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 - Primeiro Semestre de 2008

Nome -
Assinatura -

---

1. (1.5) Qual é a principal diferença entre a terceira e a quarta gerações de sistemas operacionais?

   Resp.: A principal diferença entre a terceira e a quarta gerações foi, nesta última, o surgimento dos computadores pessoais, o que permitiu a indivíduos poderem ter máquinas próprias. Esta mudança permitiu a volta dos sistemas monousuários, pois tornou possível que uma máquina fosse usada por um único usuário. Favoreceu também o desenvolvimento das interfaces gráficas para facilitar o uso das máquinas pelos usuários. Finalmente, como passou a ser mais necessária a comunicação entre as máquinas dos indíviduos, foi natural o surgimento das redes de computadores, para que estes pudessem trabalhar cooperativamente.

2. (1.5) Como podemos acessar um sistema de arquivos externo, como o

de *pendrives* ou CDs/DVDs? Justifique a sua resposta.

Resp.: Para podermos acessar qualquer sistema de arquivos externo, devemos usar o conceito de **montagem** do sistema operacional. Para fazer a montagem do sistema de arquivos de *pendrives* ou CDs/DVDs, devemos inicialmente definir um diretório, no sistema de arquivos do sistema operacional, chamado de o **ponto de montagem**, a partir do qual os arquivos do sistema de arquivos externo serão acessados. Depois disso, devemos fazer uma chamada ao sistema operacional, que fará a montagem do sistema de arquivos externo. Após esta chamada ser executada, o sistema de arquivos externo estará montado no sistema de arquivos do sistema operacional, e a sua hierarquia de diretórios poderá ser acessada a partir do ponto de montagem.

3. (1.5) Considere um sistema operacional do tipo monolítico e um do tipo cliente-servidor (micronúcleo). Descreva uma chamada de um processo ao sistema operacional em cada um dos dois modelos.

Resp.: Em um sistema monolítico, o núcleo do sistema operacional é o responsável pela execução das chamadas ao sistema operacional. Logo, quando um processo faz uma chamada ao sistema operacional, uma instrução TRAP é primeiramente executada para alternar do modo usuário, em que os processos são executados, para o modo supervisor, em que o núcleo é executado. Depois disso, o código do núcleo responsável por tratar a chamada é executado. Finalmente, o controle é passado ao processo após a execução da chamada, sendo que o núcleo passa ao processo o resultado desta chamada. Já em um sistema cliente-servidor, todas as chamadas ao sistema operacional são executadas através do envio de mensagens do processo que faz a chamada (um cliente) para o processo responsável por executar a chamada (o servidor). Ambos os processos executam no modo usuário, não sendo portanto necessário executar uma instrução TRAP. Neste caso, o processo cliente envia uma mensagem ao processo servidor requisitando a execução de um serviço (que no sistema monolítico era tratado pela chamada ao sistema) e, depois de executar este serviço, o servidor envia uma mensagem ao cliente com o resultado da execução do serviço. Note

que o núcleo do sistema (que é chamado de **micronúcleo**) somente faz as trocas de mensagens entre os processos executando no modo usuário. O acesso aos dispositivos físicos é feito através de mensagens especiais enviadas pelos processos responsáveis pelo gerenciamento destes dispositivos ao micronúcleo.

4. (1.5) Suponha que dois processos, o *firefox* e o *acroread*, estão em execução no sistema, pois o usuário está acessando um site na WEB e lendo um documento. Suponha ainda que o *firefox* está em execução no processador. Descreva, em detalhes, o que ocorrerá quando o escalonador decidir que o *acroread* deve passar a executar no processador.

   Resp.: Quando o escalonador decidir escolher o *acroread* para executar no processador, primeiramente o contexto do *firefox* será salvo em sua entrada na tabela de processos. Isso permitirá que ele possa futuramente ser reiniciado exatamente onde parou sua execução em seu código, como se nunca tivesse sido suspenso. Note que o escalonador será chamado se o *firefox* for bloqueado, ou se uma interrupção for gerada pelo temporizador que contabiliza o tempo em que o *firefox* pode executar no processador. Depois disso, o contexto do processo *acroread* será copiado da sua entrada da tabela de processos, permitindo que ele reinicie a sua execução exatamente no ponto em que parou antes de ser suspenso. Estes dois passos descritos anteriormente compõe o que chamamos de a **troca de contexto**. Finalmente, após copiarmos o contexto do *acroread*, o processador será alocado a este processo, que começará a executar no processador até ser bloqueado ou o seu tempo de uso do processador expirar.

5. (2.5) Suponha que temos n blocos de memória que podem ser usados pelos processos em execução como uma memória *cache* para blocos do disco. Como podemos garantir o uso correto destes blocos de memória, isto é, garantir que nenhum bloco seja utilizado por mais de um processo de cada vez, se usarmos semáforos binários? Justifique a sua resposta.

   Resp.: Deveremos usar dois semáforos binários: *blocos*, para o acesso

exclusivo aos blocos de memória, garantindo que um bloco seja alocado a somente um único processo, e *bloqueio*, para bloquear os processos caso não existam blocos livres suficientes. Note que um processo deverá obter todos os blocos de que precisa de uma só vez, pois caso ele obtenha mais blocos antes de liberar os que possui, a solução não funcionará para todos os casos. Por exemplo, suponha que n é par e que dois processos, A e B, obtiveram cada um $n/2$ blocos de memória. Se ambos tentarem obter mais um bloco de memória antes de liberar os $n/2$ que possuem, ambos ficarão bloqueados para sempre no semáforo *bloqueio*, pois um dos processos precisa liberar um de seus blocos para que depois o outro processo seja desbloqueado. Então, vamos supor que um processo obtém todos os blocos de memória de que precisa de uma só vez, e que depois não pede mais blocos adicionais (embora possa voltar a fazer pedidos após liberar todos os blocos que possui). O valor inicial do semáforo *blocos* será de 1, pois inicialmente nenhum processo está tentando obter um dos blocos de memória. Já o valor inicial do semáforo *bloqueio* será de 0, pois é usado para bloquear os processos caso não existam blocos disponíveis. Vamos supor também que temos uma variável *numblocos*, com o número de blocos de memória disponíveis. Como inicialmente nenhum bloco está sendo utilizado, o valor inicial de *numblocos* será de n. Também precisaremos definir uma variável *numbloqueados*, que conterá o número de processos bloqueados no semáforo *bloqueio*, e que será inicializada com o valor 0, pois nenhum processo ainda terá sido bloqueado. Esta variável será usada para garantir que somente executemos a operação **V** sobre *bloqueio* caso existam processos bloqueados, pois o semáforo *bloqueio* deve sempre bloquear, e isso somente ocorrerá se o seu valor for sempre 0. A seguir apresentamos os códigos da função $ObterBlocos(x)$, que deverá ser usada quando o processo deseja obter $x \leq n$ blocos de memória, e da função $LiberarBlocos(x, blocos)$, que deverá ser usada quando o processo não desejar mais usar os x blocos que possui. Nestes códigos, a função $PegarBlocos(x)$ retorna x blocos de memória e a função $LiberarBlocos(blocos)$ libera os blocos dados em *blocos*.

```
void ObterBlocos(x, blocos)
{
  bool tentar;
  tentar = true;
```

```
  while (tentar)
  {
    P(blocos);
    if (numblocos < x)
    {
      numbloqueados = numbloqueados + 1;
      V(blocos);
      P(bloqueio);
    }
    else
    {
      blocos = PegarBlocos(x);
      numblocos = numblocos − x;
      V(blocos);
      tentar = false;
    }
}

void LiberarBlocos(x, blocos)
{
  P(blocos)
  LiberarBlocos(blocos);
  numblocos = numblocos + x;
  if (numbloqueados > 0)
  {
    V(bloqueio);
    numbloqueados = numbloqueados − 1;
  }
  V(blocos);
}
```

6. (1.5) Suponha que um sistema operacional usa o algoritmo de escalonamento por *round robin*, e suponha que um processo A executou exclusivamente no processador, isto é, nenhum outro processo estava no estado pronto durante a sua execução. Suponha ainda que a quanta se passaram entre o início e o término da execução de A. Qual seria o novo tempo de término de A se um outro processo B, com tempo de execução de b quanta, passasse ao estado pronto durante o primeiro

quantum alocado ao processo A? Justifique a sua resposta.

Resp.: O novo tempo de término de A dependerá dos valores de a e b. Se $a > b$, então o tempo de término será de $a + b$ quanta. Neste caso, a execução de A será intercalada com a de B até B terminar a sua execução, ou seja, por $2b$ quanta. Depois disso, A finalizará a sua execução, executando exclusivamente no processador por $a - b$ quanta. Finalmente, se $a \leq b$, então o tempo de término será de $2a - 1$ quanta. Agora, a execução de A será intercalada com a de B até o término da execução de A, e A começou a executar antes de B (note que B executará exclusivamente no processador a partir do quantum $2a$).