

**Unidade Curricular de**

**Sistemas Operativos**

Ano Letivo de 2023/2024

**Grupo 30**

Diogo Miranda (a100839)

João Rodrigues (a100598)

Sandra Cerqueira (a100681)

**Março, 2024**

**SO**

Índice

[Introdução 3](#_Toc193015579)

[Estrutura 3](#_Toc193015580)

[Cliente 4](#_Toc193015581)

[Orchestrator 5](#_Toc193015584)

[Configuração 5](#_Toc193015585)

[Criação do Server\_fifo 5](#_Toc193015586)

[Parser 5](#_Toc193015587)

[Gestão de novos pedidos e atendimento e execução dos mesmos 5](#_Toc193015588)

[Fila de espera 5](#_Toc193015589)

[Política de escalonamento 6](#_Toc193015590)

[Execução de um comando 6](#_Toc193015591)

[Gestão de remoção 7](#_Toc193015592)

[Conclusão 7](#_Toc193015593)

# Introdução

Este relatório descreve a implementação de um serviço de orquestração de tarefas num computador. O objetivo principal é permitir que os utilizadores submetam tarefas para execução num servidor, sendo que o servidor é responsável pelo escalonamento e execução dessas tarefas. Cada tarefa é associada a um identificador único, que é transmitido ao cliente assim que o servidor recebe a tarefa. Além disso, o servidor redireciona as informações produzidas pelas tarefas (saída padrão ou erro padrão) para ficheiros cujos nomes correspondem aos identificadores das tarefas.

Ao longo deste documento, detalharemos a arquitetura do sistema, os componentes envolvidos e a estratégia de escalonamento adotada. Também discutiremos como os utilizadores podem interagir com o servidor para consultar o estado das tarefas em execução, em espera ou já finalizadas.

Vamos agora explorar os detalhes da implementação e os desafios enfrentados durante o processo.

# Estrutura

Este serviço de orquestração de tarefas foi implementado com uma arquitetura cliente-servidor, na qual a comunicação entre ambos é realizada através de pipes com nome (FIFOs). Optámos pela utilização de FIFOs devido à sua capacidade de proporcionar uma comunicação eficiente e assíncrona entre os componentes do sistema. Os FIFOs permitem que os clientes enviem pedidos ao servidor de forma assíncrona, sem a necessidade de esperar por uma resposta imediata, o que é especialmente vantajoso em ambientes distribuídos.

No início da execução do servidor (*orchestrator*), é criado um dos pipes denominado “Server\_fifo”, que é responsável por receber os pedidos dos clientes para posterior processamento pelo servidor. Quando um cliente envia um pedido para o servidor (escrevendo no “Server\_fifo”), o cliente cria outro pipe, com o nome “Client\_fifo\_pid”. Desta forma, os nomes dos pipes são únicos para cada cliente. O pipe do cliente serve como meio de comunicação para a informação proveniente do servidor, que posteriormente é apresentada ao cliente.

Quando o servidor recebe um pedido, ele coloca-o numa fila de espera, onde são armazenados todos os pedidos feitos pelos clientes. Esses pedidos são atendidos de acordo com a capacidade definida para o servidor. Caso o comando ./orchestrator output\_folder parallel-tasks sched-policy não seja utilizado, são definidas quatro tarefas paralelas. O servidor utiliza um algoritmo de escalonamento para priorizar as tarefas com menor tempo de execução, garantindo que sejam executadas primeiro.

# Cliente

A nossa implementação do programa ***“Client.c”*** que representa o nosso cliente, começa com a validação dos argumentos recebidos pelo programa. Caso o programa não receba argumentos ou argumentos inválidos, é apresentada no STDOUT a lista dos comandos válidos de forma a dar feedback ao cliente do que ele pode usar no programa.

O cliente tem dois comandos principais que podem ser utilizados:

## Status: Este comando tem a função de revelar ao cliente o estado do servidor, listando as tarefas em execução, as tarefas em espera para executar, bem como as tarefas terminadas. Para as tarefas em execução ou à espera para executar lista os respetivos identificadores e programas. Por sua vez, para as tarefas terminadas lista os seus identificadores, programas, e tempo de execução.

Quando o comando “status” é executado, o programa cria o seu FIFO, cujo nome tem como base o PID do processo, tal como referido anteriormente. Este FIFO será usado para receber a resposta do servidor. O comando, juntamente com o PID do cliente, é então enviado para o servidor através do FIFO do servidor. O programa então lê a resposta do servidor do FIFO do cliente e imprime-a.

## Execute: O comando execute permite ao cliente indicar o tempo estimado para a execução de uma tarefa e qual a tarefa a executar. Para isso, o utilizador fornece ao programa cliente os seguintes argumentos de linha de comando:

* time: Tempo estimado em milissegundos para a execução da tarefa;
* prog-a: O nome ou caminho do programa a executar (ex: cat, grep, wc);
* [args]: Argumentos do programa, se existirem (ex: arg-1 (…) arg-n).

O número de argumentos é variável e o tamanho total dos argumentos não deve exceder os 300 bytes. Para executar uma tarefa, o utilizador usa a opção *./cliente execute time -u "prog-a [args]"*.

Quando o comando “execute” é executado, o programa primeiro verifica se o número de argumentos é suficiente e se o tempo fornecido é maior que zero. Se alguma dessas condições não for cumprida, o programa exibe uma mensagem de erro e termina.

Em seguida, o programa cria o seu FIFO, cujo nome inclui o PID do processo, tal como no status. Este FIFO é usado para receber a resposta do servidor.

Se o próximo argumento for “-u”, o programa entra num ciclo onde calcula o comprimento total dos argumentos fornecidos. Para garantir que não são ultrapassados os 300 bytes, em caso de ultrapassar, o programa exibe uma mensagem de erro e termina.

A partir daqui, o programa irá construir a string com o comando para enviar ao servidor, incluindo o PID do cliente, o comando “execute”, o tempo estimado, o nome do programa e os seus argumentos. Esta string do comando é então escrita no FIFO do servidor.

Finalmente, o programa abre o FIFO do cliente em modo de leitura e lê a resposta do servidor. Se a resposta for recebida, o programa imprime o identificador do processo. Se não houver resposta, o programa imprime uma mensagem indicando que não recebeu resposta. Por fim, o programa fecha o FIFO do cliente e termina.

# Orchestrator

## Configuração

No arranque do ***orchestrator.c,*** que corresponde ao nosso servidor***,*** pode ser corrido o seguinte comando: *$ ./orchestrator output\_folder parallel-tasks*

Através dele são definidos a pasta onde serão guardados os ficheiros de output *(output\_folder)* e o número máximo de tarefas que podem ser executadas em paralelo *(parallel-tasks).* Caso apenas seja executado o comando: *$ ./orchestrator*

É definida como *default* a pasta output para guardar os ficheiros de output resultantes da execução dos comandos. E o número máximo de tarefas em paralelo é colocado a 4.

## Criação do Server\_fifo

No início do nosso programa “orchestrator.c”, é criado o “Server\_fifo”. Este FIFO é usado para receber os pedidos dos clientes. O Server\_fifo é criado através da função mkfifo() e é aberto em modo de leitura e escrita.

## Parser

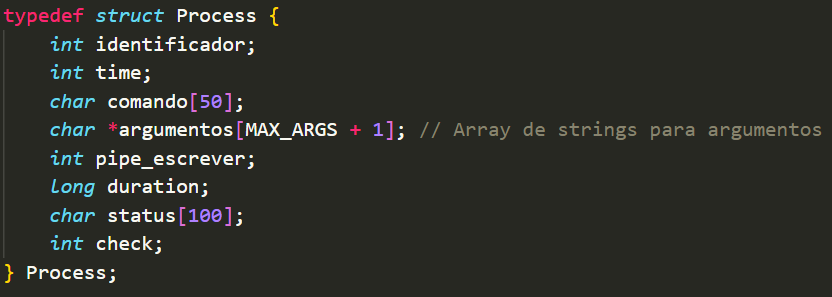
A função *‘parseArguments()’* é usada para analisar os argumentos de um comando. Ela divide a *string* de argumentos em palavras individuais e armazena-as num *array* de *strings*. As palavras são separadas por espaços, a menos que estejam entre aspas duplas.

## Gestão de novos pedidos e atendimento e execução dos mesmos

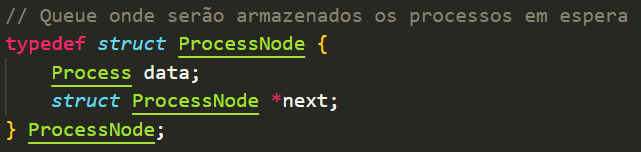
O nosso servidor, quando aberto, está constantemente à espera de novas mensagens no Server\_fifo. Quando recebe uma nova mensagem o servidor vai começar por identificar o comando pretendido. Caso o pedido, seja um ‘execute’, o nosso servidor vai verificar se o número de filhos, “numChildren”, ultrapassa ao valor máximo de filhos *(maxFilhos)* que o server pode criar, se essa condição se verificar o processo vai ser colocado na nossa fila de espera, caso contrário vai ser logo executado.

## Fila de espera

A nossa fila de espera, consiste numa lista ligada, em que, cada nodo da nossa lista corresponde a um processo a ser executado. Conforme vão chegando mais processos para serem executados, eles vão sendo armazenados na nossa lista, para tal definimos uma *struct* para guardar as informações relativas a cada processo, sendo ela a *struct* *‘Process’.*



Tendo essa estrutura definida, criamos a *struct ‘ProcessNode’.*



Esta estrutura, tal como o nome indica, vai corresponder a cada nodo da nossa lista, ela utiliza a estrutura ‘Process’, para guardar as informações do processo em que nos encontramos.

O apontador *‘next’* aponta para o próximo nó da lista, desta forma, conseguimos ter uma gestão flexível da nossa fila de espera, conseguindo manipulá-la como for necessário. Para conseguirmos adicionar novos processos à nossa lista utilizamos a função ‘insertOrderd()’.

## Política de escalonamento

A nossa função segue a política de ‘SJF’, ‘Shortest Job First’, isto é, ela vai ser ordenada consoante o tempo passado como argumento pelo utilizador aquando da execução do *‘client.c’*. Para tal, utilizamos a função ‘insertOrdered’, que vai ser responsável por inserir novos nodos na nossa lista, tendo por base a política de escalonamento.

## Execução de um comando

A função executeProcess é responsável por iniciar a execução de um processo. Ela recebe como entrada uma estrutura Process, que contém todas as informações necessárias para a execução do processo, como identificador, comando, argumentos, entre outros.

Primeiramente, a função regista o tempo de início da execução do processo utilizando a função gettimeofday(), para ser usada para calcular a duração. Em seguida, define o estado do processo como "Executando" e é colocado no array que contém todos os processos em execução.

Para cada processo é criado um processo filho que funcionará como um controlador, que por sua vez irá criar um processo neto que será o responsável pela execução do comando. O processo neto executa o comando, redireciona o seu output para um ficheiro e calcula a duração da execução para ser também escrita no ficheiro. O processo filho faz um wait pelo processo neto e quando deteta que este terminou irá escrever uma mensagem no fifo do servidor indicando que a execução terminou para este ter conhecimento.

## Gestão de remoção

Quando um processo termina a sua execução, seja porque concluiu a tarefa o servidor remove o processo da lista de processos em execução. As informações do processo são então adicionadas à lista de processos concluídos.

# Conclusão

O nosso serviço tem todas as funcionalidades básicas solicitadas no enunciado.

Gostaríamos de ter tido a oportunidade de testar mais aprofundadamente a nossa solução, com a implementação de testes e de outra politica de escalonamento para comparar com a implementada, e de ter implementado o restante das funcionalidades avançadas, mas devido à gestão de tempo isso não foi possível.

No entanto, acreditamos que tudo o que implementamos está em conformidade com o pedido e demonstra as nossas capacidades de aplicar os conhecimentos adquiridos ao longo da unidade curricular.