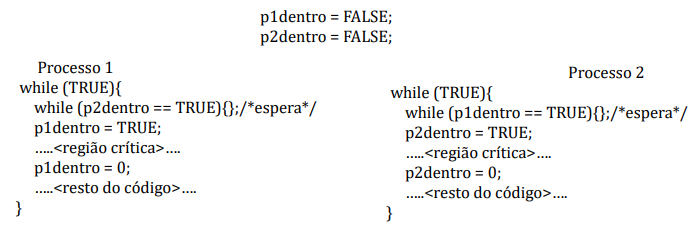
> Nenhum processo executando fora de sua região crítica deve bloquear outro processo

> Nenhum processo deve esperar um tempo arbitrariamente longo para entrar na sua região crítica (adiamento indefinido)

25. (2018/2022) Existe uma equivalência entre semáforos, monitores e mensagens. Qualquer um destes esquemas pode ser implementado usando o outro.

30. (2022) Para implementar semá foros usando monitores, basta criar um monitor chamado “semaforo\_mon”, com dois procedimentos, P() e V(). A exclusão mú tua do monitor por si só́ já́ garante o funcionamento, sem necessidade de mais código. **Falso.** Criar um monitor com exclusão mútua não é suficiente para implementar semáforos.

39. (2022) A solução abaixo resolve o problema da exclusão mútua:



**COMUNICAÇÃO ENTRE PROCESSOS**

21. (2018) Interrupções são 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 que são então ativados.

20. (2022) Interrupções são 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.** **Falso.** Essa rotina de tratamento de interrupção é escrita em linguagem de baixo nível, geralmente em linguagem Assembly.

**ESCALONAMENTO DE PROCESSOS**

26. (2018/2022) São 3 os níveis de escalonamento de processos: i. alto nível, onde se decide quais processos entram na disputa por recursos; ii. nível médio, usado para balanceamento de carga; iii. baixo nível, onde se decide quais dos processos prontos deve ter o controle da CPU.

27. (2018 (marcada como vdd)) Em sistemas **preemptivos**, o relógio é desligado e o processo decide quando deve ceder a CPU. Isso pode gerar o travamento do sistema.

27. (2022(marcada como vdd)) 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. (2018/2022 (marcada como vdd)) No sistema de multi-level feedback queues, existem várias filas de prioridade e cada processo é alocado a uma desde o início de sua execução. Desta maneira a alocação desta prioridade é essencial para o bom desempenho do sistema. Uma maneira de fazer isso são as “prioridades compradas”, onde se delega ao usuário decidir em que fila seu processo rodará. Essa é polêmica, e o Renato disse inicialmente que estava certa. Só marcar se estiver realmente **faltando** alternativas, deixar como última. Pode ter alocação dinâmica, então a alocação das prioridades não é exatamente essencial para o bom funcionamento do sistema.

29. (2018/2022) Um processo em uma fila de menor prioridade sempre demorará mais para rodar que um processo numa fila de maior prioridade. **Falso.** Renato disse explicitamente que é pegadinha. A prioridade máxima é zero, de forma que quanto menor o valor da prioridade, mas próximo de 0 e, portanto, menos demorará para rodar. Isso sem contar com a prioridade dinâmica nas multilevel feedback queues.

**DEADLOCKS**

24. (2018/2022) 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);

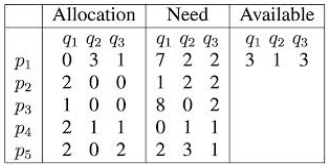
eat();

put\_chopstick(i);

put\_chopstick(i+1); }

Falso. O algoritmo permite que todos os filósofos peguem o pauzinho esquerdo, ou todos peguem o direito ou mesmo que ninguém faça mais nada e caímos em deadlock.

38. (2022) O Estado abaixo é seguro de acordo com o algoritmo do banqueiro:



**ENTRADA E SAÍDA**

**ERROS**

**MINIX FLOPPY**

**DRIVERS: RELÓGIO**

**DRIVERS DE TERMINAL**

**SYSTEM TASK DO MINIX**

**ADMINISTRAÇÃO DE MEMÓRIA**

35. (2022) Em sistemas de memó ria particionada, um processo sem memó ria disponível fica aguardando fora das filas de escalonamento primá rio (ou seja, seu estado NÃO é “pronto"}.

**MEMÓRIA REAL**

31. (2022 (marcada como falsa)) Em sistemas de memória real não é possível executar um programa maior do que a memória, daí́ a importância da memória virtual.

32. (2022) Em sistemas de memória real, a estratégia de colocação o “Eirst Eit” é baseada no princípio da localidade. **Falso**. não tem relação direta com o princípio da localidade.

33. (2022) Em sistemas de memó ria real as estratégias “best Eit” e “worst Eit” envolvem heurísticas que visam diminuir a fragmentação o externa.

34. (2022) 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.

**MEMÓRIA VIRTUAL**