MAC 2301 LABORATÓRIO DE ESTRUTURAS DE DADOS

DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO – USP

SEGUNDO EXERCÍCIO-PROGRAMA PRAZO DE ENTREGA: ATÉ 14/05/02

Multiprogramação

1. Introdução

O objetivo desse exercício-programa é fazer um programa em `C` que simule o funcionamento de um sistema simplificado de computação com multiprogramação.

Nas Seções 2 e 3 a seguir encontra-se uma consideração geral sobre simulação e multiprogramação. Para fazer este exercício-programa estas seções podem ser lidas superficialmente. Na Seção 4 descrevemos o modelo simplificado de computação que deverá ser simulado. A Seção 5 contém as especificações da implementação do modelo, inclusive as estruturas de dados que deverão ser utilizadas pelo seu programa. As especificações da saída do seu programa encontram-se na Seção 6.

Você deve entregar um disquete contendo o programa fonte e também as listagens do programa e de sua saída até o dia 14 de maio (guarde cópias do programa que você entregar).

2. Simulação

Uma das áreas onde filas têm muitas aplicações é simulação. *Simulação* é o processo de formar um modelo abstrato a partir de uma situação real para que possamos estudar o impacto de modificações e o efeito de introduzir diferentes estratégias na situação em questão. O principal objetivo de um programa para simulação é ajudar o usuário, freqüentemente especialistas em pesquisa operacional ou analistas de sistemas, a prever o que acontecerá a uma situação física sob algumas hipóteses simplificadoras. Simulação é uma ferramenta muito poderosa se aplicada corretamente.

A principal vantagem da simulação é permitir experimentação sem de fato modificar a situação. Áreas como operações militares estão mais seguras ao simular do que realmente testar.

Em uma simulação orientada-por-relógio (*time-driven*) temos um relógio principal que toma conta do ritmo da simulação. Permitimos que o relógio seja incrementado uma unidade por vez. Depois de cada incremento no

Date: 11 de abril de 2002.

tempo do relógio o programa deve examinar cada evento para decidir se ele deve ocorrer ou não.

Em uma simulação orientada-por-eventos (*event-driven*) o programa examina todos os eventos do modelo e determina qual deve ser o próximo a ocorrer. Para que façamos isto cada evento deve ter o seu próprio relógio para obtermos a próxima vez que ele vai ocorrer. Depois que o relógio principal é atualizado, a atividade associada com o evento é executada. Este método é essencialmente discreto (i.e. não-contínuo).

[O que foi brevemente tratado nesta seção foi extraído do Capítulo 3 de [2].]

3. Sistemas de multiprogramação

Um conceito fundamental em sistemas operacionais é o conceito de processo. Um *processo* é basicamente um programa em execução. Ele consiste de um programa executável, os seus dados e pilha, o seu stack pointer e registradores, enfim todas as informações necessárias para executar o programa.

Periodicamente o sistema operacional decide se a execução de um processo deve ser interrompida e a execução de um outro processo deve ser iniciada—pela razão do primeiro já ter tido mais do que a sua 'fatia' de tempo de CPU. Em um sistema de *multiprogramação* a CPU fica se alternando entre a execução de vários processos, cada um por dezenas ou centenas de milisegundos.

Um processo pode estar em um dos seguintes estados:

1. *Running*: usando a CPU naquele instante;
2. *Ready*: pronto para ser executado, temporariamente parado para que outro processo possa ser executado; e
3. *Blocked*: impossibilitado de ser executado até que algum evento externo ocorra.

Em um sistema de multiprogramação temos frequentemente a situação onde vários processos estão prontos para serem executados. Quando mais de um processo está *ready*, o sistema operacional deve decider qual processo deve ser executado primeiro. A parte do sistema operacional que toma esta decisão é chamada de *scheduler*, e o algoritmo que é usado é chamado de *scheduler algorithm*. A cada interrupçao do relógio o sistema operacional toma o controle e decide se o processo que está sendo executado deve continuar a ser executado ou deve ser *suspenso* para que outro processo passe a ser executado. A estratégia que permite que um processo que está sendo executado seja suspenso temporariamente é chamada de *preemptive schedule*.

Existem `scheduling algoritms` que assumem que todos os processos são iguais. Entretanto, existem muitas pessoas que possuem e administram centros de computação que discordam disto. Por exemplo, em um centro de computação de uma universidade pode ser dada a mais alta prioridade aos processos de diretores, depois aos de professores, depois aos de secretarias, depois aos de porteiros, ... e finalmente aos de alunos. Isto é chamado de um *priority schedule* (alguns podem chamar isto de injustiça).

Para evitar que processos que têm alta prioridade fiquem sendo executados indefinidamente o `scheduler` pode baixar a prioridade do processo que esta sendo executado a cada interrupção do relógio.

Alguns `priority schedulers` fazem uso de *classes de prioridade*. Os processos na classe de prioridade mais baixa podem ser executados sem serem suspensos por no máximo uma QUANTA (uma quantidade de tempo de CPU). Processos na próxima classe de prioridade pode ser executado por no máximo duas QUANTAS, e assim por diante. Sempre que um processo usa toda as QUANTAS alocadas para ele este processo é rebaixado de classe.

Quando um processo deseja imprimir um arquivo, ele coloca o nome do arquivo em um diretório especial chamado de `spooler directory`. Um outro processo, o `printer daemon`, periodicamente verifica se existe algum arquivo a ser impresso, se existir ele imprime o arquivo e remove o nome do arquivo do diretório `spooler`.

[O que foi brevemente tratado nesta seção diz respeito principalmente ao sistema operacional UNIX[1] e foi extraído de [1].]

4. Descrição do modelo simplificado a ser simulado

Este exercício-programa trata de uma aplicação de filas na simulação de um *timesharing computer system*. O modelo a ser simulado é esquematizado pela Figura 1.

Um fato importante a ser notado através da figura é que vários usuários e seus processos estão compartilhando os recursos do computador simultaneamente. Como temos apenas uma CPU e cinco impressoras esses recursos devem ser compartilhados entre os usuários.

Cada processo executado no computador pertence a uma das classes: `Classe 1` ou `Classe 2`.

Quando um processo é ativado, ele é colocado na fila de sua classe, onde fica esperando até poder entrar na fila da CPU.

As regras para um processo entrar na fila da CPU são as seguintes[2]:

a) Um processo de `Classe 1` sempre tem prioridade.
a) Um processo de `Classe 2` só entra na fila da CPU se a fila da `Classe 1` estiver vazia ou os últimos quatro processos que entraram na fila da CPU eram de `Classe 1`.

A utilização da CPU é dividida em intervalos de tempo (o chamado *ciclo de intervenção do sistema timesharing* que é igual a um QUANTA) com duração máxima fixada. No início de cada ciclo, a CPU passa a executar o processo que está no início de sua fila. Se, ao término do atual ciclo, a execução desse

[1] UNIX é marca registrada da AT&T (Bell Laboratories).
[2] Este `scheduling algorithm` corresponderá ao *shortest job first*.

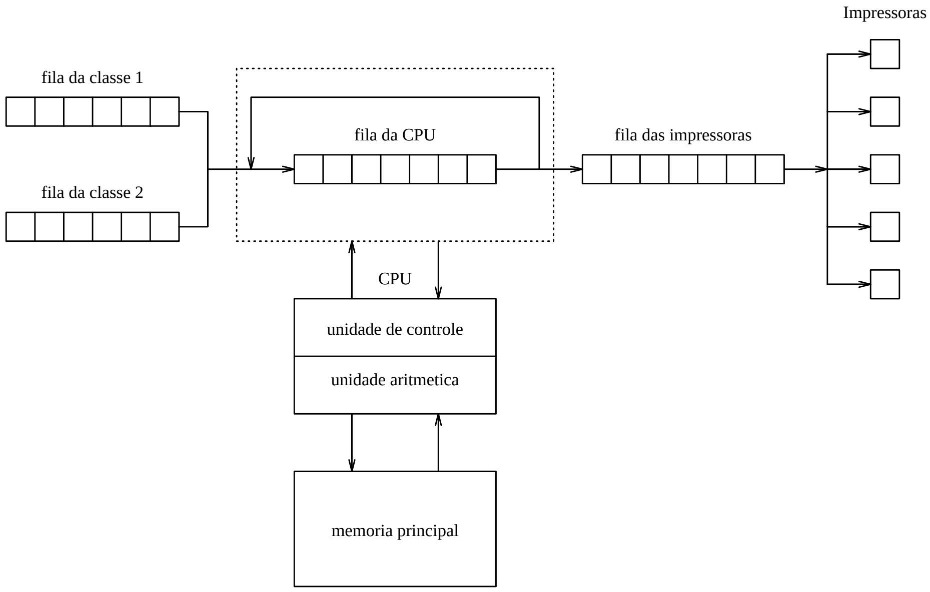

FIGURA 1. Ilustração do sistema timesharing que será simulado.

processo não for completada, ele retorna para o fim da fila da CPU. Se a execução de um processo terminar durante um ciclo, ele vai para a fila das impressoras, o ciclo termina e a CPU passa a executar o processo que está no início de sua fila.

Existe uma única fila para as cinco impressoras. Quando alguma impressora fica disponível, o primeiro processo dessa fila irá utilizar esta impressora, a menos que exista algum processo que já esteja no sistema há mais do que dez minutos.

5. Especificações da implementação do modelo

Faça um programa em `C` que simule o funcionamento desse sistema de multiprocessamento com as seguintes considerações:

1. Tempo total da simulação: 1 hora.
2. Ciclo de intervenção do sistema de *timesharing*: 100 milisegundos. Isto significa que a cada 100 milisegundos o `scheduler` decide qual é o próximo processo que deverá ser executado.
3. O `scheduler` também deve decider qual é o próximo processo que deverá ser executado imediatamente após o término da execução de qualquer processo, mesmo que o término tenha ocorrido antes de se completar um ciclo de intervenção (ou seja, 100 milisegundos).
4. Tempo gasto pelo `scheduler` para decidir qual é o próximo processo que deverá ser executado: 0.25 milisegundos.
5. Velocidade das impressoras: 3000 linhas por minuto.
6. Número máximo de processos na fila da CPU a cada instante: 7.

7. Cada classe de processos tem permissões diferentes com relação ao limite máximo de execução na CPU e ao número máximo de linhas a serem impressas. Estas restrições estão descritas na tabela a abaixo:

Classe	CPU (em segs.)	Linhas impressas
1	5	500
2	30	10000

Um novo processo deve ser ativado a cada 5 segundos, com probabilidade 0.7 que seja de `Classe 1` e probabilidade 0.3 que seja da `Classe 2`. Ao ativar um processo, gere aleatoriamente o tempo de CPU e o número de linhas que esse processo irá imprimir (dentro dos limites de sua classe). Atribua ainda um número de identificação a cada processo.
8. As filas da `Classe 1` e `Classe 2` devem ser implementadas em listas lineares ligadas com cabeça-de-lista.
9. A fila da CPU deve ser implementada numa lista circularmente ligada (com ou sem cabeça-de-lista).
10. Os processos na fila das impressoras estarão em duas filas ao mesmo tempo: uma por ordem de ingresso na fila e outra pelo tempo de permanência no sistema. Processos que estão no sistema a mais do que 10 minutos têm prioridade para serem impressos. Estas filas devem ser implementadas de modo a permitir que inserções em ambas as filas e remoção do primeiro elemento de cada fila possam ser feitas em tempo $O(\log n)$. Para isto, implemente a primeira fila utilizando uma lista circular duplamente ligada com cabeça-de-lista, e para representar a outra fila, utilize uma fila de prioridade implementada num heap.
11. As filas da `Classe 1` e `Classe 2`, e a fila das impressoras devem ser implementadas utilizando alocação dinâmica. Mas, para a fila da CPU, implemente e utilize as rotinas de manipulação de uma lista livre.

6. Especificações da saída do programa

1. Sempre que terminar a execução de um processo, imprima as seguintes informações:
   a) características do processo (número de identificação, classe, tempo de CPU e número de linhas impressas);
   b) tempo total de permanência no sistema;
   c) tempo gasto em cada uma das filas;
   d) razão entre o tempo útil (tempo de execução na CPU, ou seja, *running time*) e tempo total de permanência no sistema (ou seja, *elapsed time*).
2. Ao término da simulação, imprima o número total de processos executados, assim como a média dos itens b), c) e d) para cada classe.
3. Além disso, imprima periodicamente (digamos, a cada 100 segundos—a escolha deste número fica a seu critério) o estado de cada uma das filas, inclusive as duas filas para as impressoras:

a) Fila das classes. Imprima para cada processo o seu número e o tempo que está na fila.
b) Fila da CPU. Imprima para cada processo o seu número, tempo de CPU, e tempo restante de execução.
c) Fila das impressoras por ordem de chegada. Imprima para cada processo o seu número, número de linhas que irá imprimir, e o tempo que esta na fila.
d) Fila das impressoras por tempo de permanência no sistema. Imprima para cada processo o seu número, número de linhas que irá imprimir, e o tempo de permanência no sistema.

4. Se você achar conveniente imprimir mais informações está ótimo. A saída do programa deve, principalmente, ajudá-lo a verificar se a simulação (ou seja, o seu programa) está correta.

Referências

1. A.S. Tanenbaum, *Modern operating systems*, Prentice-Hall, Englewood Cliffs, NJ, 1992.
2. J.P. Tremblay and P.G. Sorenson, *An introduction to data structures with applications*, Computer Science Series, McGraw-Hill, Singapore, 1984, QA758 T789i.