



Universidade do Minho
Escola de Ciências

Orquestrador de Tarefas Relatório

Trabalho realizado por:
Alexis Correia - A102495
João Fonseca - A102512
Ricardo Vilaça - A102879

04 de maio de 2024

Sumário

Introdução	3
Código	4
Bibliotecas e Estruturas.....	4
Cliente	5
Orquestrador	5
Testes.....	6
Conclusão.....	8

Introdução

Desenvolvemos, de acordo com o que nos foi pedido, um sistema de execução e escalonamento de tarefas. Este relatório visa apresentar as metodologias e os processos lecionados que foram utilizados durante o projeto que tem vindo a ser realizado no âmbito desta UC.

Para a realização deste projeto foram criados principalmente 2 programas, um que é utilizado pelo cliente para que possa submeter tarefas e o outro programa – o orquestrador - que interage com o cliente e que é responsável por escalonar e executar as tarefas, mantendo em memória e em ficheiros a informação relevante para suportar as funcionalidades descritas. Pretendemos, dessa forma, explicar a lógica e as funções por de trás do código desenvolvido pelo nosso grupo.

Código

Este trabalho é constituído por quatro ficheiros, nomeadamente:

- **client.c**
- **orchestrator.c**
- **orchestrator.h**
- **Makefile**

Primeiramente, é importante notar que os códigos do cliente e do orquestrador (ficheiros com extensão “.c”) estão guardados na pasta **src** enquanto o ficheiro “.h” se encontra na pasta **include**. Dessa forma, após a utilização do comando “make” - de acordo com o Makefile disponibilizado pelos professores no guião deste trabalho prático - serão criadas as pastas **obj**, aonde será armazenado o *client.o* e *orchestrator.o*, a pasta **bin**, aonde se encontrarão os executáveis de cada código, e por fim a pasta **tmp**.

Com os executáveis compilados e as pastas prontas, primeiro executamos o orquestrador com o seguinte comando:

./orchestrator output_folder parallel-tasks sched-policy

O *output_folder* será o nome do ficheiro log – que criaremos na pasta *tmp* - aonde serão armazenadas as tarefas concluídas, *parallel-tasks* se refere ao número de tarefas independentes que podem ser executadas simultaneamente pelo computador e, por fim, *sched-policy* se refere a política de escalonamento que organizará a ordem de execução das tarefas.

Após isso, em outro processo, podemos executar o cliente. No cliente, podemos requisitar a execução de uma tarefa, que pode ser um comando único ou uma pipe de comandos, ou pedir um status das tarefas já submetidas.

./client execute [...]

Ou

./client status

Agora, explicaremos em mais detalhe o funcionamento de cada um destes códigos, começando pelo *orchestrator.h*.

Bibliotecas e Estruturas

O ficheiro *orchestrator.h* é o ficheiro no qual importamos as bibliotecas necessárias para a utilização das funções lecionadas em sala bem como de outras importantes funções. Estas bibliotecas são:

- **stdio.h**
- **stdlib.h**
- **unistd.h**
- **fcntl.h**
- **string.h**
- **sys/wait.h**
- **sys/types.h**
- **sys/stat.h**
- **sys/time.h**

Para além das bibliotecas, é neste mesmo ficheiro no qual definimos as duas estruturas (*structs*) que utilizaremos mais à frente. A primeira estrutura é a estrutura **Task** no qual armazenamos as informações sobre cada tarefa como, por exemplo, a string com o(s) comando(s) e argumento(s), o tipo de tarefa (“-u” ou “-p”), o tempo (previsto/real) de execução, o número de identificação único de cada tarefa, o PID do processo que envio

tal tarefa. A segunda struct a ser criada foi a **LTask**, uma lista ligada de tarefas, cujo utilizaremos para armazenar em memória as tarefas que estão e ser executadas no momento pelo orquestrador, bem como as tarefas que estão aguardando de acordo com o escalonamento selecionado.

Client

O programa *client.c* é, em sua essência é apenas a porta de entrada para o orquestrador. O *client* envia as tarefas submetidas pelo standard input (stdin) através de uma *pipe* nomeada (ou FIFO) - *server_pipe*, criado anteriormente em *orchestrator.c* - e recebe as respostas dos pedidos através de um novo FIFO, cujo nome é o PID (Process Id) do processo em questão, para receber a resposta do servidor. Então, o *cliente* é responsável por ler o input e imprimir a resposta na saída padrão (stdout). Neste caso, a struct *Task* é de extrema importância para organizar e encapsular os dados, facilitando assim a passagem destes mesmos entre cliente e servidor.

Na função **main**, o programa começa a verificar se foram passados argumentos suficientes na linha de comando, caso contrário, imprime as opções disponíveis e encerra a execução com código de erro -1. Em seguida, é criada uma variável *task* -do tipo *Task* - para armazenar os dados da tarefa a ser executada.

O programa então analisa os argumentos passados na linha de comando e preenche os campos da estrutura *Task* de acordo com o comando fornecido. Se o comando for "execute", todos os campos da struct são preenchidos de acordo. Se o comando for "status", a estrutura *Task* é preenchido com o próprio comando e com o PID, enquanto o tempo e o tipo são atribuídos o valor -1. É importante notar que a atribuição do *id* da tarefa é responsabilidade do orquestrador, então sempre é atribuído o valor de -1 pelo cliente. O código também confere se o comando do cliente é "rem", mas este é um caso especial que explicaremos mais a frente.

Em seguida, inicia a comunicação com o orquestrador (através do *server_pipe*) em modo de escrita e escreve a estrutura *Task* nele. Neste projeto *server_pipe* será o pipe utilizado pelos clientes para enviarem o pedido de execução ao servidor. Depois, o programa cria um nome de pipe único usando o PID como nome (isso é necessário para que o servidor saiba para qual cliente enviar a resposta, já que é possível que haja múltiplos clientes acedendo o servidor em simultâneo), e então o abre em modo de leitura.

A leitura dos dados enviados pelo orquestrador no *client_pipe* são lidos com o auxílio de um ciclo *while* e impressos na saída padrão. Após a leitura de todos os dados, os pipes são fechados e o pipe do cliente é removido do sistema com auxílio da função **unlink**. Finalmente, o programa encerra com código de retorno 0.

Orchestrator

Agora, é no programa *orchestrator.c* onde serão realizados o maior número de processos, como escalonamento e execução das tarefas. Sendo assim, implementamos uma vasta quantidade de funções auxiliares de tal forma que, não só é mais a fácil a alteração do código, mas também a sua legibilidade e perceptibilidade.

O programa começa com a definição de algumas funções auxiliares que se apresentam vitais na função *main*. A função **func_aux** divide uma string de argumentos em um array de strings, separando-os pelo caractere '|'; **command** divide uma string de comando em um array de strings, separando-os pelo caractere espaço; **length** retorna o número de elementos em uma lista ligada (de tarefas); **insertOrd** insere uma nova tarefa em uma lista encadeada de tarefas de forma ordenada de acordo com o tempo previsto de

execução; **rem** remove uma tarefa de uma lista encadeada de tarefas, com base em seu identificador; por ultimo, **append** adiciona uma nova tarefa no final de uma lista encadeada de tarefas.

As funções *insertOrd* e *append* são utilizadas para organizar as tarefas de acordo com uma política de escalonamento, respetivamente Shortest Job First (SJF) e First-Come First-Serve (FCFS). Em relação aos algoritmos de escalonamento foram aplicadas, como estruturas de dados, listas ligadas devido ao facto de operações de inserção e remoção em uma lista ligada poderem ser realizadas de forma simples, outra razão que levou a esta decisão provém do simples facto de o grupo se considerar mais experiente a trabalhar com listas ligadas do que com outras estruturas de dados.

Resumidamente, no que toca á função principal do programa, esta realiza as seguintes tarefas:

1. Verifica se o número correto de argumentos foi passado na linha de comando e exhibe uma mensagem de execução caso contrário;
2. Determina a função ser utilizado com base no algoritmo de funcionamento fornecido;
3. Abre um arquivo de log especificado na linha de comando para registo das operações;
4. Cria um pipe nomeado (“server_pipe”) para comunicação com os clientes;
5. Inicializa variáveis e estruturas de dados necessárias para o controle das tarefas.
6. Entra em um loop para receber e processar comandos do servidor - é neste loop, mencionado acima, onde são realizadas as tarefas dos clientes.

Sobre a execução de tarefas, quando “-u” é digitado na linha de comando pelo utilizador, é criado um processo filho para executar o comando especificado pela tarefa. O descritor de arquivo é duplicado para redirecionar a saída padrão (stdout) para o arquivo, e o tempo de início da execução é registado. Outro processo filho é criado para executar o comando, e o tempo de execução da tarefa é calculado quando a execução é concluída. O resultado é então registado no arquivo de log junto com outras informações relevantes, e a tarefa é removida da lista de tarefas em execução para que as tarefas agendadas possam ser executadas.

Já no caso da execução de tarefas que envolvem a comunicação entre processos por meio de pipes anónimas, são criados um conjunto de pipes para conectar os processos envolvidos na execução da tarefa. Cada comando da tarefa é executado em um processo filho separado, e a saída de um comando é conectada à entrada do próximo por meio dos pipes e da função *dup2*. O tempo de início da execução da tarefa é registado, e o último comando é responsável por redirecionar sua saída para um arquivo de log. Após a conclusão da execução da tarefa, o tempo de execução é calculado e registado no arquivo de log. Caso ocorra algum erro durante a execução de um dos comandos, uma mensagem de erro é registada no arquivo correspondente a *Task*.

Em ambos os casos, seja um *execute -u* ou um *execute -p*, ao fim da execução da tarefa e do registo do tempo, antes que o processo filho se encerre, utilizamos a função *execvp* para executar o comando “./client rem [id]”. Este foi o método mais simples que achamos para que pudéssemos remover a tarefa concluída da lista de tarefas em execução, já que os processos pai e filho não compartilham a memória - e a lista se encontra na memória do pai.

Por fim, se o cliente pediu um *status* então, é realizada uma *fork* e o processo filho escreverá no *client_pipe* as informações relevantes sobre as tarefas. Primeiramente, escreverá as tarefas que estão na lista ligada *sch* (scheduled), seguido das tarefas que estão

na lista *ex* (executing) a serem executadas pelo orquestrador ou, mais especificamente, por processos filhos do orquestrado; em seguida, abrimos o ficheiro log, lemos as structs lá armazenadas e enviamos ao cliente, sempre cuidando para manter a formatação. É claro que, seguindo as instruções fornecidas, o ficheiro log está armazenada no disco enquanto as listas ligadas são guardadas em memória.

Testes

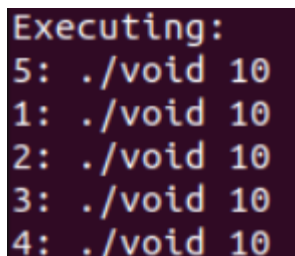
Seguindo a orientação do enunciado, realizamos alguns testes, com auxílio de scripts, para testar o funcionamento do orquestrador, em especial as políticas de escalonamento utilizadas. Também tiramos proveito dos códigos disponibilizados anteriormente, **hello.c** e **void.c**, na formulação destes scripts para confirmar o registo do tempo de execução das tarefas.

O código do script(.sh) era:

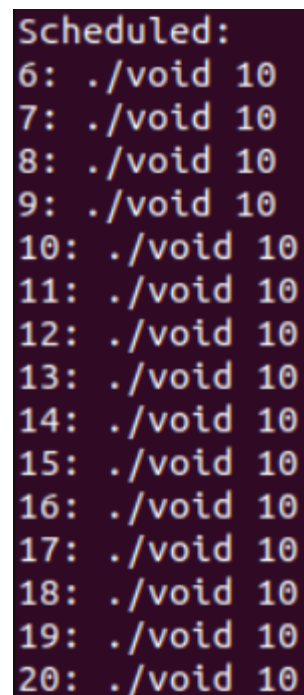
```
...
#!/usr/bin/bash

for(i=1; i<=$1; i++)
do
    x=$((RANDOM % 10000)+1))
    echo "./client execute ${x} -u ./void 10"    #./hello
    ./client execute "${x}" -u "./void 10"      #./hello
done
echo "./client status"
./client status
...
```

Como podemos observar nas imagens a seguir, quando a política de escalonamento é FCFS, as tarefas serão organizadas pelo seu id. O tempo previsto é irrelevante neste caso.



```
Executing:
5: ./void 10
1: ./void 10
2: ./void 10
3: ./void 10
4: ./void 10
```



```
Scheduled:
6: ./void 10
7: ./void 10
8: ./void 10
9: ./void 10
10: ./void 10
11: ./void 10
12: ./void 10
13: ./void 10
14: ./void 10
15: ./void 10
16: ./void 10
17: ./void 10
18: ./void 10
19: ./void 10
20: ./void 10
```

No caso de o escalonamento ser SJF, é preciso observar o valor do tempo ($\{x\}$) com cuidado.

```
./client execute 882 -u ./void 10
TASK 1 Received
./client execute 5966 -u ./void 10
TASK 2 Received
./client execute 2819 -u ./void 10
TASK 3 Received
./client execute 9641 -u ./void 10
TASK 4 Received
./client execute 7059 -u ./void 10
TASK 5 Received
./client execute 9154 -u ./void 10
TASK 6 Received
./client execute 2563 -u ./void 10
TASK 7 Received
./client execute 1404 -u ./void 10
TASK 8 Received
./client execute 5124 -u ./void 10
TASK 9 Received
./client execute 309 -u ./void 10
TASK 10 Received
```

```
./client execute 1243 -u ./void 10
TASK 11 Received
./client execute 1528 -u ./void 10
TASK 12 Received
./client execute 1189 -u ./void 10
TASK 13 Received
./client execute 2673 -u ./void 10
TASK 14 Received
./client execute 6626 -u ./void 10
TASK 15 Received
./client execute 597 -u ./void 10
TASK 16 Received
./client execute 4853 -u ./void 10
TASK 17 Received
./client execute 2795 -u ./void 10
TASK 18 Received
./client execute 6214 -u ./void 10
TASK 19 Received
./client execute 747 -u ./void 10
TASK 20 Received
```

```
Executing:
1: ./void 10
3: ./void 10
2: ./void 10
5: ./void 10
4: ./void 10
```

```
Scheduled:
10: ./void 10
16: ./void 10
20: ./void 10
13: ./void 10
11: ./void 10
8: ./void 10
12: ./void 10
7: ./void 10
14: ./void 10
18: ./void 10
17: ./void 10
9: ./void 10
19: ./void 10
15: ./void 10
6: ./void 10
```

Mas, de facto, chegamos a conclusão de que a função organiza as tarefas por tempo, do menor para o maior. No entanto, também realizamos os testes com outros comandos, como *wc*, *ls* e notamos que havia algumas irregularidades tanto no output, como tarefas repetidas múltiplas vezes no ficheiro log ou tarefas com o mesmo *id*, quanto no escalonamento de facto.

Conclusão

Para a realização deste trabalho, utilizamos todas as bibliotecas/funções lecionadas durante as aulas práticas, bem como o conhecimento acumulado das aulas teóricas para alcançar o resultado desejado.

Em resumo, o(s) cliente(s) se comunicam com o orquestrador pelo *server_pipe* e pelo *client_pipe* (criadas com a função *mkfifo*). As informações são manipuladas – seja ler ou escrever – com as funções *write* e *read*; e uma vez lidas, o orquestrador realiza todas as operações necessárias dentro de *forks* – processos filhos – para evitar que os clientes aguardem por uma resposta.

Na execução de comandos únicos (*flag -u*), são necessárias, principalmente, as funções *execvp* e *dup2* para que a tarefa seja executada e o resultado seja enviado para o ficheiro corretamente. Já para as tarefas encadeadas (*flag -p*), além das funções mencionadas acima, e da utilização de um ciclo **for** e **if's**, foi preciso da função *pipe* para criar pipes anónimas entre os processos filhos. Já no caso de tarefas “status”, também era necessário fazer um *fork*, enquanto as tarefas “rem” são realizadas rapidamente pelo próprio orquestrador devido a necessidade do acesso à memória.

Em conclusão, podemos afirmar que implementamos o conhecimento obtido durante a unidade curricular de **Sistemas Operativos** e realizamos o trabalho, apesar das dificuldades, com alto grau de comprometimento bem como de sucesso.