LISTA 1

ENRIQUE CAMPOS NOGUEIRA - 163701

DOCENTE: JOAHANNES BRUNO DIAS DA COSTA

UNIDADE CURRICULAR: SISTEMAS OPERACIONAIS

SÃO JOSÉ DOS CAMPOS

MAIO - 2025

Questão 1:

O sistema operacional é um software fundamental que funciona como um intermediário entre o hardware e o software do seu dispositivo. Ele ajuda a tornar tudo mais fácil e seguro, permitindo que você e os aplicativos interajam com o hardware sem precisar entender cada detalhe técnico que acontece por trás dos panos.

Entre as suas principais tarefas estão gerenciar processos (incluindo criação, execução e finalização), gerenciar a memória (alocando espaço quando necessário e liberando quando não precisa mais), controlar os dispositivos de entrada e saída, e garantir a segurança do sistema como um todo. Além disso, ele oferece interfaces mais simples para o usuário e garante que os recursos do sistema sejam usados de um jeito justo e eficiente.

Questão 2:

O “Zoológico dos Sistemas Operacionais” é uma maneira de ilustrar a diversidade enorme de sistemas por aí, cada um com suas características específicas, feitos para diferentes contextos de uso.

Por exemplo, há sistemas operacionais voltados para desktops e notebooks (como Windows, macOS e Linux), para dispositivos móveis (como Android e iOS), para servidores (como Windows Server e Ubuntu Server), e ainda para mainframes corporativos (como o z/OS da IBM).

Questão 3:

No contexto da disciplina, a mudança mais marcante aconteceu na quarta geração, com os microprocessadores de baixo custo e as interfaces gráficas começaram a popularizar o uso do computador pessoal. Foi justamente nessa fase que sistemas operacionais como Windows e Mac OS ficaram mais amigáveis para quem não era técnico.

- **Primeira geração (1945–1955):** Não havia sistemas operacionais propriamente ditos, os programas eram carregados por paineis e cabos, usando válvulas, e tinham rotinas básicas só para iniciar cada tarefa. O enfoque estava em viabilizar a execução de cálculos científicos e militares, mas a máquina era instável e consumia enorme energia.
- **Segunda geração (1955–1965):** O uso de transistores permitiu criar sistemas em lote (batch) que liam cartões perfurados e executavam tarefas de forma automática e sequencial. Surgiu o conceito de monitor residente, aumentando a confiabilidade e reduzindo o tempo ocioso da CPU, além de diminuir o consumo de energia em relação à geração anterior.
- **Terceira geração (1965–1980):** Com o IBM 370 e similares, nasceram os verdadeiros sistemas operacionais, gerenciamento de processos, memória, multiprogramação e tempo compartilhado. Usuários podiam compartilhar o mesmo computador simultaneamente, elevando produtividade e aproveitamento de hardware.
- **Quarta geração (1980–presente):** Os microprocessadores viabilizaram ter computadores pessoais em casa e no trabalho, rodando sistemas com interfaces gráficas, multitarefa e suporte ao mouse. Essa combinação de baixo custo, facilidade de uso e ecossistema de software transformou o computador em ferramenta cotidiana.
- **Quinta geração (1990–presente):** Mais recentemente, a convergência entre computação e telefonia fez surgir sistemas operacionais para dispositivos móveis, como iOS e Android, com gerenciamento inteligente de energia, conexão constante e aplicativos específicos, levando a computação além do espaço do escritório.

Questão 4:

Processos e threads: Um processo é um programa em execução, com sua própria memória, enquanto uma thread é uma parte menor de um processo que compartilha recursos. Usar threads permite que programas façam várias tarefas ao mesmo tempo, deixando tudo mais rápido e responsivo.

Escalonamento: Define a ordem que os processos serão executaddos pelo processador. Existem diferentes algoritmos de escalonamento, cada um com objetivos distintos, como minimizar o tempo de resposta, maximizar a utilização da CPU ou garantir justiça entre os processos. Isso ajuda o sistema a distribuir bem seus recursos e melhorar o desempenho geral.

Questão 5:

O uso de cache na hierarquia de memória é fundamental para acelerar o acesso aos dados. Ela armazena temporariamente as informações mais utilizadas em memórias muito rápidas, permitindo que o processador encontre o que precisa com mais facilidade. Isso evita acessos demorados à RAM ou ao disco, resultando em um desempenho geral muito melhor.

Por exemplo, nas CPUs modernas, há níveis de cache L1, L2 e L3: o processador tenta primeiro acessar o L1 (que é o mais rápido), depois o L2, e assim por diante, antes de buscar na memória principal. Essa estratégia melhora a capacidade e velocidade, otimizando o desempenho.

Questão 6

Núcleo Monolítico: Reúnem todas as funções principais do sistema (processos, memória, drivers, sistemas de arquivo e rede) num único espaço privilegiado, eliminando a necessidadde de comunicação externa e reduzindo latências. Isso melhora o desempenho, mas aumenta o risco de falhas completas, além de tornar a segurança mais duvidosa devido ao baixo isolamento entre os módulos.
Implementações comuns: Linux, Windows e Solaris..

Micronúcleo: Mantêm apenas o essencial no kernel (escalonamento, IPC básico e memória), executando drivers e serviços em espaço de usuário, o que melhora modularidade, segurança e tolerância a falhas, mas introduz overhead de IPC e trocas de contexto que podem reduzir desempenho.
Implementações comuns: QNX e Minix.

Questão 7:

A separação entre o modo usuário e o modo núcleo é fundamental para garantir a estabilidade e a segurança do sistema. Isso porque qualquer operação que exija privilégios elevados precisa passar por uma transição controlada para o kernel, o que impede que códigos mal-intencionados ou com falhas causem danos ao sistema como um todo.

No modo usuário, as aplicações possuem privilégios restritos e não têm acesso direto ao hardware ou à memória de outros processos, de modo que erros ou invasões afetem apenas o próprio programa.

Já no modo núcleo, o sistema operacional executa códigos com privilégios totais, tendo acesso irrestrito aos recursos do hardware. Essa divisão garante um gerenciamento seguro de processos, memória e dispositivos, protegendo o funcionamento geral do sistema.

- **Chamadas de sistema (syscalls):** para realizar E/S, acesso a arquivos ou criação de processos, o programa faz uma transição de usuário para núcleo de forma controlada.

  Drivers de dispositivo: muitos drivers operam em modo núcleo para acessar diretamente periféricos. Já em microkernels, rodam em espaço usuário mas elevam privilégio via IPC.
- **Gerenciamento de memória:** operações de paginação, mapeamento de endereços e proteção de espaço são feitas em modo núcleo, evitando violações de isolamento.

Questão 8:

Processos podem se comunicar entre si através de mecanismos de Comunicação entre Processos (IPC), como pipes, memória compartilhada e filas de mensagens. Esses mecanismos permitem que processos distintos troquem dados ou sinais de forma segura e coordenada, mesmo estando isolados em seus próprios espaços de memória. O IPC é fundamental para a sincronização e o compartilhamento de informações em aplicações que executam múltiplos processos de forma simultânea.

Um exemplo comum é o uso de pipe em sistemas Unix. Nesse cenário, o processo pai cria um pipe e depois gera um processo filho com fork(). O filho herda os descritores de leitura e escrita do pipe, permitindo que o pai escreva dados e o filho os leia, ou vice-versa. Essa comunicação é simples e eficiente, facilitando a troca de informações entre processos de maneira direta.

```c
#include <stdio.h>

#include <stdlib.h>
```

```c
#define MAX 1000


// Estrutura do nó da árvore AVL
typedef struct node {
    int data, height;
    struct node *lef, *rig, *dad;
} Tree;


// Calcula a altura de um nó
int Height(Tree *root){
    if (root == NULL)
        return 0;
    else
        return root->height;
}


// Atualiza a altura de um nó
void UpdateHeight(Tree *root) {
    int hl = Height(root->lef);
    int hr = Height(root->rig);
    if (hl > hr)
        root->height = hl + 1;
    else
        root->height = hr + 1;
```

```
}

// Rotação à esquerda
Tree* RotateL(Tree *root) {
    Tree *temp = root->rig;
    root->rig = temp->lef;
    if (temp->lef)
        temp->lef->dad = root;
    temp->lef = root;
    temp->dad = root->dad;
    root->dad = temp;
    if (temp->dad) {
        if (temp->dad->lef == root)
            temp->dad->lef = temp;
        else if (temp->dad->rig == root)
            temp->dad->rig = temp;
    }
    UpdateHeight(root);
    UpdateHeight(temp);
    return temp;
}

// Rotação à direita
Tree* RotateR(Tree *root) {
    Tree *temp = root->lef;
```

```
    root->lef = temp->rig;
    if (temp->rig)
        temp->rig->dad = root;
    temp->rig = root;
    temp->dad = root->dad;
    root->dad = temp;
    if (temp->dad) {
        if (temp->dad->lef == root)
            temp->dad->lef = temp;
        else if (temp->dad->rig == root)
            temp->dad->rig = temp;
    }
    UpdateHeight(root);
    UpdateHeight(temp);
    return temp;
}

// Balanceia a árvore aplicando rotações
Tree* Balance(Tree *root){
    if (root == NULL) return NULL;

    UpdateHeight(root);
    int balance = Height(root->lef) - Height(root->rig);

    if (balance > 1) {
```

```
            if (Height(root->lef->lef) >= Height(root->lef->rig)) {
                root = RotateR(root);
            }
            else {
                root->lef = RotateL(root->lef);
                root = RotateR(root);
            }
        }
        else if (balance < -1) {
            if (Height(root->rig->rig) >= Height(root->rig->lef)) {
                root = RotateL(root);
            }
            else {
                root->rig = RotateR(root->rig);
                root = RotateL(root);
            }
        }
        return root;
}


// Insere um valor x e balanceia
Tree* Insert(Tree *root, int x){
    if (root == NULL) {
        Tree *temp = (Tree*)malloc(sizeof(Tree));
        temp->data = x;
```

```
        temp->lef = temp->rig = NULL;
        temp->dad = NULL;
        temp->height = 1;
        return temp;
    }

    if (x <= root->data) {
        root->lef = Insert(root->lef, x);
        root->lef->dad = root;
    }
    else {
        root->rig = Insert(root->rig, x);
        root->rig->dad = root;
    }
    return Balance(root);
}

// Remove um valor x e balanceia
Tree* Remove(Tree *root, int x){
    if (x < root->data) {
        root->lef = Remove(root->lef, x);
        if (root->lef)
            root->lef->dad = root;
    }
    else if (x > root->data) {
```

```
        root->rig = Remove(root->rig, x);
        if (root->rig)
            root->rig->dad = root;
    }
    else {
        if (root->lef == NULL) {
            Tree *temp = root->rig;
            if (temp) temp->dad = root->dad;
            free(root);
            return temp;
        }
        else if (root->rig == NULL) {
            Tree *temp = root->lef;
            if (temp) temp->dad = root->dad;
            free(root);
            return temp;
        }

        Tree *temp = root->rig;
        while (temp->lef != NULL)
            temp = temp->lef;

        root->data = temp->data;
        root->rig = Remove(root->rig, temp->data);
        if (root->rig) root->rig->dad = root;
```

```
    }
    return Balance(root);
}

// Busca um valor x na árvore
Tree* Search(Tree *root, int x){
    if (root == NULL)
        return NULL;

    if (x < root->data)
        return Search(root->lef, x);
    if (x > root->data)
        return Search(root->rig, x);

    return root;
}

// Libera todos os nós da árvore
void FreeTree(Tree *root) {
    if (root == NULL) return;
    FreeTree(root->lef);
    FreeTree(root->rig);
    free(root);
}
```

```c
int main() {

    Tree *root = NULL;

    int i, j = 0, x, y, h, hl, hr, ha, hla, hra, a, b, v[MAX];

    for (i = 0; i < MAX; i++)

        v[i] = -1;



    // Inserção de valores na árvore

    while (scanf("%d", &x) == 1) {

        if (x < 0)

            break;

        root = Insert(root, x);

    }



    // Altura da árvore antes das mudanças

    int hb  = root->height;

    int hlb = root->lef ? root->lef->height : 0;

    int hrb = root->rig ? root->rig->height : 0;



    // Inserção e remoção de valores na árvore

    while (scanf("%d", &y) == 1) {

        if (y < 0)

            break;

        else if (Search(root, y) == NULL)

            root = Insert(root, y);
```

```
        else
            root = Remove(root, y);
    }

    // Altura da árvore após das mudanças
    if (root != NULL){
        ha  = root->height;
        hla = root->lef ? root->lef->height : 0;
        hra = root->rig ? root->rig->height : 0;
    }

    // Leitura do intervalo [a, b]
    scanf("%d %d", &a, &b);

    // Imprime as alturas da árvore antes e depois das mudanças
    printf("%d, %d, %d\n", hb-1, hlb, hrb);

    if (root == NULL) {
        printf("ARVORE VAZIA\n");
        return 0;
    } else {
        printf("%d, %d, %d\n", ha-1, hla, hra);
    }

    // Imprime os valores do intervalo presenter na árvore
```

```
    for (i = a; i <= b; i++) {
        if (Search(root, i) != NULL) {
            if (j == 0) printf("%d", i);
            else printf(", %d", i);
            v[j++] = i;
        }
    }

    // Imprime as alturas dos nós encontrados
    if (v[0] == -1) {
        printf("NADA A EXIBIR\n");
        FreeTree(root);
        return 0;
    }
    else {
        printf("\n");
        for (int k = 0; k < MAX && v[k] != -1; k++) {
            Tree *temp = Search(root, v[k]);
            if (temp != NULL) {
                h  = Height(temp);
                hl = Height(temp->lef);
                hr = Height(temp->rig);
                printf("%d, %d, %d\n", h-1, hl, hr);
            }
        }
```

```
    }


    FreeTree(root);
    return 0;
}
```