Universidade do Minho

Licenciatura em Engenharia Informática

Unidade Curricular de Sistemas Operativos

Ano Letivo de 2023/2024

Orquestrador de Tarefas

Beatriz Salgado Fernandes, a100602

João Silva Loureiro, a100832

SO

Maio, 2024

Índice:

Introdução:

No âmbito da cadeira de Sistemas Operativos, foi nos proposta a implementação de um serviço de orquestração de tarefas, num computador. Este relatório documenta o desenvolvimento desse serviço num ambiente Linux. O principal objetivo deste projeto é criar um sistema robusto que permita aos utilizadores submeterem tarefas para execução num servidor, com a capacidade de escalonamento eficiente e registo preciso do tempo de execução.

Para isso, foram desenvolvidos 2 programas principais, um cliente (cliente.c) e um servidor (orchestrator.c) em linguagem C, utilizando comunicação via pipes com nome (FIFOS) para garantir a interação entre os dois componentes.

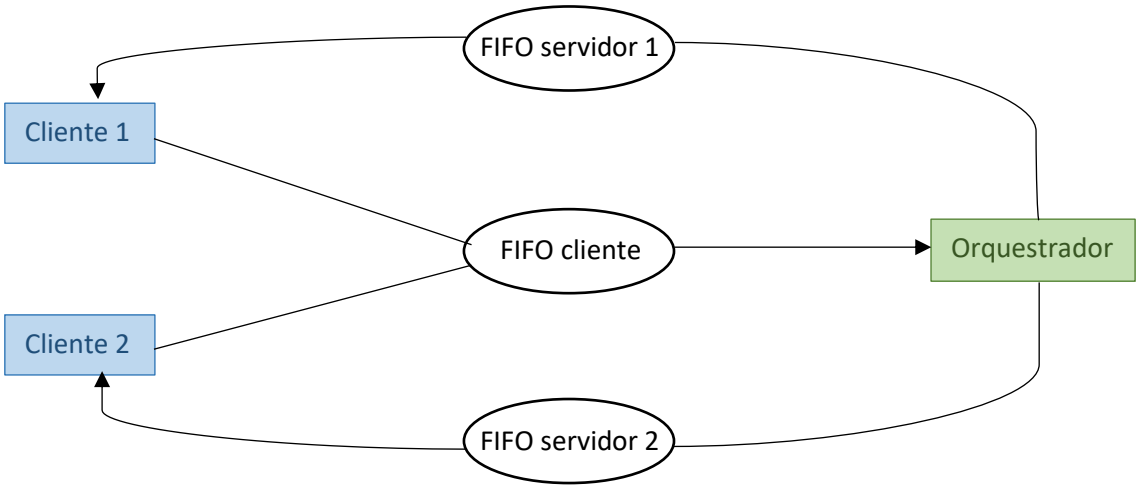
Ao longo deste relatório, serão apresentados detalhes sobre a arquitetura do sistema, as decisões tomadas, a implementação das funcionalidades exigidas e os desafios enfrentados durante o processo de desenvolvimento.

Espera-se que este relatório forneça uma visão abrangente do serviço desenvolvido, demonstrando não apenas a funcionalidade do sistema, mas também a qualidade e a eficiência da sua implementação.

Estrutura:

Como já foi referido na introdução, o nosso serviço é composto por 2 programas principais, o cliente.c e o orchestrator.c. Para além destes temos ainda um makefile que usado para compilar os nossos programas.

Todo o programa funciona à volta de pipes com nome (FIFOS). A forma como decidimos fazer esta comunicação está representada neste esquema.



O que isto quer dizer é que os clientes enviam os pedidos para o FIFO cliente, que é único e comum a todos os clientes. O servidor lê esse FIFO e manda a resposta para o FIFO servidor do cliente que mandou o pedido. Ou seja, servidor vai criar e ter um FIFO próprio para receber pedidos e os clientes vão criar o seu próprio FIFO para receber as respostas.

Esta decisão foi tomada tendo em conta que cada cliente vai ter os seus pedidos e não faria sentido receber as respostas dos pedidos de outros clientes, por isso, a melhor maneira de resolver este problema era cada cliente ter o seu FIFO servidor para receber as respostas aos seus pedidos.

client.c

Começando por explicar o programa ***client.c.*** Este é responsável por fornecer uma interface de linha de comando para os utilizadores interagirem com o sistema de orquestração de tarefas, permitindo-lhes submeter tarefas para execução, consultar o estado das tarefas e receber respostas do servidor.

O programa client suporta 2 tipos de pedidos, o execute e o status. O que ele faz é, lê o pedido do terminal e abre o FIFO do cliente para escrita, ou mais concretamente o FIFO de pedidos dos clientes.

```
if ((fd = open("fifoServerClient", O_WRONLY, 0660)) == -1) {
    perror("Erro ao abrir FIFO do cliente");
    return -1;
}
```

De seguida, constrói o comando completo com o id do cliente e o pedido específico, seja ele execute ou status, conforme o que leu no terminal.

```
char comando_com_id[50];
int_to_string(id, id_str);
strcpy(comando_com_id, id_str);
strcat(comando_com_id, " ");
strcat(comando_com_id, argv[2]);
```

Por último, escreve o comando completo no FIFO do cliente, que abriu anteriormente.

```
if (write(fd, comando_com_id, strlen(comando_com_id)) == -1) {
    perror("Erro ao escrever no FIFO do cliente");
    return -1;
}
```

O client.c cria ainda o FIFO do servidor, responsável por receber as respostas do orchestrator, onde é usado o id do cliente, de modo a termos um FIFO único para cada utilizador, como foi explicado acima.

```
char fifo_servidor[20] = "fifo_servidor_";
char id_str[10];
int_to_string(id, id_str);
string_concat(fifo_servidor, id_str);
```

```
if (mkfifo(fifo_servidor, 0660) == -1) {
    perror("Erro a criar FIFO servidor");
    return -1;
}
```

Depois de mandar o pedido, o client abre então este FIFO para leitura e imprime a resposta no terminal.

```
int fd_servidor;
if ((fd_servidor = open(fifo_servidor, O_RDONLY)) == -1) {
    perror("Erro ao abrir FIFO do servidor para leitura");
    return -1;
}
```

Por fim, fecha ambos os FIFOS de servidor e de cliente.

orchestrator.c

O ***orchestrator.c*** é o componente central do sistema. Este está encarregue de receber as solicitações dos clientes, escalonar e executar as tarefas de acordo com políticas predefinidas, além de gerenciar o registo das tarefas executadas e os seus respetivos tempos de execução.

Ao correr-mos o programa orchestrator.c, é criado o FIFO para o cliente. Logo a seguir é aberto para leitura e fica à espera de receber algum pedido.

```
if (mkfifo(FIFO, 0660) == -1) {
    perror("Erro a criar FIFO");
    return -1;
}
```

```
if ((fd_fifo = open(FIFO, O_RDONLY, 0660)) == -1) {
    perror("Erro a abrir FIFO");
    return -1;
}
```

Ao receber um pedido, ele verifica que tipo de pedido é, e trata-o de forma diferente, consoante as necessidades.

Para lidar com estes comandos, nós usamos pipes anónimos de forma a poder fazer o status independentemente de um execute estar a ser feito. Eles são tratados da seguinte forma:

- **status:**

```
if (strncmp(buffer + 2, "status", 6) == 0) {
    pid_t status_pid = fork();
    if (status_pid == 0) {
        // Processo filho executa o comando status
        mystatus("cat log.txt", fd_servidor);
        exit(0); // Termina o processo filho após a execução
    }
```

Quando o servidor recebe o comando "status" do cliente, ele cria um processo filho para lidar com a execução deste comando.

O processo filho executa o comando "cat log.txt" para ler o conteúdo do arquivo de log. Isto é feito através da função auxiliar *'mystatus'*, que recebe um comando de execução, e envia a sua saída para o FIFO do cliente, formatando a saída antes do envio. Para isto usamos o comando cat, que exige o conteúdo de um arquivo, com o ficheiro log, que tem a informação de todos os programas executados para obter a informação.

Por fim, o processo filho é terminado através do exit(0).

- **execute:**

Quanto ao execute, quando este é recebido, o número de pedidos vai ser incrementado e vai criar também um processo filho, tal como no caso acima. Por esta razão, os nossos programas passam também a ser executados em paralelo.

```
numero_pedidos++;

// Cria um filho para executar o comando
pid_t execute_pid = fork();
if (execute_pid == -1) {
    perror("Erro ao criar processo filho para execute");
} else if (execute_pid == 0) {
    // Processo filho
    write(fd_log, buffer, strlen(buffer));
    write(fd_log, "\n", 1);
```

De seguida, o comando é executado e o tempo de execução é calculado. Para tal usamos a função *'gettimeofday'* para obter a hora de início da execução e depois da execução usamos a função auxiliar *'get_elapsed_time'*, que recebe como argumento a hora de início. Esta usa a mesma função para obter a hora de fim da execução e depois calcula a diferença entre elas e passa o resultado para um double, em segundos.

```c
double tempo;
struct timeval start_time;
gettimeofday(&start_time, NULL);
mysystem(buffer + 10, numero_pedidos, fd_log, atoi(&buffer[0]));
tempo = get_elapsed_time(start_time);
```

```c
double get_elapsed_time(struct timeval start_time) {
    struct timeval end_time;
    gettimeofday(&end_time, NULL);
    return (double)(end_time.tv_sec - start_time.tv_sec) * 1000.0 + (double)(end_time.tv_usec - start_time.tv_usec) / 1000.0;
}
```

Para a execução do programa em si, é usada a função auxiliar *'mysystem'*. Esta função encapsula todo o processo de execução de um comando, desde a criação do arquivo de saída até o movimento desse arquivo para a pasta do cliente. Isso é alcançado através da criação de um ficheiro de saída, exclusivo para cada comando executado, garantindo que a saída do comando seja registada de forma organizada.

Após a execução do comando, a função espera pelo fim do processo filho, antes de fechar o ficheiro de saída e de o mover para a pasta designada do cliente, garantindo que a execução do comando seja concluída com segurança e que o resultado seja devidamente entregue ao cliente.

De volta à main, ele cria a mensagem, atualiza o log, constrói a mensagem de sucesso e escreve-a no FIFO servidor do cliente.

```c
create_message(numero_pedidos, buffer, tempo, mensagem);
int len = strlen(mensagem);

write(fd_log, mensagem, len);
write(fd_log, "\n", 1);

char mensagem1[100];
int len1 = 0;

strcpy(mensagem1, "Tarefa ");
char num_str[20];
int_to_string(numero_pedidos, num_str); // Converter o ni
strcat(mensagem1, num_str);
strcat(mensagem1, " concluída com sucesso\n");
len1 = strlen(mensagem1);
write(fd_servidor, mensagem1, len1);
// Escreve a mensagem no FIFO do cliente
write(fd_fifo, mensagem, len);

exit(0); // Termina o processo filho após a execução
```

Por fim, e tal como no status, fechamos o processo filho através do exit (0).

Comandos:

Para correr o programa, começamos por executar o nosso MakeFile com o comando *make.*

```
beatriz@Lenovo-V15-IIL:~/Desktop/SO/trabalho pratico$ make
gcc -Wall -g -o void void.c
gcc -Wall -g -o hello hello.c
gcc -Wall -g -o orchestrator orchestrator.c
gcc -Wall -g -o client client.c
```

Depois de termos compilado todos os nossos programas, podemos então começar a executa-los. Primeiro corremos então o nosso orquestrador de tarefas com o comando *./ochestrator*.

De seguida corremos os pedidos de tarefas dos clientes. Para tal usamos o comando ./client id execute t "prog-a arg-1 (...) arg-n" onde o id é o identificador do cliente, o t é o tempo estimado de duração da tarefa e o que tem a seguir é o programa que quer correr e os seus argumentos, caso tenha.

```
beatriz@Lenovo-V15-IIL:~/Desktop/SO/trabalho pratico$ ./client 1 10 "execute ./hello 10"
Tarefa 1 concluída com sucesso
```

```
beatriz@Lenovo-V15-IIL:~/Desktop/SO/trabalho pratico$ ./client 2 5 "execute ls -l"
Tarefa 2 concluída com sucesso
```

O cliente pode ainda ver o estado dos pedidos através do comando ./cliente id status.

```
beatriz@Lenovo-V15-IIL:~/Desktop/SO/trabalho pratico$ ./client 2 1 status
1 execute ./hello 10
A tarefa 1 com o pedido ./hello 10 foi executada em 10831.486 ms
2 execute ls -l
A tarefa 2 com o pedido ls -l foi executada em 1.588 ms
```

Por fim, quando tivemos acabado de usar o programa, usamos o comando *make clean* para limpar os arquivos gerados durante o processo de compilação.

Conclusão:

De uma forma geral, pensamos que este trabalho foi bem-sucedido na implementação das funcionalidades essenciais de um serviço de orquestração de tarefas. Apesar de não lidarmos diretamente com o encadeamento de programas, conseguimos satisfazer as expectativas básicas ao receber e executar tarefas pelo servidor e ao fornecer respostas ao cliente. Destacamos a capacidade do sistema de lidar com consultas de estado a qualquer momento, garantindo uma resposta consistente, mesmo durante a execução de múltiplas tarefas em paralelo.

Neste contexto, reconhecemos que o serviço apresenta uma base sólida para futuras expansões e refinamentos. Por exemplo, a inclusão do encadeamento de programas poderia aumentar significativamente a versatilidade e utilidade do serviço. Além disso, a implementação de políticas de escalonamento mais sofisticadas poderia otimizar ainda mais o desempenho do sistema em situações de carga variável.

Portanto, embora reconheçamos que há espaço para melhorias e desenvolvimentos adicionais, estamos satisfeitos com o resultado alcançado e acreditamos que este trabalho representa um passo importante na direção de um sistema de orquestração de tarefas eficiente e confiável.