SISTEMAS OPERACIONAIS

Profª. Roberta Lima Gomes (soufes@gmail.com)

1ª. Lista de Exercícios

***Data de Entrega: não há**. O objetivo da lista é ajudar no estudo individual dos alunos. Soluções de questões específicas poderão ser discutidas em sala de aula, conforme interesse dos alunos.*

INTRODUÇÃO

1. Defina o que é um *Sistema Operacional*, descrevendo suas principais funções.
2. O que é *multiprogramação*? Cite duas razões para se ter multiprogramação? Cite exemplos de problemas de segurança que podem ocorrer em sistemas multiprogramados.
3. Descreva o que é multiprocessamento. Descreva as diferenças entre multiprocessamento simétrico e assimétrico.
4. Porque sistemas distribuídos são desejáveis?
5. Explique de que forma a utilização de dois modos de operação – supervisor (kernel) e usuário – auxilia na implementação de mecanismos de proteção.
6. Quais das seguintes instruções deveriam ser privilegiadas?
    a. Set valor do relógio
    b. Ler o relógio
    c. Desligar interrupções
    d. Chavear modo usuário/kernel
    e. Limpar posição X da memória
7. Como se dá o processamento de uma interrupção de HW? Qual a relação entre interrupção e multiprogramação?

PROCESSOS

8. Qual a relação entre programa e processo?
9. O que é o PCB (*Process Control Block* ou Bloco de Controloe)? Qual é o seu conteúdo típico?
10. O que é a “imagem” de um processo?
11. Qual é o propósito das *chamadas de sistema* (SVC)?
12. Explique as funções dos escalonadores de curto, médio e longo prazo.
13. Defina *turnaround time*, *throughput*.
14. Qual a relação entre Tempo de Espera e Tempo de Resposta?
15. O que significa um processo sofrer preempção?

16. A maioria dos escalonadores Round Robin usa um quantum de tamanho fixo. Dê um argumento em favor de um quantum pequeno. Agora pense em um argumento que justifique um quantum grande.

17. Discuta o efeito de cada um dos seguintes métodos de atribuição de quantum q no escalonamento circular (round-robin).
    a. q fixo e idêntico para todos os usuários
    b. q fixo e distinto para cada tipo de processo
    c. q variável e idêntico para todos os processos
    d. q variável e distinto para cada tipo de processo

18. Considere o seguinte algoritmo de alocação por prioridade, preemptivo, baseado em prioridades que mudam dinamicamente. Números de prioridades maiores indicam prioridades mais altas. Quando um processo está esperando para entrar em execução (na fila de prontos), sua prioridade muda segundo uma taxa α; quando está em execução, sua prioridade muda segundo uma taxa β. Todos os processos têm prioridade a mesma prioridade quando são criados. Valores diferentes para os parâmetros α e β podem determinar muitos algoritmos de alocação diferentes.
    a) Qual algoritmo é obtido com $\alpha > \beta > 0$?
    b) Qual algoritmo é obtido com $\beta > \alpha > 0$?

19. Suponha um S.O. com escalonador de filas multiníveis na qual há cinco níveis (prioridade entre os níveis). O quantum do primeiro nível é 0,5 segundo. Cada nível mais baixo tem um quantum de tamanho duas vezes maior que o quantum do nível anterior. Um processo não pode sofrer preempção até o seu quantum terminar. Todos os processsos iniciam na fila mais prioritária. Se um processo é preemptado, ele deve ser colocado na fila imediatamente inferior. O sistema executa processos em lote (cpu bound) e interativos (I/O bound).
    a. Por que esse sistema é deficiente?
    b. Quais mudanças mínimas você proporia para tornar o esquema mais aceitável para o mix de processos que pretende?

20. Qual dos algoritmos de escalonamento discutidos em sala de aula poderia ser modificado para acomodar alguns processos de tempo real (processos que *devem* ter uma resposta dentro de um certo período de tempo) misturados com os outros tipos de processo? Explique como você faria isso.

21. Cinco processos, de A até E, chegam ao computador ao mesmo tempo. Eles têm seus tempos de processamento estimados em 10, 6, 2, 4 e 8 minutos respectivamente. Suas prioridades (atribuídas externamente) são 3, 5, 2, 1 e 4, respectivamente, sendo 5 o representante da prioridade mais alta. Nenhum dos processos faz I/O. Para cada um dos algoritmos de escalonamento abaixo, determine o tempo médio de turnaround dos processos. Ignore o overhead causado pela troca de contexto.
    (a) Round Robin (fila começa em A, indo em ordem até E ; quantum = 4)
    (b) Escalonamento com prioridade
    (c) FIFO (ordem de execução: A, B, C, D, E)
    (d) SJF

22. Considere um sistema operacional cuja máquina de estados inclui os estados *Ready* e *Ready-Suspended* (em disco). Suponha que seja hora do S.O. despachar (escalonar) um processo e que existam nesse momento processos tanto no estado *Ready* como no estado *Ready-Suspended*, e que pelo menos um processo no estado *Ready-Suspended* possui prioridade maior do que qualquer processo no estado *Ready*. Duas políticas extremas seriam: *(a)* sempre despachar um processo no estado *Ready*, de forma a minimizar *swapping*; e *(b)* sempre dar preferência ao processo de mais alta prioridade, mesmo que isso possa significar a ocorrência de *swapping* quando este não é necessário. Sugira uma política intermediária (explique e crie um algoritmo) que tente balancear prioridade e desempenho.

23. Considere um sistema que possui duas filas de escalonamento, com prioridades 0 e 1, sendo que somente pode ser escalonado um processo da fila de prioridade 1 não existindo processos na fila de prioridade 0. Sabendo que o algoritmo utilizado nas duas filas é o Round-Robin, escreva o pseudo-código dos procedimentos *insere(p)*, em que p é o índice da tabela de descritores (i.e. PCBs) de processos e que possui um campo (entre outros campos) que contém a prioridade dos processos e *r= seleciona()*, que retorna o índice da tabela de descritores que descreve o processo selecionado. Cite duas situações em que cada procedimento é chamado.

24. No UNIX, um processo pode encontrar-se no estado *Kernel Running* enquanto o sistema pode apresentar dois contextos de execução, *process context* e *system context*. Explique a diferença entre eles.

25. Em algumas implementações do UNIX, o kernel é não-preemptivo. O que isto significa? Quais as vantagens e desvantagens desta abordagem?

26. Descreva de forma geral como funciona o Escalonamento Tradicional no UNIX.

SVCs

27. Descreva o funcionamento da função fork(). Após o fork, como os processos pai e filho podem se comunicar/sincronizar (considere apenas as chamadas fork(), exec(), exit() e wait())?

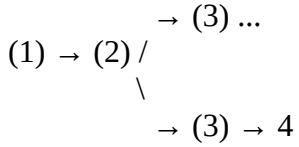
28. Analise o trecho de código a seguir e determine quantos processos são criados. Assuma que nenhum erro ocorre. Desenhe uma árvore que mostre como os processos estão relacionados. Nessa árvore, cada processo será representado por um nó (um círculo contendo um número que representa em qual chamada fork() ele foi criado). O processo original deve conter '0' e o processo criado no 1o. fork() deve conter '1'. Deve haver setas (apontando p/ baixo) saindo de cada pai em direção a cada filho.

```
c2 = 0;
c1 = fork();      /* fork 1 */
if (c1 == 0)
   c2 = fork();   /* fork 2 */
fork();           /* fork 3 */
if (c2 > 0)
   fork();        /* fork 4 */
```

29. Um grafo de precedência é um grafo direcionado em que a relação (a) → (b) indica que 'a' precede 'b'. Neste exercício, os nós do grafo devem representar as instruções numeradas no código abaixo. Desta forma o gráfo de precedência representará a ordem em que as instruções serão executadas. Observe que se o programa não tivesse fork(), o grafo seria linear pois a execução desse programa seria uma sequência de instruções (com desvios ou não). Ex:

(1) → (2) → (8) → (9) → (8) …

No entanto sempre que há um fork(), passamos a ter execuções em paralelo. Ex:



Desenhe o grafo de precedência referente ao código a seguir:

```
    int f1, f2, f3; /* Identifica processos filho*/
    int main(){
[1]   printf("Alo do pai\n");
[2]   f1 = fork;
[3]   if (f1==0)
[4]     execlp(“codigo_filho”,“codigo_filho”,NULL);
[5]   printf("Filho 1 criado\n");
[6]   f2 = fork;
[7]   if (f2==0)
[8]     execlp(“codigo_filho”,“codigo_filho”,NULL);
[9]   printf("Filho 2 criado\n");
[10]  waitpid(f1,null,0);
[11]  printf("Filho 1 morreu\n");
[12]  f3 = fork;
[13]  if (f3==0)
[14]    execlp(“codigo_filho”,“codigo_filho”,NULL);
[15]  printf("Filho 3 criado\n");
[16]  waitpid(f3,null,0);
[17]  printf("Filho 3 morreu\n");
[18]  waitpid( f2,null,0);
[19]  printf("Filho 2 morreu\n");
[20]  exit();
      }

//codigo_filho
        int main()
[21]      printf("Alo do filho\n");
[22]      exit();
        }
```

PARA PENSAR...

... No Linux, em geral há um limite superior “pequeno” para o número (PID) que um processo pode ter (por exemplo 32767). O que você acha que acontece quando esse limite é atingido?

... Em UNIX, mesmo algumas operações extremamente simples como listar os arquivos de um diretório envolvem o disparo de pelo menos um novo processo. No entanto a troca do diretório

corrente (comando "cd") não provoca o disparo de processo algum. Você saberia explicar porquê?