ESCOLA
SUPERIOR
DE TECNOLOGIA
E GESTÃO
POLITÉCNICO
DO PORTO

LICENCIATURA DE ENGENHARIA INFORMÁTICA / SEGURANÇA INFORMÁTICA
EM REDES DE COMPUTADORES
Sistemas Operativos

Trabalho Prático

João Rafael Da Cunha Guerra- 8200098
Miguel Ângelo Leão Barbosa – 8200102
Tiago Filipe Teixeira Costa - 8200124

2023/2024

ESCOLA
SUPERIOR
DE TECNOLOGIA
E GESTÃO

P.PORTO

Conteúdo

1.Introdução

1.1.Contextualização do problema

Este projeto tem como objetivo realizar a simulação de um sistema operacional para um satélite, o qual gerenciará duas unidades de processamento em tempo real: a Unidade Central de Processamento (CPU) e a Unidade de Memória (MEM). O desenvolvimento abrangerá a criação de componentes essenciais, incluindo o kernel do sistema operacional, responsável pelo processamento e tomada de decisões fundamentais para o funcionamento do sistema.

A unidade MEM terá a função de armazenar e manipular dados e informações relevantes para o sistema, enquanto a CPU desempenha o papel crucial na gestão, escalonamento e execução das tarefas. Além disso, será implementado o Middleware, um componente responsável pela comunicação entre as tarefas, utilizando protocolos de comunicação. Esse middleware fará uso dos recursos disponibilizados tanto pela MEM quanto pela CPU, garantindo uma interação eficiente e coordenada entre os diferentes elementos do sistema operacional do satélite.

1.2. Ferramentas Utilizadas

Para o desenvolvimento do trabalho foram utilizadas as seguintes ferramentas

Java

Este sistema foi desenvolvido com a linguagem de programação Java com recurso do IDE intellij.

Gradle

O Gradle é uma ferramenta de automação de construção e gerenciamento de dependências com suporte a multi-projetos, utilizando DSL baseada em Groovy ou Kotlin.

Swing

Swing é uma biblioteca gráfica em Java usada para criar interfaces gráficas de usuário (GUI) em aplicativos desktop.

1.3. Descrição do Sistema

O sistema operativo do satélite é composto por vários componentes fundamentais que trabalham em conjunto para assegurar o seu funcionamento eficiente. O kernel do sistema operativo desempenha um papel central, sendo responsável pelo processamento e pelas decisões essenciais para o sistema. Ele mantém as estruturas de dados, valida informações, controla as tarefas a serem executadas e gerencia o lançamento e término de subcomponentes.

A MEM (Unidade de Armazenamento e Manipulação de Dados) é a entidade encarregada do armazenamento e manipulação de dados e informações. Essa unidade fornece os recursos necessários para o funcionamento global do satélite, garantindo a eficiente gestão e acesso aos dados quando necessário.

A CPU (Unidade Central de Processamento) é responsável pela gestão, escalonamento e execução das tarefas do satélite. A sua função principal é garantir a eficiência das operações, otimizando o desempenho geral do sistema.

O Middleware atua como uma camada intermediária entre as diversas tarefas do satélite, facilitando a comunicação por meio de protocolos específicos. Executando "em cima" do kernel, o Middleware controla mensagens e objetos, estabelecendo um serviço de comunicação para o satélite. Utiliza recursos disponibilizados pela MEM e pela CPU para garantir uma interação suave entre as partes do sistema.

Para simplificar a troca de informações entre as tarefas escalonadas pela CPU, o sistema utiliza uma estrutura de dados com tamanho de cinco unidades. Essa estrutura é compartilhada, permitindo que as tarefas escrevam e leiam dados conforme necessário.

2.Compilação

Para gerir e construir o nosso projeto de forma eficiente, optámos por utilizar o Gradle como ferramenta de automação. O Gradle oferece-nos flexibilidade e potência na definição e execução de tarefas relacionadas com a compilação, teste e execução da nossa aplicação.

Dentro do contexto do nosso projeto, a tarefa crucial de executar a aplicação é realizada através do comando personalizado do Gradle. Ao utilizar o Gradle para correr a nossa aplicação, empregamos o seguinte comando:

> gradle run

Este comando está configurado no nosso ficheiro build.gradle para iniciar a execução da aplicação. Trata automaticamente da compilação, resolução de dependências e de qualquer outra configuração necessária para garantir que a aplicação é executada corretamente.

Ao adotar o Gradle e incorporar o comando gradle run no nosso fluxo de trabalho, facilitamos significativamente o processo de desenvolvimento, permitindo que a equipa execute a aplicação de forma rápida e eficiente, simplificando assim o ciclo de desenvolvimento e teste do nosso projeto

3.Descrição das funcionalidades implementadas

3.1. UserCRUD

A classe UserCRUD em Java encapsula operações de manipulação de dados de usuários, seguindo o padrão CRUD, e interage com um arquivo JSON para persistência de dados. Ela utiliza a biblioteca Jackson para manipulação de dados JSON, mantém registros de eventos por meio de logs, e oferece funcionalidades para criar, ler, atualizar e excluir usuários. A classe trata exceções relacionadas à leitura e gravação de dados, e inclui constantes para facilitar a manutenção do caminho do arquivo JSON. Esta classe serve como uma interface para manipulação de dados de usuários, podendo ser integrada a interfaces gráficas ou outras partes do sistema.

3.2. Logs

Esta implementação gera logs usando a classe LogEntry para registrar eventos importantes. Esses logs são adicionados ao ler mensagens da memória, enviar mensagens para o satélite, receber respostas do satélite e outros eventos relevantes. Os logs são úteis para monitorar o fluxo de execução do software, identificar ações significativas e diagnosticar potenciais problemas na comunicação entre os componentes do sistema.

3.3.Graph

O código Java cria uma aplicação gráfica com o Swing e JFreeChart para exibir um gráfico de barras das contagens diárias de login, lidas de um arquivo JSON. A classe LoginChart utiliza a biblioteca Jackson para processar o JSON e

JFreeChart para gerar o gráfico. O gráfico representa a evolução das contagens de login ao longo do tempo.

3.4. Interface Gráfica

Para a interface gráfica usamos a biblioteca gráfica Swing que facilita o desenvolvimento de interfaces gráficas de utilizador (GUI).

A escolha de utilizar uma interface gráfica não apenas transforma o software em uma ferramenta mais acessível, mas também enriquece a experiência do usuário, proporcionando uma interação mais intuitiva, visualmente agradável e eficiente com as funcionalidades do sistema.

3.5. Gestão de Threads

Inicialmente são criadas 10 Threads para serem utilizadas por tasks e 5 para a middleware, estas threads são inicializadas pela classe CPU onde ficam numa pool de Threads em espera de serem chamadas.

3.5.1. Implementação de Tasks

As tarefas são adicionadas através do método ‘addNewTask’ presente na classe Kernel, este método cria uma instância da classe Task, passando a mensagem e a referência para a middleware. A Task será então passada para a Cpu através do método ‘executeTask’, ao receber a task a Cpu aloca uma thread disponível à Task em questão.

Enquanto isso, a Middleware encontra-se em espera de receber uma permit de um semáforo.

Após receber a thread a task inicializa a sua execução e envia a mensagem para a middleware que por sua vez escreve a mensagem recebida na memória partilhada.

Após a mensagem ser escrita na memória é libertada uma permit que permite a execução da secção *run* da Middleware, que irá verificar continuamente se existem mensagens na memória para serem processadas.

Caso a memória esteja vazia a middleware entra em um estado de espera através do método wait libertando a thread para ser utilizada em outras tarefas.

Quando uma mensagem é escrita na memória a middleware é notificada e sai do estado em espera, procedendo com a leitura da mensagem e enviando-a para o satélite.

A mensagem será então processada pelo satélite para fornecer a resposta apropriada à mensagem recebida, após a resposta ser definida o satélite enviará a mensagem de volta para o middleware.

Por fim é feita uma libertação de recursos associados à tarefa.

4. Descrição e justificação da utilização de mecanismos de sincronização e comunicação entre módulos

4.1. Synchronized

```
1  public void sendMessage(Message message) {
2      mem.writeToMemory(message);
3
4      notify();
5  }
6
7  public Message receiveMessage() throws InterruptedException {
8      while (mem.isEmpty()) {
9
10         wait();
11     }
12     return mem.readFromMemory();
13 }
```

Estes métodos estão a interagir com a classe 'Mem' para trocar mensagens, o que pode levar a *race conditions*, dado que várias threads podem estar a aceder a 'Mem' simultaneamente. A ausência do bloco 'synchronized' aumenta o risco dessas situações, uma vez que múltiplas threads podem tentar escrever ou ler na 'Mem' simultaneamente. A inclusão do bloco 'synchronized' é imperativa nesta situação, pois assegura que apenas uma thread por vez pode acessar estes métodos, prevenindo possíveis conflitos e garantindo a consistência das operações.

4.2.Semáforo

```
1   private Semaphore startSemaphore = new Semaphore(1);
```

Aqui, é declarado um semáforo com o nome ‘startSemaphore’ e com uma permissão inicial de 1. O valor 1 significa que apenas uma thread pode adquirir o semáforo simultaneamente. Este semáforo é utilizado para controlar o início da execução da thread Middleware.

```
1   public void startMiddlewareThread() {
2           startSemaphore.release();
3       }
```

Este método é chamado para libertar uma permissão no semáforo, permitindo assim que a thread Middleware comece a executar. É uma forma de sincronização para garantir que a thread não comece a executar até que seja explicitamente permitido.

```
try {
    startSemaphore.acquire(); // Acquire a permit, or block until one is available
} catch (InterruptedException e) {
    // Handle the interruption or any other exception
    e.printStackTrace();
    Thread.currentThread().interrupt();
}
```

Dentro do método 'run', a thread tenta adquirir uma permissão do semáforo. Se o semáforo já estiver com uma permissão disponível, a thread adquire a permissão e continua a execução. Caso contrário, ela bloqueia até que uma permissão esteja disponível. Isso é feito para garantir que a execução da thread Middleware não comece até que a permissão seja explicitamente permitido, o que ocorre quando o método 'startMiddlewareThread' é chamado.

```
startSemaphore.release();
```

No final do loop da thread 'Middleware', uma permissão é libertada no semáforo. Isto permite que outras instâncias da thread Middleware possam adquirir o semáforo e começar a executar.

5.Conclusão

Ao explorarmos e implementarmos conceitos como threads, não só aprofundamos o nosso entendimento teórico sobre programação concorrente, como também adquirimos uma valiosa experiência prática. O desafio de gerir a concorrência, sincronização e comunicação entre threads trouxe com ele uma compreensão mais rica dos pormenores complexos envolvidos na construção de sistemas eficientes e escaláveis.

5.1 Limitações

No entanto, é importante destacar que, devido a determinadas limitações, não foi possível implementar um gráfico que mostrasse a quantidade de threads em tempo real durante a execução do programa. A complexidade técnica envolvida na sincronização e na comunicação em tempo real entre as threads, juntamente com as restrições de tempo do projeto, impediram a implementação bem-sucedida dessa característica.

Uma outra limitação notável envolveu a impossibilidade de implementar uma única pool de threads para otimizar o middleware e a classe de tarefas devido a desafios técnicos e práticos. Como alternativa, adotamos duas pools separadas para cada componente. Essa adaptação destaca a importância da flexibilidade no desenvolvimento, apontando para possíveis melhorias futuras na gestão de recursos e desempenho do sistema.