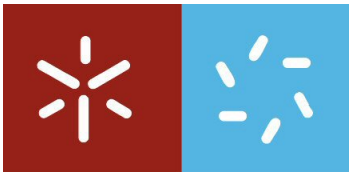

Universidade do Minho

Licenciatura em Ciências da Computação

Sistemas Operativos - Trabalho Prático

Grupo 12

Relatório de Desenvolvimento

Catarina Quintas	Pedro Martins	Tomás Campinho
(A91650)	(A91681)	(A91668)

7 de maio de 2024

Conteúdo

Capítulo 1

Introdução

Foi-nos proposto, no âmbito da unidade curricular de Sistemas Operativos, o desenvolvimento um serviço de orquestração de tarefas num computador. Este serviço consta de um cliente, designado por "Client", que oferece uma interface com o utilizador via linha de comandos e de um servidor, designado por "Orchestrator"responsável por escalonar e executar as tarefas. A comunicação entre estes dois processos é feita via pipes com nomes.

Neste relatório, vamos explicitar a nossa abordagem ao problema, justificando a estrutura do nosso serviço implementado e demonstrar os conhecimentos adquiridos durante as aulas, tais como a utilização/criação de processos, duplicação de descritores, criação de *pipes* com nome, execução de processos e *system calls*.

Capítulo 2

Funcionalidades Básicas

2.1 Execução de programas do utilizador

O cliente é responsável por solicitar a execução de programas dos utilizadores, como "cat", "grep", "wc", entre outros, através da opção *execute*.

Ele comunica ao servidor os detalhes da execução. O servidor, por sua vez, é responsável por receber os pedidos de execução do cliente e executar os programas solicitados. Ele armazena informações sobre o estado da execução e disponibiliza essas informações para consulta posterior.

2.1.1 Antes da execução do programa

O utilizador passa ao cliente os seguintes argumentos:

- A opção *execute -u*;
- O tempo estimado pelo utilizador para executar uma tarefa
- O nome do programa a executar;
- Os argumentos do programa, caso existam.

Sintaxe

```
./client execute 100 -u "prog_a arg_1 (...) arg_n"
```

```
./orchestrator output_folder parallel_tasks
```

2.1.2 Aquando a execução do programa

Estrutura de implementação

A comunicação entre o *Cliente* e o *Servidor* é feita via *pipes com nomes*. Começamos então por criar um pipe no programa *orchestrator* usando a função **mkfifo()** para onde o *Cliente* vai escrever a informação pedida e de onde o servidor vai ler essa mesma informação.

Para fazer a execução do programa, criamos um processo filho para cada tarefa a ser executada através da chamada de sistema **fork()**. Também criamos um array nomeado como *array executing*. Este array é uma estrutura importante dentro do servidor, pois é responsável por acompanhar as tarefas atualmente em execução. Após a sua conclusão, o resultado é registado, incluindo o tempo de execução da tarefa, e armazenado em um ficheiro.

O nome do programa a executar está na posicão argv[4], este nosso argumento é passado em forma de string e contém não só o nome do programa como também os seus argumentos. Deste modo decidimos dividir a string em tokens usando a funcão **strsep()** e armazenamos os tokens resultantes no array *exec_args*. Para acedermos ao nome do programa basta então fazermos *exec_args[0]*.

Através da função **gettimeofday(&start, NULL)** conseguimos obter o tempo imediatamente antes do início da execução do programa.
A variável *duracao* armazena o tempo total decorrido desde o momento em que a tarefa foi adicionada até sua conclusão. Optamos por esta contabilização do tempo, porque o cliente deve esperar o término da tarefa. Em outras palavras, *duracao* regista o intervalo de tempo total em que o cliente está aguardando a execução da tarefa.

Depois de toda a informação recolhida, esta será armazenada na *struct ClientData* e de seguida será escrita, pelo *Cliente*, no *FIFO* correspondente e lida pelo *Servidor*.

Prosseguindo agora à execução do programa, visto que guardamos o nome do programa e os seus argumentos num array, usamos a chamada ao sistema **execvp(exec_args[0], exec_args)**. Como é o processo filho que executa o programa, o processo pai espera por ele fazendo uma chamada de sistema **wait()**.

2.1.3 Após a terminação de um programa

Estrutura de implementação

Quando o comando *end* é recebido, o servidor irá encerrar. Ele aguardará até que todas as tarefas que estão na fila agendadas sejam executadas, saiam da fila e que não haja mais nenhuma tarefa pendente. Após o término das operações, os pipes de leitura e escrita do servidor serão fechados, encerrando assim o servidor juntamente com os outros dois fifos criados para o status.

Sintaxe

```
./client end
```

O servidor coordena as tarefas recebidas, as aloca para execução, mantém um controlo das tarefas em andamento e gere uma fila de espera para aquelas que não podem ser imediatamente executadas devido à capacidade limitada do servidor.

Se o comando *"status"* for recebido, o servidor comunica com os clientes através de pipes *FIFO*, enviando mensagens sobre o estado das tarefas e respondendo às solicitações dos clientes.

2.2 Consulta de programas em execução

2.2.1 Status

O comando *status* permite ao cliente obter uma visão atualizada do status das tarefas em execução no servidor.

As tarefas são exibidas de forma organizada, divididas em três categorias principais:

- *Executing* (Em execução)
- *Scheduled* (Agendadas)
- *Completed* (Concluídas)

Cada categoria é acompanhada das tarefas correspondentes, fornecendo informações como o ID da tarefa, os programas associados. Quando a categoria é *"Completed"* também nos fornece o tempo de execução.

Sintaxe

```
./client status
```

Estrutura de implementação

A resposta à opção *status* deve ser realizada pelo *Servidor*, portanto foi necessário fazer um *fork()* e criar 2 novos *FIFOS*, chamados *fila* e *execs*.

Inicia abrindo o pipe *FIFO* do servidor para escrita e enviando o comando *status*. De seguida, abre os 2 FIFOS criados para leitura e exibe os processos *"Scheduled"* e *"Executing"* do lado do *Client*, respetivamente.

Quando as tarefas estão terminadas, lê e exibe as mesmas partir do arquivo *"Tarefas terminadas.txt"*, proporcionando uma visão completa do histórico de execução.

Finaliza fechando os pipes FIFO e o arquivo "Tarefas terminadas.txt"após a conclusão das operações.

Capítulo 3

Funcionalidades Avançadas

3.1 Execução encadeada de programas

O sistema permite a execução encadeada de programas usando o comando -p, função **mysystem_P** que utiliza pipelines para encadear a saída de um programa como entrada para outros.

Na execução de pipelines, existem 3 casos:

- Primeiro Programa (*Índice 0*): Neste caso, é criado um pipe para a saída do programa. Um processo filho é criado para executar o programa e seu STDOUT é redirecionado para o pipe. O processo pai fecha o extremo de leitura do pipe.

- Programas Íntermédios: O STDIN do programa é redirecionado para a saída do programa anterior. O STDOUT é redirecionado para a entrada do próximo programa. Um processo filho é criado para executar o programa. O processo pai fecha os extremos de leitura e escrita do pipe do programa anterior e do próximo programa.

- Último programa (*Índice i-1*): o STDIN do programa é redirecionado para a saída do programa anterior. O STDOUT do último programa é redirecionado para um arquivo de saída especificado (tarefa). Um processo filho é criado para executar o programa. O processo pai fecha o extremo de escrita do pipe do programa anterior.

Sintaxe

```
./client execute 3000 -p "prog-a arg-1 (...) arg-n |
prog_b arg_1 (...) arg_n |prog_c arg_1 (...) arg_n"
```

3.2 Processamento de várias tarefas em paralelo

Ao receber comandos de clientes, o servidor cria processos filhos para executar as tarefas, garantindo que múltiplas operações possam ocorrer simultaneamente.

A função **comando_execute** é responsável por receber as informações das tarefas a serem executadas, como o tempo estimado para a conclusão e os comandos a serem executados. Ela cria uma estrutura de dados que representa a tarefa e a envia para o servidor através de um FIFO. Após a conclusão da tarefa, o servidor responde ao cliente através de outro FIFO.

O servidor, por sua vez, gere estas tarefas em uma fila de espera e em um array de tarefas em execução. Ele aloca processos filhos para executar essas tarefas usando as funções **mysystem_U** e **mysystem_P** dependendo do tipo de tarefa recebida. O resultado das tarefas é redirecionado para arquivos de saída específicos.

3.3 Avaliação de políticas de escalonamento

Para fazer a avaliação, escolhemos a política de escalonamento First-Come, First-Served (FCFS). Nesta política, os processos são escalonados na ordem que chegam, ou seja, o primeiro processo a chegar é o primeiro a ser executado. Implementamos uma fila de espera *queue* onde através da função **enqueue** é adicionado um novo processo no final da fila de espera, e da função **dequeue** é removido o próximo processo a ser executado, garantindo que a ordem de chegada seja respeitada.

Reconhecemos que a política de escalonamento escolhida não seja uma das politicas mais eficientes, pois não leva em consideração a prioridade dos processos ou sua importância, ou seja, os processos de curta duração podem ficar presos atrás de processos de longa duração, é que não é algo eficaz.

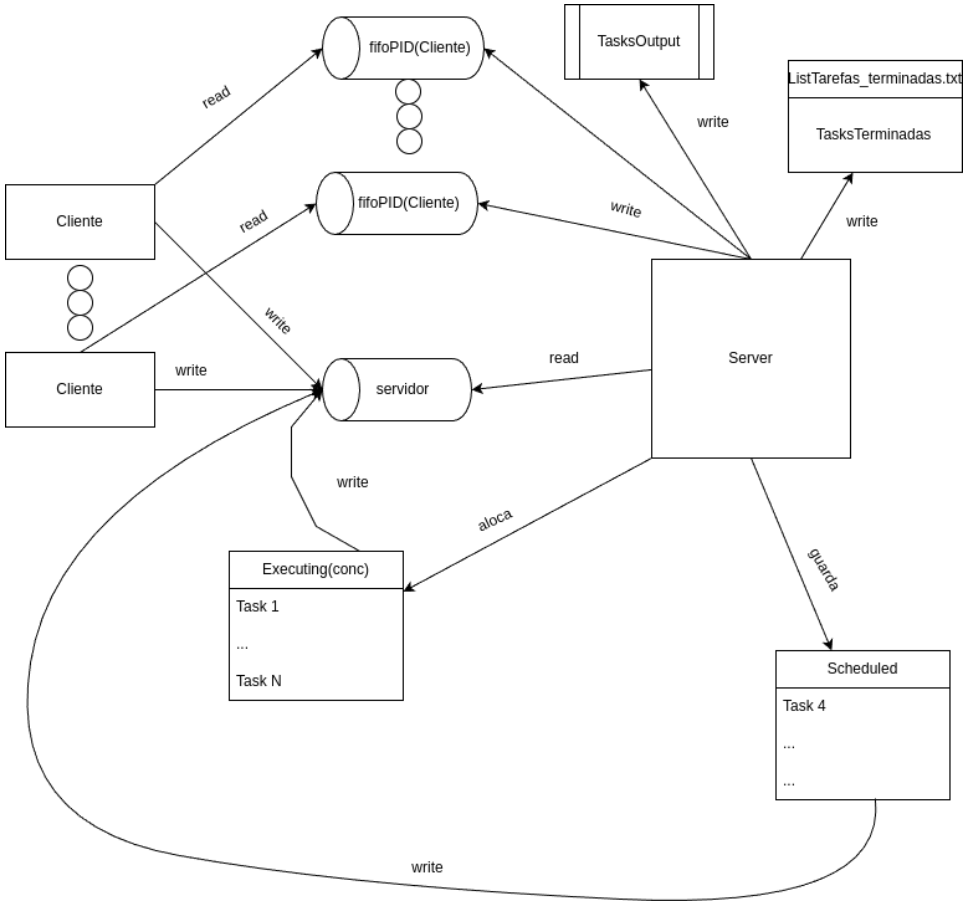

Figura 3.1: Diagrama geral do programa

Capítulo 4

Conclusão

Uma das peças fundamentais para a concretização do projeto foram as resoluções dos guiões práticos apresentadas pela equipa docente e até mesmo as nossas, pois serviram de apoio para o esclarecimento de dúvidas que surgiram.

Devido à dificuldade de entendermos alguns dos problemas que nos iam surgindo, não conseguimos concluir as funcionalidades avançadas, sendo este um aspeto a melhorar futuramente.

Este trabalho permitiu-nos consolidar toda a matéria prática e teórica lecionada na cadeira de Sistemas Operativos. Acreditamos que, apesar de não termos conseguido concluir todas as funcionalidades, como desejado, o nosso trabalho está bem concebido.