Lista de Exercícios de Sistemas Operacionais

01) Implemente um programa C que recebe como parâmetros na linha de comando 10 números inteiros (desordenados). O programa MAIN armazena os números em um array e cria um processo filho. Em seguida o MAIN deve ordenar o array usando "ordenação simples" enquanto o filho deve fazer "quick sort". Ao final da ordenação, cada processo deve exibir o tempo gasto para realizar a mesma. O Pai deve fazer um wait() para esperar o filho terminar.

Dicas:

```
#include <time.h>
...
clock_t c1, c2; /* variáveis que contam ciclos do processador */
float tmp;
c1 = clock();
//... código a ser executado
c2 = clock();
tmp = (c2 – c1)*1000/CLOCKS_PER_SEC;  // tempo de execução em milissegundos
-----------------------------------------------------------------
void quickSort(int valor[], int esquerda, int direita){

int i, j, x, y;

i = esquerda;
j = direita;
x = valor[(esquerda + direita) / 2];
while(i <= j){
        while(valor[i] < x && i < direita){
        i++;
        }
        while(valor[j] > x && j > esquerda){
        j--;
        }
        if(i <= j){
        y = valor[i];
        valor[i] = valor[j];
            valor[j] = y;
        i++;
        j--;
        }
}
if(j > esquerda){
```

```
            quickSort(valor, esquerda, j);
        }
        if(i < direita){
            quickSort(valor,  i, direita);
        }
}
```

02) Analise o trecho de código a seguir e determine quantos processos são criados. 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 'P'. O processo criado no ‘fork 1’ deve conter 'F1', o processo criado no ‘fork 2’ deve conter 'F2', e assim sucessivamente. Deve haver setas (apontando p/ baixo) saindo de cada processo pai em direção a cada processo filho. Assuma que nenhum erro ocorre ao realizar a chamada *fork()*.

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

03) Escreva um programa, em Java, que funcione como um shell simples. O programa deve ler um comando do usuário, por exemplo "cat file.txt", e depois criar um processo externo separado que executa este comando.

04) Escreva um programa Java que calcule o determinante de uma matriz A de ordem n >= 3. Lembre-se que:

i) o determinante de uma matriz *A* de ordem n=2 pode ser calculado como a diferença entre o produto dos termos da diagonal principal e o produto dos termos da diagonal secundária:

$$det(\mathrm{A}) = det\begin{bmatrix} a & b \\ c & d \end{bmatrix} = ad - bc$$

ii) o determinante de uma matriz de ordem n >= 3 pode ser calculado utilizando o Teorema de Laplace:

$$det(A) = \sum_{j=1}^{n} (-1)^{i+j} . a_{ij} . \det(A\_i\_j)$$

onde n é o número de linhas da matriz, i é a posição em relação às linhas (qualquer inteiro fixado entre 1 e n), j é a posição em relação às colunas e $det(A_{-i\,-j})$ é o determinante da submatriz que exclui a linha i e a coluna j.

Exemplo com uma matriz 4x4:

Seja a matriz com 4 linhas e 4 colunas

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}$$

Aplicando a fórmula mencionada acima, temos:

$$det(A) = \sum_{j=1}^{4} (-1)^{1+j} . a_{1j} . det(A_{-1-j})$$

Desenvolvendo o determinante pela primeira linha obtemos:

$$\det A = a_{11} . (-1)^{1+1} . \det\left(A_{-1,-1}\right) + a_{12} . (-1)^{1+2} . det\left(A_{-1,-2}\right) + a_{13} . (-1)^{1+3} . det\left(A_{-1,-3}\right) + a_{14} . (-1)^{1+4} . det\left(A_{-1,-4}\right),$$

onde $A_{-i,-j}$ representa a matriz obtida a partir de A, com a retirada da i-ésima linha e da j-ésima coluna.

$$\det A = a_{11} . (-1)^2 . \begin{vmatrix} a_{22} & a_{23} & a_{24} \\ a_{32} & a_{33} & a_{34} \\ a_{42} & a_{43} & a_{44} \end{vmatrix} + a_{12} . (-1)^3 . \begin{vmatrix} a_{21} & a_{23} & a_{24} \\ a_{31} & a_{33} & a_{34} \\ a_{41} & a_{43} & a_{44} \end{vmatrix} + a_{13} . (-1)^4 . \begin{vmatrix} a_{21} & a_{22} & a_{24} \\ a_{31} & a_{32} & a_{34} \\ a_{41} & a_{42} & a_{44} \end{vmatrix} + a_{14} . (-1)^5 . \begin{vmatrix} a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \\ a_{41} & a_{42} & a_{43} \end{vmatrix}$$

O valor do determinante de uma matriz A deve ser calculado por um thread independente do programa principal. A matriz A deve ser passada como parâmetro pelo construtor da classe deste thread. O valor do determinante deve ser calculado no corpo do método *run*. Assim, este thread (classe) pode tanto armazenar o valor calculado para o determinante de A em um objeto também passado como parâmetro no construtor ou disponibilizar um método para retornar este valor. Sempre que necessário, um thread que realiza o cálculo do determinante deve criar novos threads filhos para calcular o determinante das submatrizes. Neste caso, o thread pai deve esperar pelo término do cálculo do determinante de todas as submatrizes para calcular seu próprio determinante.

05) Um spooler de impressão é um processo responsável por gerenciar todos os jobs (tarefas) contendo requisições para a impressão de um arquivo. Este processo recebe as requisições de impressão encaminhas por diferentes processos, servindo-as uma por vez, isto é, imprimindo um arquivo de cada vez. Caso o spooler esteja servindo um job, um processo requisitante deve aguardar sua vez. Implemente uma classe Java (PrintSpooler.java) que simule um spooler de impressão utilizando um semáforo. Esta classe deve implementar os seguintes métodos:

```
public class PrintSpooler {
    // construtor classe - recebe como parâmetro o nome do arquivo de spool
    public PrintSpooler(String spoolFile);

    // método utilizado para abrir o arquivo de spool
    public boolean openPrintSpooler();

    // método utilizado para imprimir um job
    public void printJob(String jobName);

    // método utilizado para fechar o arquivo de spool
    public void closePrintSpooler();
```

Todas as solicitações recebidas (arquivos texto apenas), devem ser salvas ("impressas") em um mesmo arquivo de saída ("arquivo de *spool*") na ordem em que foram recebidas. A impressão dos jobs ocorre de forma simulada, mediante a cópia linha por linha do arquivo a ser impresso para o arquivo de *spool*. Certifique-se de que você possa distinguir o início de cada impressão no arquivo de saída, por exemplo, por meio da impressão de um cabeçalho.

Implemente também um thread (PrintJob.java) responsável pela submissão de uma requisição de impressão. Este thread deve esperar um período de tempo aleatório (até 5 segundos) antes de solicitar a impressão de um job.

Finalmente, escreva também um programa (Exe5.java) para testar o spooler. Este programa deverá receber via linha de comando uma lista de arquivos a serem impressos. Para cada arquivo a ser impresso, uma instância do thread PrintJob deve ser criado. Antes que os threads de impressão possam encaminhar suas solicitações para o spooler, o arquivo de *spool* deve ser aberto. Após a conclusão de todas as impressões (encerramento de todos os threads de impressão), o arquivo de *spool* deve ser fechado.