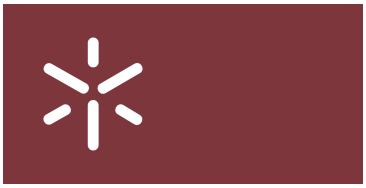

Universidade do Minho
Escola de Engenharia
Licenciatura em Engenharia Informática

Unidade Curricular de Sistemas Operativos

Ano Letivo de 2023/2024

Orquestrador de Tarefas

Gustavo Barros
A100656

Luís Figueiredo
A100549

Miguel Gramoso
A100835

May 07, 2024

SO

Índice

1. Introdução

Este relatório tem como intuito abordar o projeto da UC Sistemas Operativos do ano letivo 2023/2024.

Este relatório irá abranger as decisões tomadas pelo grupo tal como o método de raciocínio para desenvolvimento do mesmo. O objetivo do projeto é implementar um **serviço de monitorização** dos programas executados numa máquina.

2. Funcionalidades

De acordo com o enunciado proposto, desenvolvemos as seguintes funcionalidades:

- Receção de programas pedidos pelo utilizador (Client)
- Escalonamento das tarefas no Servidor (Orchestrator)
- Execução das tarefas
- Output dos comandos redirecionado para ficheiros num output_folder
- Execução encadeada de programas (pipelines)
- Consulta de programas em execução e a ser executados
- Armazenamento e consulta de informação de programas terminados (para o status)

3. Arquitetura

O objetivo da aplicação é seguir o modelo cliente/servidor onde temos um ou mais clientes que enviam pedidos ao servidor para serem processados.

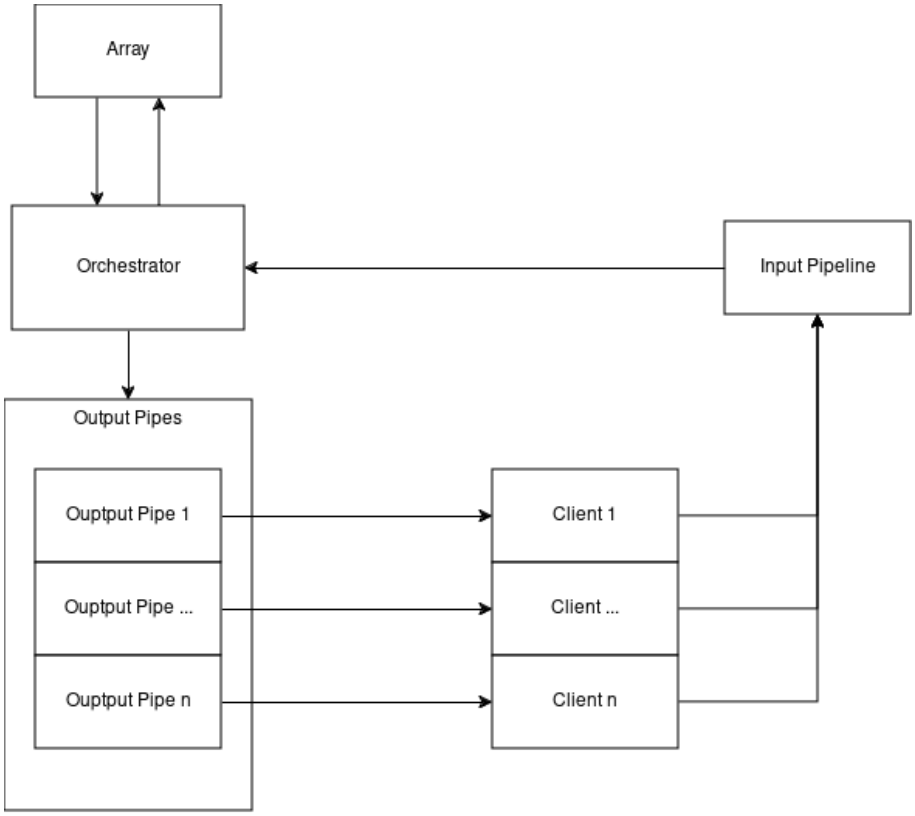

Figura 1: Esquema de funcionamento do programa

3.1. Orchestrator

O *Orchestrator* desenvolve o papel de servidor no nosso programa. É responsável por definir o método de escalonamento das tarefas e por aguardar a chegada de qualquer request. Atualmente, o servidor suporta dois métodos de escalonamento, nomeadamente o *Small Job First* e o *First Come First Served*.

Após a receção de um pedido *execute* por um *named pipe* chamado *input_pipe*, o servidor dá-lhe um id único e acrescenta-o aos pedidos a executar de acordo com a política de escalonamento e devolve ao cliente o id do pedido através de outro *named pipe* chamado *output_pipe* . Depois o servidor passa o pedido desse array para outro (que funciona como uma *queue*) de pedidos a serem executados, idealmente isto seria feito por outro processo (daí esta implementação) mas não conseguimos implementar. Posteriormente o servidor chama uma função encarregue de executar o pedido, dependendo se este é simples ou um *pipe*. Estas funções, basicamente, fazem um *fork* onde se cria o ficheiro de output do pedido (na pasta dada como argumento no início do servidor) e onde se executa o comando (redirecionando o output e o error para o ficheiro). O pai espera pelo filho, calcula o tempo demorado até ali e guarda essas informações num ficheiro (tarefas.txt) e retira o pedido da *queue* dos em execução.

Os pedidos *status* devolvem ao utilizador uma lista contendo informação sobre todos os pedidos já realizados (lido do tarefas.txt), os pedidos a serem executados e os pedidos que se encontram neste momento em execução. Isto também é feito num *fork* com a ideia do *status* ser independente do resto das funções do servidor mas não conseguimos implementar, então fazemos com que o pai (server) espere pelo filho (o que executa o status).

Posteriormente a uma execução de um pedido *status*, o servidor envia o resultado do pedido para o cliente através de um *named_pipe* chamado *status_output*.

3.2. Client

O *client* é responsável por receber a indicação de pedido via terminal e pela sua criação e comunicação. Para tal, usamos os três *pipes* mencionados anteriormente, um de entrada (para o servidor), um de saída (do servidor para o id) e um para ler o resultado do *status* do programa. Quando um pedido é introduzido pelo *client*, este é passado ao *orchestrator* através do *input_pipe* e a sua resposta (id) é devolvida através do *output_pipe*.

Levando isto em conta, o cliente consegue realizar diferentes tipos de pedidos. Estes podem ser um *execute* de um único comando ou então um conjunto de comandos encadeados, simulando a funcionalidade de *piping* no *Bash*. Estes dois pedidos são diferenciados com a *flag* do pedido, respetivamente as *flags -u* para comando únicos e a *flag -p* para *pipelines*.

Por fim, existe também o comando *status*, que devolve o estados dos programas realizados, em execução e a ser executados no futuro. De maneira a não interferir com os programas em execução, optamos por criar um *pipe* exclusivo para este tipo de pedido, permitindo que a aplicação não tenha conflitos.

4. Implementação

Após múltiplas abordagens ao enunciado atribuído, optamos por prosseguir com a estrutura que iremos apresentar. Este método permite aplicar múltiplos conceitos estudados em outras cadeiras, mas sobretudo da unidade curricular de Sistemas Operativos.

4.1. Pedidos

De maneira a facilitar a comunicação entre servidor e cliente utilizamos a seguinte estrutura:

```c
typedef struct pedido {
    int id; // gerado automaticamente pelo servidor
    int comando; //STATUS ou EXECUTE
    int tempo_execucao;
    char flag[3]; // -u ou -p
    char argumentos[MAX_ARG_SIZE]; // comandos a executar
    int ESTADO; //SCHEDULED, EXECUTING ou COMPLETED


} PEDIDO;
```

Listing 1: Estrutura Pedido

Com esta estrutura, conseguimos guardar as informações mais pertinentes de cada pedido, e conseguimos mais facilmente verificar o estado e tipo de pedido.

4.2. Gestor de pedidos

Quando um pedido é enviado ao servidor confirmamos o tipo do mesmo. Caso este seja um execute o servidor necessita guardar os dados dos mesmos em algum lado. Para isso criamos dois *arrays*, um para pedidos *scheduled* e outro para *executing*. Para isso utilizamos a seguinte estrutura:

```c
typedef struct array_pedidos {
    int max_pedidos;
    int num_pedidos;
    PEDIDO **array;
} ARRAY_PEDIDOS;
```

Listing 2: Estrutura array de pedidos

Para usar os *arrays* usa-se funçoes definidas no ficheiro utils.c que têm em conta a politica de escalonamento.

5. Conclusão

Com a finalização deste trabalho, apesar de não termos conseguido aclcançar todos os objetivos propostos, acreditamos que alcançamos os objetivos principais. Para além da implementação dos objetivos não atingidos, achamos que existe espaço para melhoria, desde implementação de modularidade à implementação de conceitos de *clean code*.

Acreditamos que o desenvolvimento deste projeto ajudou na consolidação do nosso conhecimento de C bem como os conceitos abordados na UC de Sistemas Operativos. Agora estamos mais familiarizados com estes conhecimentos e acreditamos que estes serão muito úteis no desenvolvimento de projetos futuros nesta área.