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 2007

Nome -
Assinatura -

---

1. (1.25) Nos primeiros computadores, cada byte de dados em um dispositivo de E/S era lido ou escrito diretamente pelo processador (isto é, não havia nenhum mecanismo para realizar a transferência de bytes diretamente entre o dispositivo e a memória). Que implicações esse arranjo tem para a multiprogramação?

   Resp.: Quando não existir uma DMA no hardware, o processador será o responsável pelas transferências de dados entre os dispositivos físicos e a memória principal do computador. Com isso, o processador tenderá a estar ocupado na maior parte do tempo, mesmo que os processos executem operações de E/S com freqüência. Isso ocorrerá porque para a maior parte dos dispositivos (com exceção dos mais lentos, como, por exemplo, um modem ligado a uma linha telefônica) o tempo da transferência dos dados entre a memória e o dispositivo dominará o tempo total da operação de E/S. Logo, o principal ganho da multiprogramação, que é o de evitar que o processador fique ocioso quando

operações de E/S são executadas, será reduzido, e a multiprogramação essencialmente permitirá a execução de vários processos no processador.

2. (1.25) Os pipes são indispensáveis? Haveria uma grande perda de funcionalidade se eles não estivessem disponíveis?

   Resp.: Não, os pipes não são tão essenciais, como veremos a seguir. Podemos simular um pipe entre dois processos usando um arquivo, em que o primeiro processo salvará a sua saída ao final do arquivo, e o segundo processo lerá a sua entrada do arquivo. Mas isso dificultará a programação, pois o processo que ler os dados do arquivo deverá manter um registro da posição do arquivo da qual ele leu os dados pela última vez. Além disso, teremos a inconveniência de usar um arquivo e de desperdiçar o espaço do disco. Também, isso talvez não funcione em todos os casos, pois um arquivo armazenado no disco tem um tamanho limitado pelo espaço disponível neste disco.

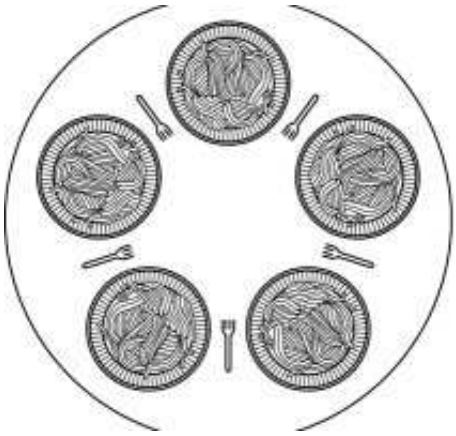
3. (1.5) Como são implementadas as chamadas ao sistema operacional em um sistema baseado em micronúcleo?

   Resp.: Em um sistema baseado em micronúcleo, as chamadas ao sistema operacional são implementadas por processos servidores executando no modo usuário, ao invés de usarmos a instrução TRAP para chamar um código dentro do núcleo do sistema. Neste caso, quando um processo executar uma função da biblioteca que precise de algum serviço do sistema operacional, uma mensagem será enviada para o servidor responsável pelo serviço. Depois de receber a mensagem, o servidor fará o que for necessário para executar o serviço. Como pôde ser visto na Aula 3, o servidor poderá enviar mensagens aos outros servidores para auxiliá-lo a executar o serviço requisitado. Depois de executar o serviço, o servidor enviará uma mensagem ao código da função da biblioteca com o resultado do serviço requisitado. Finalmente, após fazer alguns processamentos adicionais, a função passará ao processo o resultado do serviço executado pelo servidor.

4. (2.0) Um processo que realize uma computação estritamente seqüencial pode ser visto como um único fluxo de controle. No entanto, no caso mais geral, é possível que um processo possa ter mais de um fluxo de controle, correspondendo a ações executadas concorrentemente. Cada um desses fluxos de controle é chamado de **thread** (fio). Assim como os processos, as threads são escalonadas para execução pelo sistema operacional. Porém, o escalonamento das threads é mais leve que o dos processos, já que seu contexto é bem menor. A pilha deve ou não fazer parte desse contexto a ser salvo no caso das threads? Justifique a sua resposta.

   Resp.: Como foi visto na disciplina de Organização de Computadores, o objetivo da pilha é o de armazenar várias informações referentes ao fluxo de execução das instruções no processador. Dentre estas informações, temos as variáveis locais do procedimento atualmente em execução no processador, e os endereços de retorno das chamadas que foram executadas e que ainda não foram terminadas. Como um processo com múltiplas threads possui múltiplos fluxos de controle, e como cada um deles pode armazenar variáveis locais na pilha e chamar procedimentos, deveremos ter uma pilha para cada thread do processo, que deverá ser salva no contexto desta thread. Note que enquanto o escalonador estiver escolhendo threads do mesmo processo, somente o contexto da thread deve ser salvo, pois todas as threads de um processo compartilham o contexto deste processo. O contexto do processo somente deverá ser salvo quando o escalonador escolher uma thread de um outro processo, pois esta thread não compartilhará o contexto do processo da thread que estava em execução. Logo, o escalonamento de threads do mesmo processo é mais leve do que o de threads pertencentes a processos diferentes, pois o contexto a ser salvo é menor. Note também que se a pilha não fosse salva no contexto, deveríamos executar as threads do processo seqüencialmente, o que não seria lógico, pois o objetivo de termos múltiplas threads é exatamente o de permitir que elas executem concorrentemente. Perceba que no caso de termos mais de um processador ou de o único processador possuir múltiplos contadores de programa, as threads poderão de fato executar paralelamente.

5. (2.0) Um problema clássico da comunicação de processos é o problema dos filósofos. Neste problema, cada filósofo passa por períodos alternados de comer e de pensar, e estão dispostos em uma mesa circular. Nesta mesa existe um prato de espaguete a frente de cada filósofo e um garfo entre dois pratos consecutivos, como podemos ver na figura a seguir, em que temos 5 filósofos (e 5 garfos e pratos de espaguete):



Quando um filósofo está com fome, ele tenta pegar os garfos à sua esquerda e à sua direita, pois o espaguete está muito escorregadio e por isso são necessários dois garfos para comê-lo. Se conseguir pegar os garfos, o filósofo come por um tempo um pouco do espaguete e depois recoloca os garfos na mesa e volta a pensar.

A seguir é dada uma solução para o problema com n filósofos baseada em n semáforos binários, sendo que o semáforo $fork[i]$ está associado ao garfo i, $1 \leq i \leq n$. Os semáforos são todos inicializados com o valor 1. Esta solução funcionará se os processos que implementam os filósofos executarem a função *philosopher* dada a seguir, sendo que cada um destes processos está associado a um identificador i, $1 \leq i \leq n$? Justifique a sua resposta.

```
void philosopher(int i)
{
  while (1) {
    pensar();
    P(fork[i]);
    P(fork[(i+1) % n]);
    comer();
    V(fork[i]);
    V(fork[(i+1) % n]);
  }
}
```

Resp.: A solução não funcionará, pois podemos ter um impasse, como veremos a seguir. Suponha que todos os processos não são bloqueados ao executarem a primeira operação **P** do procedimento *philosopher*, isto é, a segunda operação **P** somente será executada após todos os processos executarem a primeira operação **P**. Neste caso, depois de cada processo executar a primeira operação **P**, o valor de cada um dos semáforos binários $fork[i]$, $1 \leq i \leq n$, será igual a 0. Com isso, todos os processos serão bloqueados ao executarem a segunda operação **P**, e nunca mais serão desbloqueados, pois nenhum deles poderá executar a operação **V** sobre um dos semáforos binários, e somente a execução desta operação poderá desbloquear um dos processos.

6. (2.0) Suponha que um processo requer, em média, um tempo T de execução antes que se bloqueie por E/S. Suponha também que uma comutação entre processos requer um tempo S, o qual é efetivamente desperdiçado (*overhead*). Para o escalonamento por *round robin* com quantum Q, dê uma equação para a eficiência do processador (fração do tempo usado para executar os processos) para cada um dos seguintes casos:

   (a) $Q = \infty$.

   Resp.: Como T é menor do que Q (pois Q é infinitamente grande), então o processo executará no processador por um tempo T antes de bloquear, e depois será necessário fazer uma troca de contexto,

o que irá requerer um tempo S. Logo, a eficiência do processador será de $T/(T+S)$.

(b) $Q > T$.

Resp.: Se Q for finito mais maior do que T, a eficiência será a mesma do Item 6a, isto é, $T/(T+S)$, pois o processo executará, assim como antes, por um tempo T antes de ser bloqueado.

(c) $S < Q < T$.

Resp.: Se Q for menor do que T, então teremos que fazer T/Q trocas de contexto até que o processo seja finalmente bloqueado. Com isso, o tempo gasto com as trocas de contexto será de ST/Q. Logo, a eficiência do processador será de $T/(T+ST/Q)$, isto é, $Q/(Q+S)$.

(d) $Q = S$.

Resp.: Assim como no caso anterior, teremos que executar $T/Q = T/S$ trocas de contexto. Logo, para obter a eficiência do processador, basta substituir, na equação dada no Item 6c, Q por S. Ao fazer isso, vemos que a eficiência do processador será de $S/(S+S) = S/2S = 1/2$.

(e) Q perto de zero.

Resp.: Quando Q tender a zero, o número de trocas de contexto tenderá a infinito. Com isso, a eficiência do processador tenderá a zero, pois quanto mais Q tender a 0, mais tempo será gasto com as trocas de contexto.