MaC0422 –Sistemas Operacionais
Prova 1
29/Setembro/2016
Prof. Alan Durham

Esta prova tem duas partes. **A primeira parte** contém 29 afirmações, você deve selecionar 14 destas que estão ERRADAS. Sua folha de resposta deve conter as questões erradas **NA ORDEM** com uma justificativa do porquê estão erradas. A justificativa deve ter apenas uma sentença (curta, não deve ter mais que 3 linhas). Respostas sem justificativa nãoserão consideradas. Cada resposa vale 1 ponto. Não serão consideradas mais que 14 respostas. **A segunda parte** tem duas questões dissertativas, que devem ser respondidas no espaço alocado. Cada questão vale 3 pontos.

PARTE I

1. Um processo pode estar em 4 estados diferentes: rodando, pronto, bloqueado e suspenso.
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).
3. 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. 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 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 torna 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 a chamada de sistema `` `fork()` `` e a chamada de sistema `` `execve()` ``.
12. 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.
13. 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 saída, utilizamos a chamada `` `dup()` ``.
17. O comando `` `chroot()` `` é muito útil para usuários uma vez que permite mudar o diretório corrente de um processo, simplificando referencias a arquivos no mesmo diretório.
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 superusuário.
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.
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:
    i. Só um processo deve entrar na região crítica de cada vez.
    ii. Não deve ser feita nenhuma hipótese sobre a velocidade relativa dos processos.
    iii. Nenhum processo executando fora de sua região crítica deve bloquear outro processo.

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);
}
```

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 quão 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 feeback 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.

29. Um processo em uma fila de menor prioridade sempre demorará mais para rodar que um processo numa fila de maior prioridade.

30. 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()` ``.

31. No EP2, é equivalente criar uma função `` `execve_batch()` `` no lugar de `` `fork_batch()` `` para criar processos *batch*.

32. No EP2, a execução da chamada de sistemas `` `fork_batch()` `` exige que a tabela de processo seja modificada no *Kernel* (por meio do *System Task*), no *Process Manager* e no *File System*.

33. No EP2, o escalonamento *batch* proposto minimiza o tempo entre a submissão e o término do processo a ser rodado.

34. No EP2, o escalonamento *batch* proposto mantém a CPU ocupada todo o tempo possível.

35. No EP2, o escalonamento *batch* proposto é capaz de priorizar a execução de processos que façam bastante entrada e saída sobre processos que usam muita CPU.

PARTE II

1. (2 pontos) 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?

2. (2 pontos) 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);
}
```

3. (2 pontos) Mostre se o estado abaixo é seguro ou não 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			