1. 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.

2. 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.

3. 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. 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.

5. Pipes são tratados como arquivos no MINIX.

6. 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).

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

8. 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.

9. O funcionamento do Minix com envio de mensagens tornao 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.

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

11. 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. 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.

13. 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.

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.

Falso. Também existem arquivos especiais e arquivos comuns (ASCII ou binários, por exemplo), alé de diretórios, que são arquivos do sistema.

15. Para criarmos um pipe precisamos usar não somente a chamada pipe mas a chamada close.

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

17. 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. 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.

19. 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. 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.

21. 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.

22. 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 criticas".

23. 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)

24. 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.

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

26. 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. Em sistemas 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 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. 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.

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

DISSERTATIVAS

1. **O Minix possui processos de usuário que fazem várias funções tradicionalmente atribuídas ao kernel de um SO. Desenhe as camadas de processos do Minix com seus principais representantes e explique como se dá a comunicação entre elas. Quais as vantagens do microkernel do Minix sobre o kernel monolítico de outros SOs?**

Vantagens do microkernel:

- vulnerabilidade não afeta todo o kernel.
- Se um dos processos que seria parte do kernel morrer, ele pode ser reativado sem que o sistema operacional crashe.
- Modularização facilita que você ative partes novas do SO sob demanda.

Comunicação entre as diferentes partes é feita com mensagens síncronas de cima pra baixo e assíncronas de baixo pra cima (sinais). Lembrando que camada 4 só fala com 3, e camadas 3 e 2 falam com 1.

Desenho:

2. A solução abaixo para o problema dos filósofos está errada. Porque?

```
#define N 5
Philosopher(i){
      int I;
      think();
      take_chopstick(i);
      while(!take_chopstick((i+1) % N){
            put_chopstic(i);
            wait(random);
            take_chopstick(i);
      }
      eat();
      put_chopstick(i);
      put_chopstick(i+1);
}
```

O problema do código é que ele pode sofrer starvation = espera indefinida. É capaz de esperarem pra pegar de forma sincronizada e ficarem todos pegando e largando sem conseguir pegar o da direita.

[OBS: Deadlock é quando o recurso de fato nao está disponível, é uma situação estática. Starvation é uma situação dinâmica, o recurso está disponível mas você não tem acesso a ele (por uma questão de sincronização ou velocidade, por exemplo).]

3. Mostrar se o estado abaixo é seguro de acordo com o algoritmo do banqueiro.

[OBS: seguro significa que tem solução. O abaixo não é o mesmo da prova mas fiquei com preguiça de escrever outro, a lógica é a mesma]

	Allocation			Need			Avaiable		
	q1	q2	q3	q1	q2	q3	q1	q2	q3
p1	0	3	1	7	2	2	3	1	3
p2	2	0	0	1	2	2			
p3	1	0	0	8	0	2			
p4	2	1	1	0	1	1			
p5	2	0	2	2	3	1			

```
 3 1 3 →  P4 pode terminar: libera 2 1 1
 5 2 4 →  P2 pode terminar: libera 2 0 0
 7 2 4 →  P1 pode terminar: libera 0 3 1
 7 5 5 →  P5 pode terminar: libera 2 0 2
 9 5 7 →  P3 pode terminar: libera 1 0 0
10 5 7 →  Sem mais processos: ESTADO SEGURO
```