

UNIVERSIDADE DO MINHO

Licenciatura em Ciências da Computação

Sistemas Operativos - Trabalho Prático

Ano Letivo 2023/2024

Grupo 31

Pedro Miguel Ramôa Oliveira (a97686)

Diogo Coelho da Silva (a100092)

João Pedro da Silva Barbosa (100054)

Conteúdo

1. Introdução
2. Explicação da arquitetura do projeto
3. Cliente
4. Orchestrator
5. Conclusão

1. Introdução

Este relatório tem como objetivo explicar o nosso processo de pensamento para a realização deste projeto prático, assim como dar o nosso ponto de vista sobre o mesmo, e também, sobre os aspetos bons e menos bons do nosso trabalho.

Foi proposto a realização de um orquestrador de tarefas. Este orquestrador é responsável por receber pedidos de execução de tarefas por parte de clientes, e processar os mesmos e executar no servidor. A criação de um orquestrador deste tipo tem vários pontos cruciais a considerar.

Ao longo do relatório vão ser explicadas as decisões tomadas pelo grupo para o desenvolvimento deste orquestrador.

2. Explicação da arquitetura do projeto

Para a realização de um orquestrador de tarefas, é necessário ter um ou mais clientes, responsáveis pela criação de pedidos de instrução para um servidor, responsável pelo processamento dos mesmos e pela execução. Para isso, é necessária a criação de um *FIFO*, ou seja, uma forma de comunicar entre o servidor e os diversos clientes. Esta é uma parte importante deste trabalho, que é pelos *FIFO's* onde a informação relativa às diversas estruturas de dados vai passar. Mais a frente vamos explicar as decisões relativas à criação de *FIFO's*. Decidimos, enquanto grupo, que os nossos *FIFO's* apenas iriam funcionar de forma unidirecional. Nos nossos clientes apenas vamos utilizar o *FIFO* "*fifo_sv*", definido como *SERVER_NAME* para escrever para o nosso servidor. E no servidor apenas vamos utilizar o *FIFO* *CLIENT_NAME* para escrever para os nossos clientes. Este último *FIFO* tem uma particularidade que vai ser explicada no ponto 3. que fala sobre as nossas escolhas relativas ao cliente.

Foram criadas duas estruturas para o nosso projeto. A estrutura *Instruction*, que como o nosso indica guardará todas as informações relativas a uma instrução enviada pelo cliente:

```
typedef struct Instruction{
    commandType type;           //EXECUTE OR STATUS
    pid_t pid;                  //Identificador do cliente
    char args[INSTRUCTION_SIZE_MAX];
    executionType isPipe;       //-u → 0|-p →1
    Estado estado;              //A_EXECUTAR, EXECUTADO, POR_EXECUTAR
    int id;                     //Identificador da tarefa
    int time;                   //Tempo esperado de execução da tarefa
    long executionTime;         //Tempo de execução da tarefa
    PriorityLevel priority;
}Instruction;
```

O tamanho definido por *INSTRUCTION_SIZE_MAX* é de 300 *bytes*.

A outra estrutura criada foi a *InstructionQueue*, responsável por criar uma estrutura de lista ligada para a criação de filas de espera com instruções.

```
typedef struct InstructionQueue{
    Instruction* Instruction;
    struct InstructionQueue *next;
}InstructionQueue;
```

Esta *struct* define uma lista de instruções como uma lista ligada de instruções, com um apontador para a próxima instrução.

3. Cliente

O programa cliente tem como objetivo o *parsing* e envio de instruções para o servidor. As instruções tanto podem ser de execução de tarefas ou pedidos de *STATUS* para o servidor. Este status tem como objetivo recolher informações sobre processos que estejam a correr no servidor, à espera para serem executados ou até mesmo processos que já tenham sido executados.

Para que seja possível enviar ao servidor estes pedidos de execução ou *status*, o cliente tem que arranjar forma de comunicar com o servidor e arranjar também uma maneira de enviar todas estas informações.

O nosso cliente pode ser executado da seguinte forma:

```
./cliente execute 10 -u "args"
```

OU

```
./cliente status
```

Sendo a primeira um pedido de execução de tarefa ao servidor e a segunda um pedido de *status*.

A partir desta forma de arranque do nosso servidor podemos começar então por fazer um *parsing* da informação que pretendemos enviar para o servidor.

Para isto vamos utilizar o argumento *argv[]* da *main* no ficheiro. A partir de: *./cliente execute 10 -u "args"*, podemos retirar que *argv[0]* = *./cliente*, desnecessário para o que pretendemos, *argv[1]* = *execute*, ou *argv[1]* = *status*, necessário para informar o servidor qual é o tipo de execução que pretendemos, *argv[2]* = *10 (time)*, que é introduzido pelo *user* para informar quanto tempo aquela tarefa deve demorar, *argv[3]* = *-u* ou *-p*, que vai informar o servidor se a execução é de apenas um comando ou então a execução encadeada de comandos e por fim *argv[4]* = *"args"* que vão ser os argumentos que o utilizador pretende que sejam executados, por exemplo um *"ls -l"*. No nosso programa cliente, apenas argumentos com "aspas" no terminal são aceites.

Com recurso à função *parseArgsInstruction*, o nosso programa instancia uma instrução com estas informações provenientes do *argv*.

```
void parseArgsInstruction(int argc, char *argv[], Instruction *instruction){
    //./client(argv(0)) commandType[EXECUTE OR STATUS](argv(1)) Time(argv(2)).

    instruction->type = EXECUTE;
    instruction->pid = getpid();
    instruction->estado = A_EXECUTAR;
    instruction->time = atoi(argv[2]);
    instruction->executionTime = 0;
    instruction->id = -1;
}
```

O campo *pid* da nossa instrução vai ser importante para que o servidor consiga falar exclusivamente com este cliente. Isto é possível porque no nosso programa cliente vai ser criado um *FIFO* com um nome gerado tendo em conta este *pid*.

```
char fifoc_name[30];
sprintf(fifoc_name, CLIENT_NAME "%d", pid);

if(mkfifo(fifoc_name, 0666) == -1){
    perror("Error: Erro ao criar fifo do clien");
    return -1;
}
```

Na parte restante do código do cliente apenas é o *parsing* do tipo de execução, ou seja, se é um comando ou o encadeamento de comandos, e essa informação é guardada no campo *isPipe* do nosso cliente.

De forma semelhante ao bloco de código da parte da execução de comandos, temos também a parte relativa ao pedido de *status*, que só guarda nos campos da estrutura *Instruction*, o *pid*, igualmente necessário para o servidor conseguir responder ao cliente e o tipo de execução, neste caso *STATUS*.

Terminadas todas as operações de *parsing* de informações, a estrutura *Instruction* é enviada para o servidor via *FIFO*. Este *FIFO* é criado no servidor, e aberto no programa cliente com permissões de escrita.

```
int fd_fifocsv = open(SERVER_NAME, O_WRONLY);
if(fd_fifocsv == -1){
    perror("ERROR: Nao consegui abrir o fifo para escrever no server\n");
    return -1;
}

write(fd_fifocsv, instruction, sizeof(struct Instruction));
close(fd_fifocsv);
```

Por fim, é também aberto o *FIFO* "*fifoc_name*", único para cada cliente, que vai ser utilizado para receber a confirmação de pedidos por parte do servidor. Esta confirmação é depois escrita numa mensagem que é enviada pelo servidor no *FIFO* específico do cliente, lida no cliente e depois impressa no *standard output* ou 1 que é o descritor do *standard output*.

```
char message[350];
ssize_t bytes_read;
while((bytes_read = read(fd_fifocsv, &message, sizeof(message) - 1)) > 0){
    message[bytes_read] = '\0';
    write(1, message, bytes_read);
}

if (bytes_read < 0) {
    perror("ERROR: Erro ao ler a mensagem do servidor!");
    return -1;
}
```

Vamos agora mostrar um exemplo de um pedido de execução e um pedido de *status*.

Pedido de execução de um programa *hello* com argumento 10.

```
λ ./bin/client execute 2 -u "./progs-TP23_24/hello 10"
Task id: 2
```

Pedido de *status*:

```
└─λ ./bin/client status
A executar...

A executar

Processos Terminados:
Instrução:ID: 0
ARGS: ls -l
Execution Time: 4 ms

nova
Instrução:ID: 1
ARGS: ls -l
Execution Time: 6 ms

Instrução:ID: 2
ARGS: ./progs-TP23_24/hello 10
Execution Time: 11449 ms
```

4. Orchestrator

Explicado o nosso programa cliente, vamos passar a explicar o nosso pensamento para a implementação do servidor, ou *orchestrator*. É aqui que vão ser executadas as tarefas e pedidos dos clientes. Nesta parte do projeto apareceram diversos problemas que nos impossibilitaram de cumprir um dos requisitos do trabalho prático. De início pretendíamos implementar 4 políticas de escalonamento para que fosse possível comparar os diferentes tipos de tempo de execução e se possível escolher um melhor *allaround*. O problema foi a nossa incapacidade de executar os processos de forma sequencial, e então não conseguimos utilizar uma política de escalonamento. Neste preciso momento, o nosso escalonador funciona numa espécie de execução concorrente de processos, mas sem número máximo de processos a decorrer ao mesmo tempo. Com muita pena do grupo, não conseguimos solucionar este problema, mas pensamos que o problema possa estar relacionado com a gestão de *queues*, pondo esse problema de parte, temos a certeza que o resto do código funciona, e se, este problema for solucionado, os escalonadores feitos funcionam da forma pretendida.

Para que o servidor possa receber vários pedidos dos clientes, o mesmo não pode ficar bloqueado à espera de que o pedido seja processado. Para isso não acontecer, implementamos uma função *selectorPolicy*, que com base na política de escalonamento escolhida pelo utilizador ao começar o programa *orchestrator*, escolhe uma forma de colocar a instrução numa fila de espera, gerida de acordo com essa política. No caso do nosso trabalho temos 2 políticas de escalonamento. Uma FCFS, que coloca a instrução no final da fila de espera. Uma de SJF, que coloca o processo com menor tempo de execução, indicado pelo utilizador no cliente em primeiro lugar para ser executado. Depois dos processos serem adicionados à respetiva *queue*, os mesmo vão ser executados pelas funções *execCommandUnique*, se o comando for único ou pela função *executeTaskPIPELINE*, se o comando for um encadeamento de comandos. Após a execução das instruções, as mesmas são removidas das *queues*.

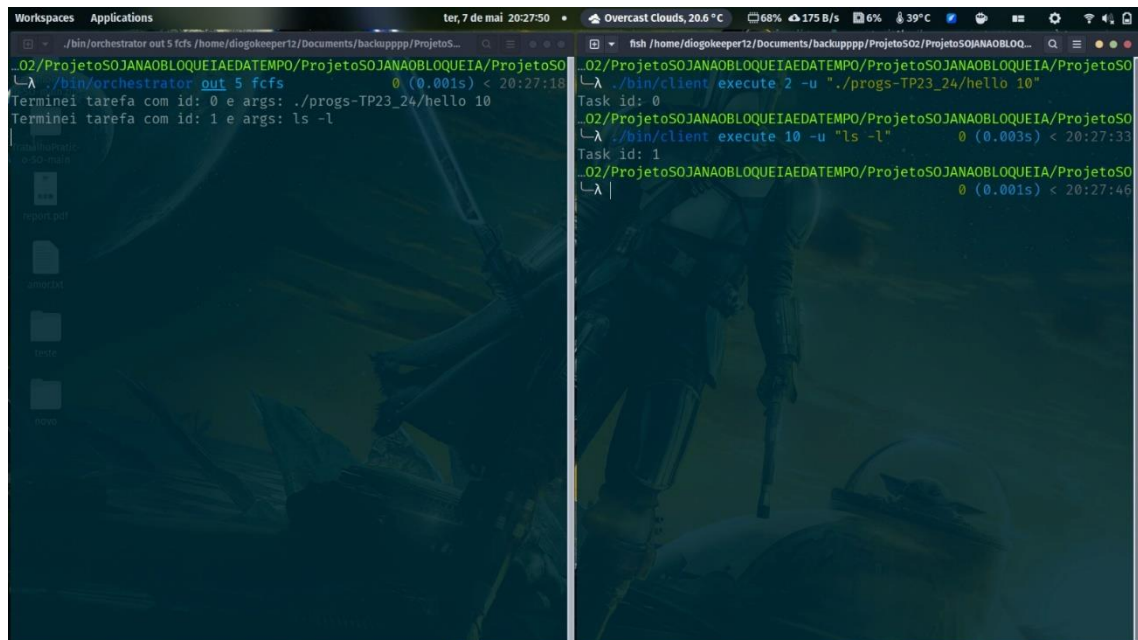
Estas duas funções, anteriormente citadas, realizam um *fork* dentro de um *fork*. Decidimos realizar a execução de comandos desta forma para prevenir que os clientes fiquem bloqueados pelo servidor enquanto estão processos a decorrer. Isto também previne que os processos filhos fiquem zombies à espera de serem recolhidos pelo processo pai. Por último, implementamos este método de execução para satisfazer uma das “funcionalidades avançadas” que era a execução paralela de N processos ao mesmo tempo. Embora conseguimos fazer execução paralela de processos, não conseguimos limitar o número de processos a correr ao mesmo tempo. Sendo este um dos pontos que gostaríamos de conseguir colocar a funcionar para a apresentação final do relatório.

Para a execução dos processos com encadeamento de comandos, utilizamos um esboço realizado numa aula prática, em que tivemos de ter em atenção a abertura e fechos de extremos de escrita e de leitura para evitar bloqueios.

Ambos estas funções redirecionam o seu output para ficheiros persistentes.

A função *execStatus*, responsável por processar pedidos do tipo de execução *Status*, itera sobre todas as instruções na *queue*. O problema é que a *queue* também não está funcionando, como já foi citado em cima, o que impossibilita ver as instruções que estão agendadas para executar e as que estão a executar. As que já foram executadas, são obtidas a partir do ficheiro output das nossas funções *exec*.

Exemplo output da função *execCommandUnique*.



The image shows two terminal windows side-by-side. The left window has a title bar that reads "/bin/orchestrator out 5 fcfs /home/diogokeeper12/Documents/backupppp/ProjetoS...". The right window has a title bar that reads "fish /home/diogokeeper12/Documents/backupppp/ProjetoSO2/ProjetoSOJANA...". Both windows show a prompt character (λ) followed by a command and its output. The left window shows the command `./bin/orchestrator out 5 fcfs` and its output `0 (0.001s) < 20:27:18`, followed by two lines of text: `Terminei tarefa com id: 0 e args: ./progs-TP23_24/hello 10` and `Terminei tarefa com id: 1 e args: ls -l`. The right window shows the command `./bin/client execute 2 -u './progs-TP23_24/hello 10'` and its output `Task id: 0`, followed by the command `./bin/client execute 10 -u 'ls -l'` and its output `Task id: 1`, and finally the command `./bin/client execute 10 -u 'ls -l'` and its output `0 (0.003s) < 20:27:33`.

```
./bin/orchestrator out 5 fcfs 0 (0.001s) < 20:27:18
λ ./bin/orchestrator out 5 fcfs 0 (0.001s) < 20:27:18
Terminei tarefa com id: 0 e args: ./progs-TP23_24/hello 10
Terminei tarefa com id: 1 e args: ls -l

fish /home/diogokeeper12/Documents/backupppp/ProjetoSO2/ProjetoSOJANA...
λ ./bin/client execute 2 -u './progs-TP23_24/hello 10'
Task id: 0
λ ./bin/client execute 10 -u 'ls -l' 0 (0.003s) < 20:27:33
Task id: 1
λ ./bin/client execute 10 -u 'ls -l' 0 (0.001s) < 20:27:46
```

Pedido execução de dois comandos por parte do cliente para o server. Em que o output foi guardado nestes ficheiros:

nosso grupo. Todos os mecanismos de comunicação entre os programas clientes e servidores também foram desenvolvidos de forma trivial. Ahamos também que as nossas funções de execução de comandos, quer únicos quer de encadeamentos de comandos também foram realizadas com bastante sucesso, inclusive o redireccionamento para os respetivos ficheiros. A parte que não nos correu tão bem, e que acabou por prejudicar e atirar para baixo o nosso projeto, foi a gestão e escalonamento de processos, mais concretamente as *queues*. Nunca conseguimos encontrar uma forma de trabalhar com as *queues* de forma correta. Sempre que conseguíamos compilar corretamente os programas com as *queues* e adicionar instruções as mesmas, todas os pedidos dos clientes ficavam bloqueados, o que ia de contra o objetivo do trabalho. Portanto, enquanto grupo reunimos e achamos por bem apresentar o trabalho com a execução concorrente de processos, mesmo não sendo escalonados da forma correta. Ahamos a 100% que se vissemos resolvido este problema, tudo o resto ganhava uma vida extra e funcionaria 100% como pretendido. Iremos sem dúvida tentar resolver este problema a tempo da apresentação final, coisa que não foi possível fazer para a entrega deste relatório.

Ahamos por isso, que a maior parte dos temas abordados nesta UC foram realizados de forma satisfatória, e teriam mesmo sido todos, senão os problemas que encontramos.