Curso de Tecnologia em Sistemas de Computação
Disciplina de Sistemas Operacionais
Professores: Valmir C. Barbosa e Felipe M. G. França
Assistente: Alexandre H. L. Porto

Quarto Período
AD1 - Segundo Semestre de 2014

Atenção: Cada aluno é responsável por redigir suas próprias respostas. Provas iguais umas às outras terão suas notas diminuídas. As diminuições nas notas ocorrerão em proporção à similaridade entre as respostas. Exemplo: Três alunos que respondam identicamente a uma mesma questão terão, cada um, 1/3 dos pontos daquela questão.

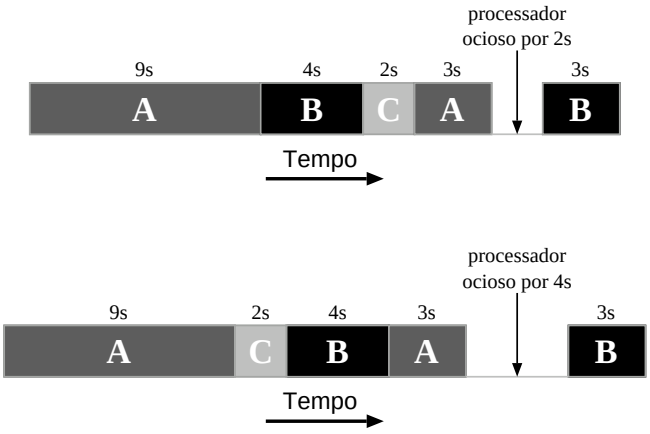
Nome -
Assinatura -

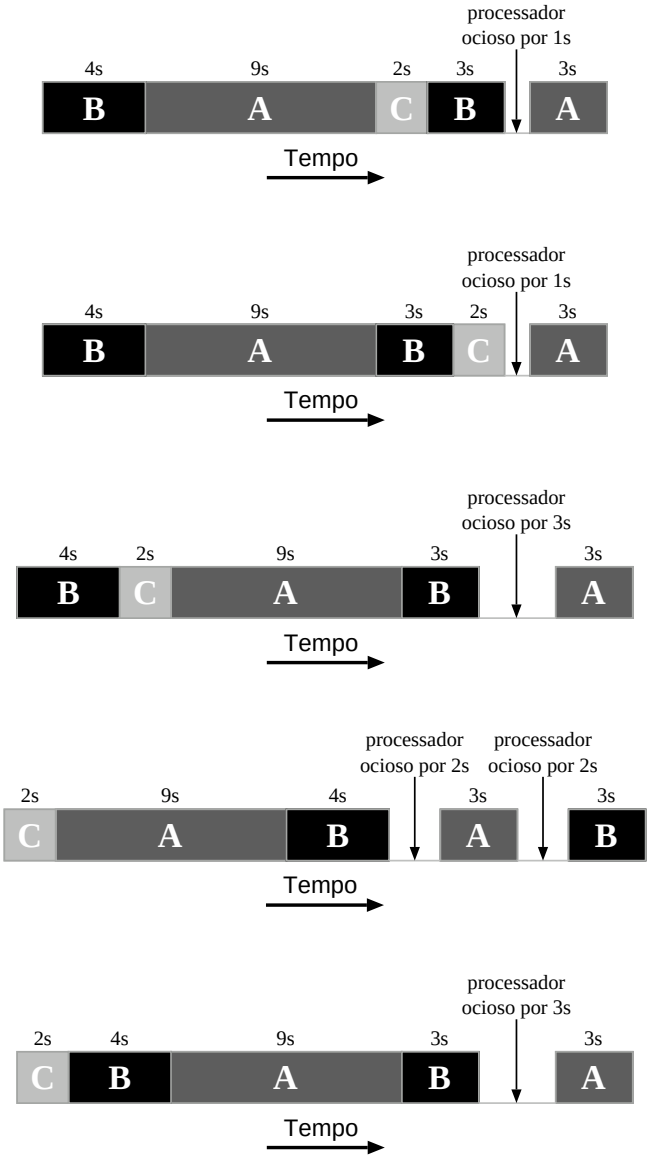
---

1. (2,0) Suponha que somente três programas, A, B e C, estejam em execução no processador do computador. O programa A foi o primeiro a executar no processador: executou por 12s, tendo precisado fazer uma operação de E/S, com duração de 6s, após os primeiros 9s de execução. O programa B, que executou por 7s, também precisou fazer uma operação de E/S, com duração de 7s, após os primeiros 4s de execução. Finalmente, o programa C executou por 2s sem fazer operações de E/S. Se o sistema operacional não usar a multiprogramação, qual será o tempo de ociosidade do processador? Agora, se o sistema usar a multiprogramação, qual será o maior tempo de ociosidade do processador, caso sejam consideradas todas as possíveis ordens

de execução dos processos? Justifique a sua resposta.

Resp.: -Se o sistema operacional não usar a multiprogramação, então um programa executará até terminar a sua execução e, devido a isso, o processador ficará ocioso durante o tratamento de todas as operações de E/S desse programa. Como no exemplo da questão somente os programas A e B fazem operações de E/S, então o tempo total de ociosidade do processador será igual à soma do tempo dessas operações de E/S, ou seja, será de 6s+7s= 13s.
-Agora, se a multiprogramação for usada, poderemos evitar a ociosidade do processador quando o programa em execução fizer uma operação de E/S, ao colocar um outro programa para executar no processador que não esteja executando uma operação de E/S. Na nossa resposta vamos supor que a multiprogramação somente é usada para evitar a ociosidade do processador quando operações de E/S são feitas, mas você pode, na sua resposta, também supor que o programa execute por um dado intervalo fixo de tempo antes de ser substituído por um outro. Nas figuras a seguir mostramos todas as possíveis ordens de execução dos programas A, B e C de acordo com a nossa suposição e, pelas figuras, vemos que o maior tempo de ociosidade será de 4s.





2. (1,5) Qual é o número máximo de chamadas ao sistema operacional por segundo que o computador pode executar, supondo que cada chamada ao sistema requeira 750 instruções, incluindo a de TRAP e todas as necessárias à troca de contexto, e que o processador possa executar 2 700 000 instruções por segundo, sendo que 5/6 da capacidade do processador são usados para executar códigos do usuário? Justifique a sua

resposta.

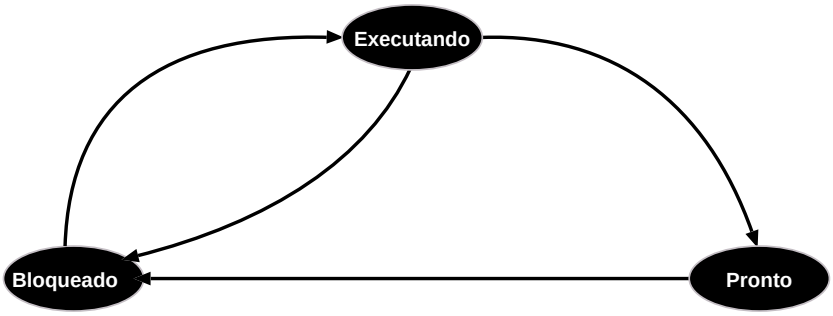
Resp.: Se denotarmos por x o número máximo de chamadas ao sistema operacional, então o computador precisa usar $750x$ instruções para fazer essas chamadas. Agora, como 5/6 da capacidade do processador estão disponíveis para executar códigos do usuário, então 1/6 da capacidade está disponível para executar as chamadas ao sistema, ou seja, o processador pode executar $2\,700\,000/6 = 450\,000$ instruções por segundo para tratar as chamadas. Logo, temos que $750x = 450\,000$ e portanto $x = 600$, significando que podemos executar no máximo 600 chamadas ao sistema operacional por segundo.

3. (2,0) Suponha que o sistema operacional esteja executando diretamente sobre o hardware de um computador cujas operações de E/S demorem 0,35ms. Suponha ainda que um processo tenha executado por 9s e que, durante a sua execução, tenha feito 6 000 operações de E/S. Se o sistema operacional agora executar sobre uma máquina virtual que reduza a velocidade do processador em $x\%$ e a velocidade das operações de E/S em 75%, qual será o valor de x se o processo executar no processador virtual por $17,025$s e fizer novamente 6 000 operações de E/S? Justifique a sua resposta.

Resp.: Como cada operação de E/S demora 0,35ms e como o processo faz 6 000 operações, então 2 100ms do tempo de execução de 9s ou 9 000ms desse processo são gastos em operações de E/S, quando ele executa no sistema operacional sobre o hardware do computador. Logo, o processo executa no processador do hardware por $9\,000 - 2\,100 = 6\,900$ms. Note que a velocidade do processador ser reduzida em $x\%$ significa que a velocidade do processador virtual é $(100 - x)\%$ da velocidade do processador real, o que por sua vez significa, durante os 6 900ms, que somente $(100 - x)\%$ das instruções serão executadas. Com isso, quando o processo executar no sistema operacional sobre a máquina virtual, o tempo de execução dele no processador virtual será de $\frac{6\,900}{\frac{100-x}{100}} = \frac{690\,000}{100-x}$ms. Além disso, como o tempo gasto por cada operação de E/S na máquina virtual é de $0,35/0,25 = 1,4$ms, pois ela reduz a velocidade das operações de E/S em 75% e, como o processo

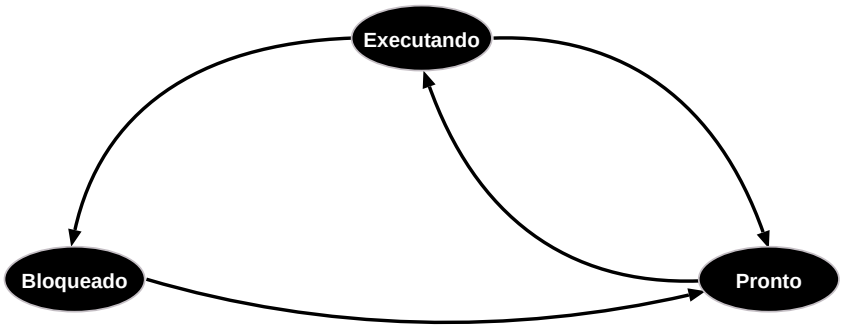
executa ainda 6 000 operações de E/S, então o processo gastará agora 8 400ms para executar as operações de E/S, o que significa que ele executará no processador virtual por 8 625ms, pois o seu tempo total de execução na máquina virtual será de 17, 025s ou 17 025ms. Igualando o tempo obtido anteriormente com o tempo de execução do processo no processador virtual obtido a partir do tempo de execução desse processo no processador real, temos que $\frac{690\,000}{100-x} = 8\,625$, ou seja, $x = 20$, o que significa que a máquina virtual reduz a velocidade do processador real em 20%.

4. (1,0) Considere o diagrama de transição entre os estados dos processos dado na figura a seguir. O diagrama está correto? Justifique a sua resposta.



Resp.: Não, existem três erros no diagrama dado na figura. O primeiro erro é a aresta orientada do estado **Bloqueado** para o estado **Executando**. Essa aresta está incorreta porque o processo em execução e talvez algum outro processo no estado **Pronto** podem ser mais prioritários do que o processo que acabou de ser desbloqueado. Note que, caso o processo seja de fato o mais prioritário, ele passará do estado **Pronto** para o estado **Executando** quando o escalonador for executado. O segundo erro é a aresta orientada do estado **Pronto** para o estado **Bloqueado**. Como um processo no estado **Pronto** não está em execução, ele não pode ser suspenso e depois bloqueado para esperar por um evento externo. Como observamos no primeiro erro, essa aresta estaria correta se fosse orientada do estado **Bloqueado** para o estado **Pronto**. Finalmente, foi omitida uma aresta muito importante, a orientada do estado **Pronto** para o estado **Executando**, pois sempre

que o escalonador é executado, o processo no estado **Executando**, se deixar de ser o mais prioritário, passará ao estado **Pronto** e o processo mais prioritário no estado **Pronto** passará ao estado **Executando**. Na figura a seguir damos o diagrama correto.



5. (2,0) Suponha que dois processos, A e B, compartilhem uma pilha, usada para armazenar números. Suponha ainda que essa pilha, inicialmente vazia, possa armazenar até n números, e que não exista uma variável para contabilizar a quantidade de números armazenados na pilha. O processo A continuamente insere x números de cada vez na pilha. Já o processo B continuamente remove y números de cada vez da pilha. Como os semáforos de contagem devem ser usados para garantir a correta execução dos processos A e B? Justifique a sua resposta.

Resp.: Precisamos usar três semáforos para garantir o correto funcionamento dos processos A e B: dois de contagem, *cheia* e *vazia*, e um binário, *acesso*. O semáforo binário *acesso* garante o acesso exclusivo de um processo à pilha, e é inicializado com 1 porque inicialmente nenhum processo está acessando a pilha. Já o semáforo de contagem *cheia* conta o número de entradas usadas na pilha, e é usado para bloquear o processo B se a pilha estiver vazia. Finalmente, o semáforo de contagem *vazia* conta o número de entradas vazias na pilha, e é usado para bloquear o processo A quando a pilha estiver cheia. Como inicialmente a pilha está vazia, então os semáforos *cheia* e *vazia* são inicializados, respectivamente, com 0 e n. A seguir mostramos os códigos para os processos A e B, sendo que a função *removetopo*() remove e retorna o número no topo da pilha, e a função *inseretopo*(*dado*) insere *dado* no topo da pilha.

```
void ProcessoA(void)
{
  while (1);
  {
    // Código para gerar x números e salvá-los no vetor
    // num.
    // Garante que x entradas da pilha estejam vazias.
    for( int i = 0; i < x; i++)
      P(vazia);
    // Garante o acesso exclusivo à pilha.
    P(acesso);
    // Insere cada elemento gerado na pilha.
    for( int i = 0; i < x; i++)
      inseretopo(num[i]);
    // Libera o acesso exclusivo à pilha.
     V(acesso);
    // Usa a operação V sobre cheia para registrar que
    // x elementos foram inseridos na pilha.
    for( int i = 0; i < x; i++)
      V(cheia);
  }
}
```

```
void ProcessoB(void)
{
  while (1);
  {
    // Garante que y entradas da pilha estejam preenchidas.
    for( int i = 0; i < y; i++)
      P(cheia);
    // Garante o acesso exclusivo à pilha.
    P(acesso);
    // Código para alocar um vetor num com y entradas.
    // Remove y elementos do topo da pilha e os salva no
    // vetor num.
    for( int i = 0; i < y; i++)
      num[i] = removetopo();
    // Libera o acesso exclusivo à pilha.
     V(acesso);
    // Usa a operação V sobre vazia para registrar que
    // y elementos foram removidos da pilha.
    for( int i = 0; i < y; i++)
      V(vazia);
    // Código para usar os y números lidos em num.
}
```

6. (1,5) Suponha que três processos, A, B, e C, estejam em execução no computador. Suponha ainda que o sistema operacional use o algoritmo por *round robin*, com um quantum de x unidades de tempo, ao escalonar os processos. Quais serão os tempos decorridos desde o início até o término dos processos A, B e C, supondo que A, B e C precisem executar no processador por, respectivamente, ax, $4ax$ e $2ax$, unidades de tempo, onde $a > 0$, e supondo que o escalonador inicialmente escolheu (nos três primeiros quanta) A, depois C e finalmente B?

   Resp.: Como vimos na aula 6, o algoritmo por *round robin* executa os processos no estado **Pronto** de modo alternado, sendo que cada processo executa por um tempo igual ao quantum. Além disso, um processo somente volta a executar, se necessário, por mais um quantum, quando todos os outros processos no estado **Pronto** também já tenham

executado por um quantum. Pelo enunciado, vemos que a ordem de execução dos processos é a dada na figura a seguir pois, como cada processo precisa executar por um tempo que é múltiplo do tempo do quantum, então notamos que A, B e C precisam executar por, respectivamente, a, $4a$ e $2a$ quanta. Por essa ordem de execução, e lembrando que um quantum tem x unidades de tempo, vemos que os tempos decorridos desde o início até o término dos processos A, B e C são de, respectivamente, $(3a-2)x$, $(3a-2)x+2ax+2ax$ e $(3a-1)x+(2a-1)x$, ou seja, são de, respectivamente, $3ax-2x$, $7ax-2x$ e $5ax-2x$ unidades de tempo.