MaC0422 –Sistemas Operacionais
Prova 1
01/06/2022
Prof. Alan Durham

Esta prova tem duas partes.
A primeira parte tem DUAS questões dissertativas (VEJA O VERSO), que devem ser respondidas no espaço alocado. Cada questão vale 3 pontos.
A segunda parte contém 39 afirmações, 16 destas estão ERRADAS. Você deve selecionar 10 destas e indicar porque estão erradas na sua folha de resposta (APENAS 10). 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 não serão consideradas. Cada resposta vale 1 ponto. Não serão consideradas mais que 10 respostas.

IMPORTANTE: cuidado com as justificativas, estas não devem ser apenas a negação da afirmação, elas devem mostrar que você sabe PORQUE a afirmação está falsa!
O QUE ENTREGAR: Você deve entregar duas folhas, uma com as respostas da primeira parte e a folha da segunda parte preenchida. COLOQUE SEU NOME NA PROVA.

NOME:_____ Num USP:_____

PARTE I

1. Descreva brevemente os passos que você executou para implementar o EP1. Quais partes do sistema precisaram ser mexidas (se não lembrar do nome de rotinas, ok, apenas explique os módulos alterados e o que foi feito)

2. Descreva brevemente o passos que você executou para implementar o EP2. Quais partes do sistema precisaram ser mexidas?

PARTE 2

1. Um processo pode estar em 3estados diferentes: rodando, bloqueado e pronto.
2. 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. 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. 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. 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. 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.*
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 são chamadas de "regiões criticas".
22. 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.
23. Existe uma equivalência entre semáforos, monitores e mensagens. Qualquer um destes esquemas pode ser implementado usando o outro.
24. 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.
25. 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 início de sua execução. Desta maneira a alocação desta prioridade é essencial para o bom desempenho do sistema.

29. 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. Para implementar semáforos usando monitores, basta criar um monitor chamado "semaforo_mon", com dois procedimentos, P() e V(). A exclusão mútua do monitor por si só já garante o funcionamento, sem necessidade de mais código.

31. Em sistemas de memória real não é possível executar um programa maior do que a memória, daí a importância da memória virtual

32. Em sistemas de memória real, a estratégia de colocação "first fit" é baseada no princípio da localidade.

33. Em sistemas de memória real as estratégias "best fit" e "worst fit" envolvem heurísticas 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 disponível fica aguardando fora das filas de escalonamento primário (ou seja, seu estado NÃO é "pronto"}.

36. 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 síncronas

38. 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			

39. 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>....
}
```