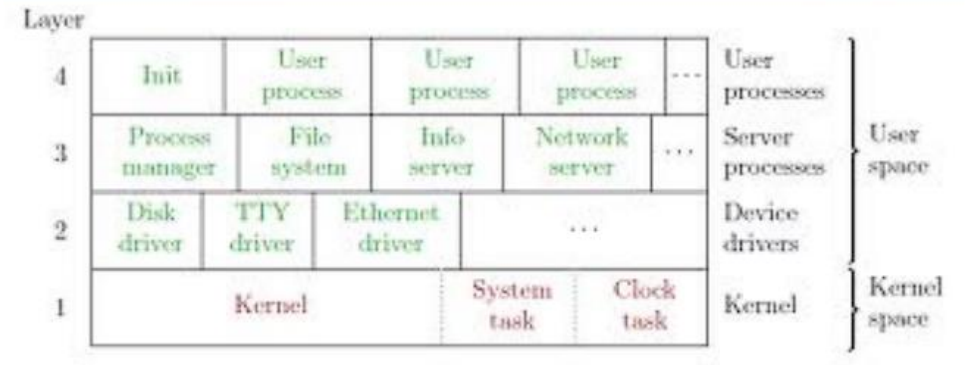
VERDADEIRAS:

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

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



3. Pipes são tratados como arquivos no MINIX.

4. Em Minix, chamadas de sistema não são realmente “chamadas”. A biblioteca do sistema transforma a chamada de procedimentos no envio de mensagens síncronas. (Essa técnica de passagem de mensagens é usada para garantir que as chamadas de sistema em Minix sejam seguras e confiáveis. Em vez de permitir que os processos de usuário acessem diretamente o núcleo do sistema operacional, o sistema operacional gerencia o acesso aos recursos do sistema e fornece uma interface segura e padronizada para os processos de usuário.)

5. Todos os processos em Minix, inclusive os do Kernel (System Task e Clock Task) funcionam de maneira síncrona, com um loop de recebimento d mensagens. (Os processos síncronos em Minix usam um loop de recebimento de mensagens para aguardar a chegada de mensagens de outros processos antes de prosseguir com a execução. O processo de destino envia uma mensagem para o processo de origem, e o processo de origem aguarda a chegada da mensagem antes de continuar a executar.)

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

7. 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.

8. Para criarmos um pipe no Minix precisamos utilizar não somente a chamada pip() mas também a chamada close()

9. Quando queremos redirecionar entrada e saída, utilizamos chamada dup().

10. 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. (a chamada setuid() é utilizada para diminuir os privilégios de um processo, permitindo que ele execute apenas as operações permitidas para o usuário atualmente associado a esse processo.)

11. 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.

12. 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**.

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

14. São 3 os níveis de escalonamento de processos: 1. alto nível onde se decide quais processos entram na disputa por recursos; 2. nível médio usado para balanceamento de carga; 3. baixo nível, onde se decide qual dos processos prontos deve ter o controle da CPU.

15. Em sistemas não preemptivos, o relógio é desligado e o processo decide quando deve ceder a CPU. Isso pode gerar o travamento do sistema. (Se um processo em execução em um sistema não preemptivo entrar em um loop infinito ou ficar preso em uma operação de entrada/saída bloqueante, por exemplo, ele não irá ceder a CPU voluntariamente, deixando outros processos em espera por um longo período de tempo e tornando o sistema inutilizável. Isso ocorre porque o processo em execução tem o controle da CPU e não é interrompido pelo sistema operacional para permitir que outros processos sejam executados.)

16. Em sistemas de memória real as estratégias best fit e worst fit envolvem heurísticas que visam diminuir a fragmentação externa. (Na estratégia Best Fit, o sistema busca pelo bloco livre mais próximo do tamanho requerido pelo processo, ou seja, o bloco que será mais bem "ajustado" para o processo que precisa ser alocado. Isso ajuda a diminuir a fragmentação externa, pois ao escolher o bloco que melhor se encaixa no tamanho do processo, a sobra de espaço livre é menor. Já na estratégia Worst Fit, o sistema aloca o maior bloco livre disponível na memória para o processo, mesmo que esse bloco seja maior do que o necessário. Dessa forma, sobra um espaço livre maior, o que pode parecer contra-intuitivo em um primeiro momento. No entanto, em longo prazo, essa estratégia ajuda a diminuir a fragmentação externa, pois a tendência é que os processos ocupem mais espaço do que precisam, e com o tempo, as sobras de espaço livre serão maiores e mais propícias a receber novos processos.)

17. 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.

18. Em sistemas de memória particionada, um processo sem memória disponível fica aguardando fora das filas de escalonamento primário (ou seja, seu estado NÃO é pronto).

19. As mensagens do Minix são síncronas.

20. O Estado abaixo é seguro de acordo com o algoritmo do banqueiro:

	Allocation			Need			Available		
	q_1	q_2	q_3	q_1	q_2	q_3	q_1	q_2	q_3
p_1	0	3	1	7	2	2	3	1	3
p_2	2	0	0	1	2	2			
p_3	1	0	0	8	0	2			
p_4	2	1	1	0	1	1			
p_5	2	0	2	2	3	1			

P4: 5 2 4

P2: 7 2 4

P1: 7 5 5

P5: 9 5 7

P3: 10 5 7

21. O comando chroot() é muito útil para usuários uma vez que permite mudar o diretório corrente de um processo, simplificando refecias a arquivos no mesmo diretório. (O comando chroot() permite que um processo mude o diretório raiz do sistema de arquivos para um diretório diferente do padrão. Isso cria um ambiente isolado para o processo, com seu próprio diretório raiz, que não tem acesso aos arquivos e recursos do sistema de arquivos fora desse diretório. Essa funcionalidade é útil em várias situações, como em servidores web que hospedam vários sites diferentes em um único servidor. Ao utilizar o chroot(), cada site pode ser isolado em seu próprio diretório raiz, garantindo que um site comprometido não possa acessar ou afetar os arquivos de outros sites no mesmo servidor.)

FALSAS:

1. 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). (vários processos são executados simultaneamente, compartilhando recursos de hardware, como a CPU. o sistema operacional é executado como um processo em segundo plano, e cada processo em execução tem acesso à CPU em períodos intercalados, em um processo chamado de compartilhamento de tempo. Assim, cada processo tem sua parcela justa de tempo de execução, e o sistema operacional coordena o acesso à CPU para garantir que nenhum processo monopolize a CPU indefinidamente.)

2. No Minix existem apenas 3 maneiras dos processos se comunicarem, todas síncronas: arquivos, pipes e mensagens. (O sistema operacional Minix oferece mais do que apenas três maneiras síncronas de comunicação entre processos. Sinais: permitem que um processo envie uma interrupção para outro processo ou para si mesmo. Os sinais podem ser usados para interromper ou notificar processos sobre eventos externos, como a chegada de um sinal de alarme ou a ocorrência de um erro. Semáforos: permitem que um processo bloqueie ou libere o acesso a um recurso compartilhado por outros processos. Os semáforos podem ser usados para garantir a exclusão mútua ou para sincronizar o acesso a recursos compartilhados. Registros compartilhados: permitem que processos compartilhem informações por meio de uma região de memória compartilhada. Os registros compartilhados podem ser usados para transferir grandes quantidades de dados entre processos de maneira eficiente. Sockets: permitem que processos em diferentes máquinas se comuniquem por meio da rede. Os sockets podem ser usados para implementar aplicativos cliente-servidor ou para transferir dados entre processos em diferentes máquinas. RPC (Remote Procedure Call): permite que um processo chame uma função em outro processo como se estivesse chamando uma função local. O RPC pode ser usado para simplificar a programação de aplicativos cliente-servidor e para separar as responsabilidades de diferentes processos em um sistema distribuído.)

3. 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. (Na versão 3.2.1a, o SO se divide em 3 camadas).

4. O funcionamento do Minix com envio de mensagens torna o sistema menos seguro. (Na verdade, o sistema de envio de mensagens do Minix aumenta a segurança do sistema, ao invés de torná-lo menos seguro. o sistema de mensagens do Minix é projetado para garantir a integridade e a autenticidade das mensagens, por meio do uso de assinaturas digitais. Isso garante que as mensagens enviadas sejam originárias do processo correto e não foram modificadas durante a transmissão.)

5. As duas maneiras de se criar um processo em Minix são a chamada de sistema fork() e a execve(). (Na verdade, a chamada de sistema fork() é uma forma de criar um novo processo no Minix, enquanto a chamada de sistema execve() é uma forma de carregar um novo programa em um processo existente.)

6. A rotina 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. (A rotina malloc() não precisa necessariamente utilizar diretamente a chamada de sistema brk() para alocar memória. Na verdade, existem outras formas de alocar memória, como por exemplo através da chamada de sistema mmap(), que pode ser utilizada para mapear uma região de memória no espaço de endereçamento do processo. Quanto à rotina free(), ela é responsável por liberar a memória alocada anteriormente com a rotina malloc() ou outras funções similares. Quando a memória é liberada com a chamada da função free(), ela é devolvida para o sistema operacional e pode ser reutilizada por outras alocações de memória futuras.)

7. A chamada open() é necessária apenas para verificar as permissões do arquivo. (A chamada open() não é usada apenas para verificar as permissões de um arquivo, ela é utilizada para abrir um arquivo e obter um descritor de arquivo, que pode ser utilizado para ler e escrever dados no arquivo.)

8. 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. (Embora o kernel seja responsável pela manutenção da tabela de processos, os programas do usuário podem obter acesso a essa tabela por meio do diretório /proc. Não é necessário fazer chamadas ao System Task para obter informações sobre os processos.)

9. 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. (As interrupções geralmente são tratadas por rotinas de tratamento de interrupção (interrupt handlers) escritas em linguagem de baixo nível (geralmente em assembly) e armazenadas em uma tabela de vetores de interrupção (interrupt vector table). Quando uma interrupção ocorre, o processador consulta a tabela de vetores de interrupção para determinar qual rotina de tratamento de interrupção deve ser executada e, em seguida, transfere o controle para essa rotina. O System Task é uma tarefa específica do Minix que é responsável por gerenciar o sistema, incluindo o tratamento de interrupções. Quando uma interrupção ocorre, o System Task é notificado e pode executar o tratamento adequado. No entanto, os tratamentos de interrupção não são escritos diretamente em C, mas sim em linguagem de baixo nível.)

10. São 3 as condições para uma boa solução para o problema da exclusão mútua: 1. Só um processo deve entrar na região crítica de cada vez. 2. Não deve ser feita nenhuma hipótese sobre a velocidade relativa dos processos. 3. Nenhum processo executando fora de sua região crítica deve bloquear outro processo. (4 condições para uma boa solução; 1. Só um processo deve entrar na região crítica de cada vez; 2. não deve ser feita nenhuma hipótese sobre a velocidade relativa dos processos; 3. nenhum processo executando fora de sua região crítica deve bloquear outro processo; 4. nenhum processo deve esperar um tempo arbitrariamente longo para entrar na sua região crítica (adiamento indefinido)

11. Um processo em uma fila de menor prioridade sempre demorará mais para rodar que um processo numa fila de maior prioridade. (É possível que um processo em uma fila de menor prioridade seja escalonado para a CPU antes de um processo em uma fila de maior prioridade, se o processo de menor prioridade tiver usado menos tempo de CPU do que o processo de maior prioridade.)

12. O código abaixo resolve o provlema 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);
}
```

(Primeiro todos pegam o da esquerda, depois todos pegam o da direita e no final ninguém pega nada (Deadlock))

13. Para implementar semáforos usando monitores, basta criar um monitor chamado semaforo_mon, com dois procedimentos, P() e V(). (Para implementar semáforos com monitores, é necessário criar uma classe ou um módulo que represente o semáforo e, dentro dele, implementar os procedimentos P() e V() que controlam o acesso à variável de condição do monitor. Além disso, é necessário garantir que a fila de processos bloqueados esteja sendo manipulada corretamente, ou seja, os processos devem ser colocados na fila correta de acordo com o valor do semáforo e serem desbloqueados na ordem correta quando o valor do semáforo mudar. A exclusão mútua do monitor por si só já garante o funcionamento, sem necessidade de mais código. (Embora a exclusão mútua de um monitor ajude a garantir a corretude da implementação do semáforo, ela não garante sozinha o correto funcionamento de um semáforo, especialmente em casos de múltiplos semáforos ou semáforos com valores iniciais diferentes.)

14. Em sistemas de memória real não é possível executar um programa maior do que a memória, daí a importância da memória virtual. (Embora seja verdade que um programa não pode ser executado se não puder ser carregado inteiramente na memória principal, os sistemas operacionais modernos têm uma técnica chamada swapping, que permite que partes do programa atualmente não utilizadas sejam movidas para a memória secundária (geralmente um disco rígido), liberando espaço na memória principal para outras partes do programa. Além disso, a memória virtual não é utilizada apenas para lidar com programas maiores do que a memória principal, mas também para permitir que diferentes processos tenham seus próprios espaços de endereçamento virtuais, isolados uns dos outros, o que aumenta a segurança e a estabilidade do sistema.)

15. Em sistemas de memória real, a estratégia de colocação first fit é baseada no princípio da localidade. (Na verdade, a estratégia de alocação first fit é baseada no princípio da eficiência e simplicidade, e não no princípio da localidade. O objetivo do first fit é encontrar o primeiro bloco de memória livre que seja grande o suficiente para alocar uma solicitação de memória, sem levar em consideração a localização dos blocos de memória livre.)

16. No minix o grande desafio de implementação é impedir que muitas mensagens se acumulem e causando um overflow do buffer de mensagens. (No Minix as mensagens são enfileiradas em buffers específicos para cada processo, e o sistema operacional gerencia essas filas de maneira eficiente. Quando o buffer de mensagens de um processo está cheio, o Minix simplesmente bloqueia o processo, impedindo que ele envie mais mensagens até que haja espaço disponível na fila. Além disso, o Minix possui mecanismos de controle de fluxo que evitam a sobrecarga do sistema de mensagens, garantindo a eficiência e segurança na comunicação entre processos.)

17. A solução abaixo resolve o problema da exclusão mútua:

```
p1dentro = FALSE;
p2dentro = FALSE;
```

```
Processo 1
while (TRUE){
   while (p2dentro == TRUE){};/*espera*/
   p1dentro = TRUE;
   .....<região crítica>....
   p1dentro = 0;
   .....<resto do código>....
}
```

```
Processo 2
while (TRUE){
   while (p1dentro == TRUE){};/*espera*/
   p2dentro = TRUE;
   .....<região crítica>....
   p2dentro = 0;
   .....<resto do código>....
}
```

(Os dois testam ao mesmo tempo e entram)

18. No sistema 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. (Em primeiro lugar, no sistema multi-level feedback queues, o nível de prioridade de um processo é dinamicamente ajustado de acordo com seu comportamento de execução. Processos que utilizam muito a CPU são movidos para filas de prioridade mais baixas, enquanto aqueles que usam pouca CPU são movidos para filas de prioridade mais altas. Em segundo lugar, delegar ao usuário a decisão de em qual fila seu processo rodará não é uma boa prática, pois o usuário pode não ter conhecimento suficiente sobre o comportamento do sistema e pode tomar decisões que prejudicam o desempenho do sistema. Por fim, a alocação de prioridades não é feita por meio de prioridades compradas, mas sim por meio de algoritmos de escalonamento que levam em consideração vários fatores, como tempo de espera, tempo de execução e prioridade atual do processo.)