Universidade do Minho

Sistemas Operativos

Ciências da Computação

Bruno Fernandes
(A95972)

Marta Pereira
(A97402)

Tiago Silva
(A97450)

6 de maio de 2024

Índice

Capítulo 1

Introdução

Este relatório foi elaborado no âmbito do trabalho prático da Unidade Curricular de Sistemas Operativos, no qual foi proposta a implementação de um serviço de orquestração de tarefas. Neste, o cliente recebe tarefas do utilizador e envia-as ao servidor que, por sua vez, realiza o escalonamento e a execução das mesmas. O cliente é, também, capaz de consultar o servidor para saber quais as tarefas em execução, em espera e terminadas.

Capítulo 2

Funcionalidades

2.1 Funcionalidades Básicas

2.1.1 Execução de tarefas do utilizador

O cliente recebe as tarefas do utilizador através de um comando *execute* que indica o tempo de execução, o nome do programa e os seus argumentos. Essa tarefa é enviada ao servidor. Este tem a capacidade de escalonar as tarefas recebidas de duas formas (*FCFS, SJF*). Após a execução de cada tarefa o *standard input* e o *standard error* são redirecionados para um ficheiro com o identificador da tarefa.

2.1.2 Consulta de tarefas em execução

Os utilizadores podem utilizar o comando *status* através da linha de comandos para obter a lista de todas as tarefas: as que estão em execução, as que estão em espera e as terminadas.

2.2 Funcionalidades Avançadas

2.2.1 Execução encadeada de programas

O cliente tem a possibilidade da execução encadeada de programas do utilizador (*pipelines*). Neste comando são inseridos vários programas com os seus argumentos, separados por uma barra vertical.

2.2.2 Processamento de várias tarefas em paralelo

O servidor consegue processar várias tarefas em simultâneo, sendo o número de tarefas indicado no arranque do mesmo.

Capítulo 3

Implementação

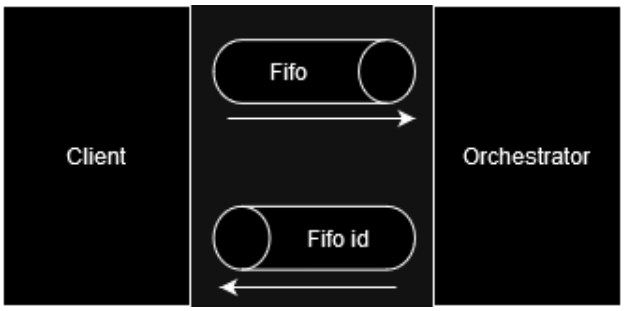

Figura 3.1: `Comunicação Client-Orchestrator`

3.1 Cliente (*client.c*)

O cliente recebe, do utilizador, um de dois comandos: *execute* ou *status*. O comando *execute* envia ao servidor as tarefas pedidas para serem executadas. A informação desse comando é colocada numa estrutura do tipo *program* que posteriormente é enviada ao servidor através de um *fifo*. Ainda antes desse envio, é criado um *fifo* de retorno, cujo nome é o *PID* do processo de modo a não haver trocas no destinatário do envio pelo servidor. Através deste, *fifo* o servidor envia ao programa cliente o identificador único da tarefa, para que este seja comunicado ao utilizador.

```
tiago@tiago-IdeaPad-3-15IML05:~/SO/ProjetoSO/bin$ ./client execute 11000 -p "./hello 10 | grep paths | wc -l "
TASK 10144 Received
```

Figura 3.2: `Exemplo de um comando execute`

Da mesma forma que o comando *execute*, o comando *status* é colocado na estrutura *program* e enviado ao servidor. Também é criado um *fifo* pelo qual o programa cliente recebe a informação dos programas que estão a ser executados, que estão em espera e que já terminaram. Esta informação é apresentada ao utilizador da seguinte forma:

```
tiago@tiago-IdeaPad-3-15IML05:~/SO/ProjetoSO/bin$ ./client status
Executing
 10951 ./hello 10
 10954 ./hello 10 | grep paths | wc -l
Scheduled
 10963 ./void 7
 10961 ./hello 10
Completed
 10945 ./void 7 7003 ms
```

Figura 3.3: `Exemplo de um resultado do comando status`

3.2 Servidor (*orchestrator.c*)

O servidor começa por receber, através de um *fifo*, a estrutura com a tarefa do cliente. A nossa implementaçao contém duas listas do tipo *struct program* para guardar em memória a informação dos programas que estão a executar (*executing*) e dos que estão em espera (*ready*). Para além disso, a informação dos programas que já terminaram é escrita no ficheiro *terminados.txt*.

```
struct program
{
    int type;
    int state;
    char args[MAX_PROG];
    int time;
    int start_time;
    int id;
    int pid;
    struct program* next;
};
```

Figura 3.4: *Struct program*

O servidor verifica, através da variável *type*, se o comando é um *status*, um *execute -u* (um único comando), um *execute -p* (*pipeline*) ou se é para recolher o estado de algum processo.

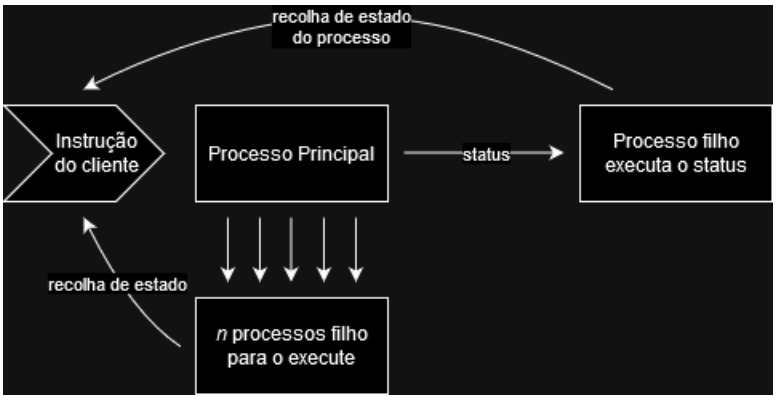

Figura 3.5: `Diagrama de processos do servidor`

3.2.1 *status*

Para o primeiro caso, começa por criar um processo para não bloquear o servidor de receber mais pedidos dos clientes. Abrimos o *fifo* cujo nome é a variável *id* da estrutura e, de seguida, enviamos os que estão na lista *executing*, os que estão na lista *ready* e os que estão no ficheiro *terminados.txt*. Por fim, alteramos as variáveis *PID*, *type* e *id* da estrutura recebida. A variável *PID* toma o valor do *PID* do processo que está a executar o status, o *type* é atualizado para 3 para indicar que queremos recolher o estado e o *id* é atualizado

para -1, para que, na recolha do estado, o servidor tenha a informação de que é para recolher o estado de um comando *status* e assim não libertar qualquer *slot* de execução de tarefas, visto que, a execução do *status* não é considerada uma tarefa e não é sujeita a escalonamento. Enviamos esta estrutura para o *fifo* do servidor.

3.2.2 *execute*

Para o caso do comando *execute*, inicialmente, enviamos, através do *fifo* cujo nome é a variável *id* da estrutura, ao programa cliente o identificador único da tarefa (escolhemos como *id* da tarefa o *PID* que recebemos na estrutura do programa cliente). Em seguida, verificamos se existem *slots* disponíveis para a execução.

Caso não hajam, guardamos o momento em que a tarefa chegou ao servidor, com a função *gettimeofday()*, e guardamos este valor junto da informação recebida na lista *ready*, de acordo com o critério de escalonamento definido no arranque do servidor.

Caso hajam, guardamos o tempo de execução inicial com a função *gettimeofday()*. Em seguida, inserimos a informação da tarefa na lista *executing* e, de forma a não bloquear o servidor, criamos um processo filho que servirá para executar o comando ou a sequência de comandos (*pipeline*).

Já dentro deste processo filho e ainda antes da execução, criamos uma estrutura chamada *retorno* com a informação da tarefa que vamos executar mas com o *type* igual a 3 (indica que queremos recolher o estado de um processo que já terminou a execução) e *PID* igual ao *PID* do processo criado (*getpid()*).

Abrimos o ficheiro com o identificador da tarefa e redirecionamos o *standard output* e o *standard error* desse programa para esse ficheiro através das funções *dup* e *dup2*.

Para executar, verificamos se é um único comando ou uma *pipeline* (-u ou -p) e deixamos a execução para as respetivas funções.

Logo após o término da execução do programa, esta estrutura é enviada para o *fifo* que o servidor lê, para que este recolha o estado do processo e guarde o momento em que terminou a execução (*gettimeofday()*).

3.2.3 Recolha do estado do processo

Para o último caso, começamos por recolher o estado do processo. Em seguida, verificamos se o valor da variável *id* é diferente de -1. Neste caso, sabemos que estamos a recolher o estado de um processo que executou uma tarefa (*execute*) e, portanto, guardamos o momento em que terminou a execução (*gettimeofday()*) e calculamos o tempo total da execução da tarefa. De forma a terminar esta tarefa, guardamos no ficheiro *terminados.txt* o identificador, o comando e o tempo de execução através da estrutura *Wrt*. Removemos a tarefa com o identificador correspondente da lista *executing*.

```
typedef struct wrt{
    int pid;
    char args[MAX_PROG];
    int time;
} Wrt;
```

Figura 3.6: *Struct Wrt*

Como sabemos que uma tarefa terminou, verificamos se a lista *ready* está vazia. Caso esteja, libertamos um *slot* de execução. Para o caso em que temos tarefas na lista *ready*, removemos a tarefa que está à cabeça e esta é executada da mesma forma que seria executada se aquando da sua chegada tivesse slots disponíveis.

3.2.4 Funções auxiliares

As funções auxiliares utilizadas foram as seguintes:

- *executec*: Executa um comando. Separa os argumentos pelos espaços e coloca-os numa lista. Cria um processo filho para executar o comando, através da função *execvp*.
- *executep*: Executa uma *pipeline*. Cria *pipes* anónimos para os comandos comunicarem entre si.
- *exec_command*: Semelhante à função *executec*, exceto na criação do processo filho. Utilizada para executar cada comando das *pipelines*.
- *send_completed*: Lê do ficheiro *terminados.txt* as tarefas terminadas, guardando numa estrutura do tipo *Msg* o *PID*, os comandos e o tempo de execução.
- *send_scheduled*: Percorre a lista *ready*, recolhe a informação do *PID* e dos comandos das tarefas e utiliza a estrutura *Msg* para enviar a informação para o cliente.
- *send_executing*: Percorre a lista *executing*, recolhe a informação do *PID* e dos comandos das tarefas e utiliza a estrutura *Msg* para enviar a informação para o cliente.
- *insere*: Insere um programa na lista de acordo com a política de escalonamento escolhida. Caso seja *FCFS* acrescenta no fim. Caso seja *SJF* insere ordenadamente na lista, pelo tempo esperado de execução.
- *removeq*: Retira o primeiro elemento de uma lista e retorna-o.
- *removeExecuting*: Retira o elemento correspondente a um certo *PID* de uma lista e retorna-o.

Capítulo 4

Avaliação de políticas de escalonamento

Neste projeto, implementamos duas políticas de escalonamento: *FCFS* e *SJF*. Para testar a implementação acima descrita, criamos um *script* de teste igual para ambas as políticas. Este teste envia um número de tarefas escolhido pelo utilizador, sendo que a primeira tem um tempo previsto de 20.000 ms e é decrementado 1.000 ms por cada tarefa que é enviada ao servidor para execução.

Este teste foi feito num servidor com capacidade para executar 3 tarefas em paralelo.

```
tiago@tiago-IdeaPad-3-15IML05:~/SO/ProjetoSO$ ./testetempo.sh 10
./client execute 20000 -u "./hello 20"
TASK 7972 Received
./client execute 19000 -p "./hello 19 | tail -n 100 | head -n 10 | grep paths | wc -l "
TASK 7975 Received
./client execute 18000 -u "./hello 18"
TASK 7977 Received
./client execute 17000 -p "./hello 17 | tail -n 100 | head -n 10 | grep paths | wc -l "
TASK 7985 Received
./client execute 16000 -u "./hello 16"
TASK 7986 Received
./client execute 15000 -p "./hello 15 | tail -n 100 | head -n 10 | grep paths | wc -l "
TASK 7987 Received
./client execute 14000 -u "./hello 14"
TASK 7988 Received
./client execute 13000 -p "./hello 13 | tail -n 100 | head -n 10 | grep paths | wc -l "
TASK 7989 Received
./client execute 12000 -u "./hello 12"
TASK 7990 Received
./client execute 11000 -p "./hello 11 | tail -n 100 | head -n 10 | grep paths | wc -l "
TASK 7991 Received
```

Figura 4.1: `Execução do` *`script`* `de teste`

```
tiago@tiago-IdeaPad-3-15IML05:~/SO/ProjetoSO/bin$ ./client status
Completed
 7977 ./hello 18 20008 ms
 7975 ./hello 19 | tail -n 100 | head -n 10 | grep paths | wc -w 21023 ms
 7972 ./hello 20 22290 ms
 7986 ./hello 16 38927 ms
 7987 ./hello 15 | tail -n 100 | head -n 10 | grep paths | wc -w 38927 ms
 7985 ./hello 17 | tail -n 100 | head -n 10 | grep paths | wc -w 39025 ms
 7990 ./hello 12 52218 ms
 7989 ./hello 13 | tail -n 100 | head -n 10 | grep paths | wc -w 53188 ms
 7988 ./hello 14 54292 ms
 7991 ./hello 11 | tail -n 100 | head -n 10 | grep paths | wc -w 64294 ms
```

Figura 4.2: Com a política *FCFS*

```
tiago@tiago-IdeaPad-3-15IML05:~/SO/ProjetoSO/bin$ ./client status
Completed
 8227 ./hello 18 19728 ms
 8225 ./hello 19 | tail -n 100 | head -n 10 | grep paths | wc -w 20871 ms
 8222 ./hello 20 22020 ms
 8241 ./hello 11 | tail -n 100 | head -n 10 | grep paths | wc -w 31904 ms
 8240 ./hello 12 34152 ms
 8239 ./hello 13 | tail -n 100 | head -n 10 | grep paths | wc -w 36369 ms
 8238 ./hello 14 47438 ms
 8237 ./hello 15 | tail -n 100 | head -n 10 | grep paths | wc -w 50727 ms
 8236 ./hello 16 54077 ms
 8234 ./hello 17 | tail -n 100 | head -n 10 | grep paths | wc -w 66116 ms
```

Figura 4.3: Com a política *SJF*

Média do *FCFS*: 40419.2ms

Desvio padrão do *FCFS*: 14807.5ms

Média do *SJF*: 38340.2ms

Desvio padrão do *SJF*: 14953.7ms

Após a observação desta estatística, observamos que com a estratégia *SJF* obtemos um sistema mais eficiente, permitindo que as tarefas tenham menor tempo de espera. Apesar de neste teste a diferença entre

os devios padrão não ser muito significativa, conseguimos perceber que a estratégia *SJF* será aquela que tenderá a ter um desvio padrão superior, isto porque, quando é enviada ao servidor uma tarefa com tempo de execução elevado, esta tem que esperar que todas as tarefas mais curtas terminem a execução, e se estiverem sempre a chegar tarefas mais curtas esta pode nunca executar (*starvation*).

Quanto maior for a capacidade de paralelização do servidor menor será o tempo de espera. Testando o *script* acima numa versão do servidor com capacidade para apenas 2 tarefas em paralelo o tempo médio de foi 51398.1ms para a estratégia *SJF* e 56315.5ms para a estratégia *FCFS*, uma diferença bastante significativa em relação à versão do servidor com 3 tarefas em paralelo.

Capítulo 5

Conclusão

O nosso trabalho executa corretamente todas as funcionalidades pedidas. Concluímos, por isso, que foi bem conseguido. Apesar de algumas dificuldades que surgiram ao longo do seu desenvolvimento, como por exemplo, na escolha da estrutura mais adequeada para tratamento da informação necessária na gestão dos slots para a execução de tarefas ou até na execução das *pipelines*. Assim, consolidamos os nossos conhecimentos de Sistemas Operativos de uma forma prática e motivadora. Para além disso, a resolução dos guiões práticos auxiliou na busca e e implementação da nossa solução.