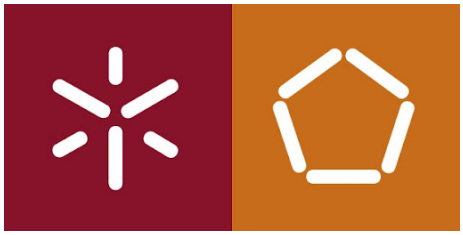


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Sistemas Operativos

Trabalho prático

7 de maio de 2024

Desenvolvido por:

Diogo Esteves - a104004
Rodrigo Ferreira - a104531
Rodrigo Fernandes - a104175

Conteúdo

1 Introdução

O objetivo deste relatório é clarificar o funcionamento e processo de desenvolvimento do projeto solicitado, bem como fazer uma reflexão acerta dos testes realizados.

Ae longo deste relatório vamos fazer uma breve explicação do projeto proposto e da forma como este foi desenvolvido. No final, vamos mostrar os testes realizados, e fazer uma reflexão acerca dos mesmos.

2 Servidor e cliente

Este trabalho consiste na implementação de um serviço de orquestração (i.e., execução e escalonamento) de tarefas num computador.

Os utilizadores devem usar um programa cliente para submeter ao servidor a intenção de executar uma tarefa, dando uma indicação da duração em milissegundos que necessitam para a mesma, e qual a tarefa (i.e., programa ou conjunto/pipeline de programas) a executar.

Cada tarefa tem associado um identificador único que deve ser transmitido ao cliente mal o servidor recebe a mesma. O servidor é responsável por escalonar e executar as tarefas dos utilizadores.

Através do programa cliente, os utilizadores podem ainda consultar o servidor para saberem quais as tarefas em execução, em espera para execução, e terminadas.

3 Cliente

Como referido anteriormente, o programa cliente tem o objetivo de submeter pedidos de execução ao servidor, bem como solicitar ao servidor um pedido de estado, para verificar o estado das tarefas em execução. Para que tal seja possível, o cliente escreve cada pedido ao servidor (quer pedido de tarefa, quer pedido de estado) através de uma 'struct', definida num ficheiro header 'scheduler.h', como uma 'struct' do tipo Task:

```
typedef struct task {
    int id;
    pid_t pid;
    char command[10];
    char program[300];
    char flag[3];
    int time;
    bool ocupation;
} Task;
```

Figura 1: Estrutura Task que armazena os pedidos do cliente

Na 'struct' Task estão presentes os seguintes campos:

- **id:** Esta variável é uma variável que será manipulada pelo servidor. Nela será armazenado o id da tarefa solicitada, aquando da sua receção por parte do servidor. É este 'id' que aparece no STDOUT do cliente quando este solicita a tarefa.
- **pid:** Esta é a variável onde o pid da tarefa é armazenado, assim que a tarefa entra em execução. O pid poderá, junto dos demais campos relevantes, ser consultado no ficheiro gerado pelo servidor (tasks.log).
- **command:** É nesta string, de comprimento fixo de 10 caracteres, que é armazenado o comando passado pelo cliente ao servidor. Este pode assumir os seguintes valores: 'execute', 'status', 'help', 'quit'.
- **program:** É nesta string que o programa e os seus argumentos é armazenado. O programa é o input, que o cliente passa na linha de comandos, que está entre aspas aquando de um comando execute (ex: ./client execute 300 -u **"ls -l"**).
- **flag:** String de 3 caracteres que armazena a flag a ser executada.
- **time:** Nesta variável, o servidor escreve o tempo, em milissegundos, que a tarefa levou a ser executada. No entanto, esta escrita do servidor é posterior à escrita por parte do cliente do tempo previsto, também em milissegundos, ou seja, numa fase inicial armazena o tempo previsto pelo cliente e numa fase final, o tempo real da tarefa.
- **ocupation:** Esta é uma variável que é usada pelas funções do módulo 'scheduler' para auxiliar à mainupalção da fila de espera (queue).

Para que o cliente possa fazer um pedido ao servidor, é necessário armazenar na struct, falada anteriormente, os diferentes campos do pedido, de maneira a uniformizar os pedidos, poupando assim o servidor de parsing adicional desnecessário. Assim o cliente é o único que faz o parsing do seu input e, para isso, são feita enúmeras verificações dos argumentos recebidos. Dependendo do resultado dessas verificações são colocados na struct os dados de forma específica, resultando em diferentes tipos de pedido, tais como os pedidos demonstrados abaixo.

3.1 O pedido de "execute"

```
diogoesteves@MacBook-Air-de-Diogo bin % ./client execute 10000 -u "../progs-TP23_24/void 10"
ID da tarefa: 3
```

Figura 2: Pedido execute

Aqui, faz-se uma verificação inicial para que seja analisada a ordem e sintaxe dos argumentos introduzidos. Se estiver tudo correto, passa-se para o preenchimento da struct para envio. De seguida a tarefa é escrita no fifo de comunicação entre o cliente e o servidor. Após a receção da tarefa, o servidor escreve neste mesmo fifo o id atribuído e o cliente imprime esse id no seu STDOUT.

3.2 O pedido de "status"

Para que o cliente saiba o estado atual das tarefas no servidor, é necessário o envio de um pedido de status ao servidor. Neste pedido, a resposta obtida por parte do servidor é o estado das tarefas em execução, já executadas e em espera.

```
diogoesteves@MacBook-Air-de-Diogo bin % ./client status

Executing:
7 ../progs-TP23_24/void 10
8 ../progs-TP23_24/void 10
9 ../progs-TP23_24/void 10

Schedule:
10 ../progs-TP23_24/void 10
11 ../progs-TP23_24/void 10
12 ../progs-TP23_24/void 10
13 ../progs-TP23_24/void 10
14 ../progs-TP23_24/void 10
15 ../progs-TP23_24/void 10
16 ../progs-TP23_24/void 10
17 ../progs-TP23_24/void 10
18 ../progs-TP23_24/void 10
19 ../progs-TP23_24/void 10

Executed:
1 ../progs-TP23_24/void 10 10015ms
2 ../progs-TP23_24/void 10 10009ms
3 ../progs-TP23_24/void 10 10017ms
4 ../progs-TP23_24/void 10 10007ms
```

Figura 3: Pedido status

3.3 Pedido de "help"

Se um cliente achar necessária ajuda para operar o servidor, pode enviar um pedido de ajuda ao mesmo, através do pedido de "help". Quando o mesmo envia este pedido, o servidor responde com uma mensagem de ajuda com as funcionalidades disponiveis.

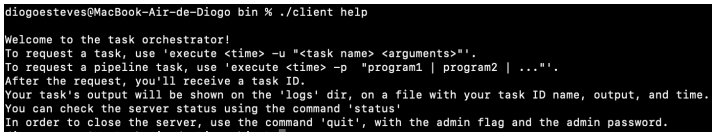


Figura 4: Mensagem de ajuda ao cliente

3.4 Pedido de "quit"

Para que seja possível o fecho do servidor, um cliente com o status de admin pode mandar um pedido de fecho ao mesmo, encerrando-se assim a atividade do servidor. Para isso é necessária a introdução de uma flag de admin (flag '-a') e a introdução de uma password que está armazenada internamente no servidor (ex: ./client quit **-a "password"**).

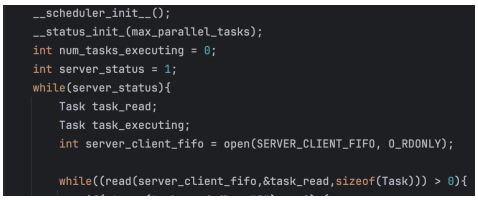
4 Servidor

O servidor é o responsável pela gestão e execução das tarefas e pedidos recebidos pelo cliente. Para isso, é necessário que este leia e escreva no fifo de ligação entre os dois, bem como criar processos filho para a execução de tarefas e contagem do tempo de execução das mesmas. No entanto, este processo não pode ser bloqueado, ou seja, o cliente e a execução de tarefas não podem ficar bloqueadas, ou seja, o servidor tem que ser capaz de lidar com a receção de tarefas pelo fifo e pela ausência de pedidos no fifo.

Para que não haja um bloqueio no servidor nos casos listados em cima, é necessária uma boa gestão de leitura e escritas no fifo. Para isso o servidor possui um ciclo 'while' para leitura constante do fifo, e um 'if', externo ao ciclo anterior, para lidar com a ausência de pedidos.

4.1 Leitura do fifo

Como referido anteriormente, é necessário que o servidor esteja constantemente à procura de pedidos no fifo. Para isso é aplicado o seguinte ciclo:



```
__scheduler_init__();
__status_init_(max_parallel_tasks);
int num_tasks_executing = 0;
int server_status = 1;
while(server_status){
    Task task_read;
    Task task_executing;
    int server_client_fifo = open(SERVER_CLIENT_FIFO, O_RDONLY);

    while((read(server_client_fifo,&task_read,sizeof(Task))) > 0){
```

Figura 5: Ciclo de leitura do fifo

Neste ciclo, enquanto houverem pedidos para serem lidos, estes são guardados numa fase inicial numa variável local designada de 'task_read'. De seguida, esta é adicionada à lista de espera e, se for possível executar a tarefa de imediato (i.e se o número de tarefas em execução no momento não exceder o número máximo passado ao servidor quando este é executado), então remove-se a tarefa da lista de espera a tarefa e escreve-se a mesma na variável 'task_executing', passando-se à execução da mesma através da invocação da função de execução do módulo 'engine'. Esta função varia caso a tarefa seja uma tarefa unitária (flag -u) ou um pipeline (flag -p).

4.2 Execução de tarefas da fila de espera

Caso não existam pedidos no fifo no momento, passa-se ao 'if' externo, referido anteriormente. Neste, se o servidor o permitir, é executada a tarefa com tempo previsto mais pequeno da lista de espera, para que a execução de tarefas não esteja dependente de novos pedidos do cliente.

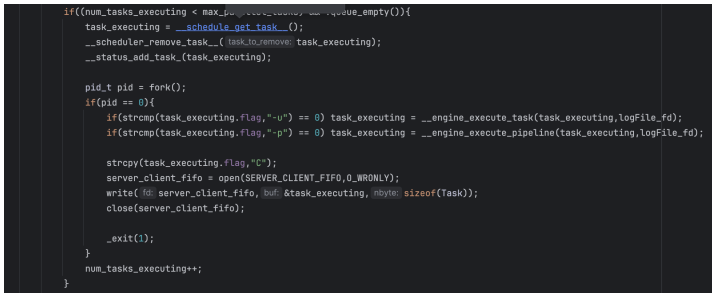


Figura 6: Execução da fila de espera

4.3 Limite de tarefas em paralelo

Quando se executa o servidor, além de se fornecer o 'path' para a diretoria onde serão criados os ficheiros de output, também é fornecido ao servidor um limite de tarefas em paralelo que este pode executar. Este limite é respeitado para que não haja uma sobrecarga no computador no processamento de tarefas.

Quando uma tarefa é executada, o contador de tarefas em paralelo é incrementado. Para que este valor seja atualizado quando uma tarefa é concluída, é escrito no fifo a tarefa executada com uma flag específica (o valor da flag na struct tarefa é alterado para "C"), que informa o servidor que uma tarefa foi concluída. Assim, quando o servidor lê do fifo uma tarefa, se a 'task_read' tiver a flag "C", o contador de tarefas em execução é decrementado.

Figura 7: Verificação de tarefa concluida

5 Engine

O módulo de engine contém as funções necessárias para a execução dos dois tipos de tarefa possíveis: tarefas unitárias e tarefas em pipeline. Daqui resultam as funções `__engine__execute_task` e `__engine__execute_pipeline`.

5.1 `__engine__execute_task`

Esta função é a função responsável por executar uma tarefa uniária e contabilizar o seu tempo de execução, em milissegundos. Para isso é criado um processo filho para a execução do programa pretendido, enquanto que o processo pai espera que o filho acabe a tarefa e conta o tempo que esta demorou a ser feita. É ainda feita a escrita no ficheiro de logs do output do programa e um parse do programa recebido pela função, por forma de uma struct Task para que seja possível a chamada da função 'execvp' no processo filho.

Ainda de se notar que o STDOUT e STDERR do processo filho são direcionados para o ficheiro de output.

Poderá consultar a definição da função na secção de anexos.

5.2 `__engine__execute_pipeline`

A função responsável pela execução do pipeline tem um funcionamento equivalente à função de execução de tarefas em pipeline, no que toca à contabilização, parse do input e redirecionamento do STDERR.

As particularidades desta função começam na execução do pipeline propriamente dito. Para que isso seja possível, é necessário que seja criados tantos processos filhos, quanto o número de programas no pipeline. É ainda necessário que sejam criados pipes anónimos para que os processos filho comuniquem entre si (o número de pipes criados é sempre um a menos do que o número de processos filho criados).

Para que o pipeline funcione corretamente, é necessário uma boa gestão dos direcionamentos dos STDOUT e STDIN dos processos filho. Quando o primeiro filho é criado, este direciona o seu STDOUT para o seu pipe de escrita. Os processo filho intermediários redirecionam o seu STDIN para o pipe de leitura do filho anterior e redirecionam o seu STDOUT para o seu pipe de escrita. Quando chegamos ao último filho da pipeline, basta redirecionar o seu STDIN para o pipe de leitura do filho anterior e o seu STDOUT para o ficheiro de output. De notar que o STDERR de cada um dos filhos também é redirecionado para o ficheiro de output.

Poderá consultar a implementação da função na secção de anexos.

6 Scheduler

O módulo 'scheduler' é o responsável pela gestão da fila de espera do servidor. O servidor implementa uma única política de escalonamento de tarefas: a politica de execução da tarefa com tempo previsto mais rápido.

Para que a política referida anteriormente seja implementada eficazmente, assim que é necessário adicionar uma tarefa à fila de espera, o 'scheduler' faz uma inserção ordenada por tempo de execução previsto, garantindo assim que a tarefa a ser executada (mais rápida) esteja sempre à cabeça da lista. Assim quando uma tarefa é executada, basta ir à cabeça da fila para se obter a tarefa.

7 Status

Para que seja possível ao servidor fornecer ao cliente o seu estado atual, foi necessária a criação deste módulo. Nele encontramos um array auxiliar, com o tamanho do número de tarefas paralelas permitidas, onde serão armazenadas as tarefas em execução no momento. Além do array, encontramos funções que pretendem responder às outras necessidades de um pedido de status: as tarefas já executadas e as tarefas em espera.

7.1 Tarefas em execução

Quando uma tarefa é executada, é armazenada no array referido e quando ela é terminada, é removida do mesmo. Assim, quando um pedido de status chega ao servidor, basta ler o estado atual deste array.

7.2 Tarefas executadas

Assim que uma tarefa é executada, esta é escrita num ficheiro auxiliar na diretoria 'tmp'. Graças a este ficheiro, à chegada de um pedido de status, para se obter o histórico de tarefas executadas anteriormente, basta ler do ficheiro, poupando-se assim espaço na memória principal do computador e não se perdendo este histórico quando o servidor é fechado.

7.3 Tarefas em espera

Por fim, para se obter as tarefas em espera, basta pedir ao módulo 'scheduler' que devolva o estado atual da fila de espera.

Juntando-se estes três pontos numa string, estamos em condições de enviar ao cliente o estado atual do servidor.

8 Testes

Para a validação de todas as funcionalidades do servidor, realizamos uma série de testes que consideramos relevantes. Estes testes podem ser divididos em dois tipos: testes de interação e testes automáticos.

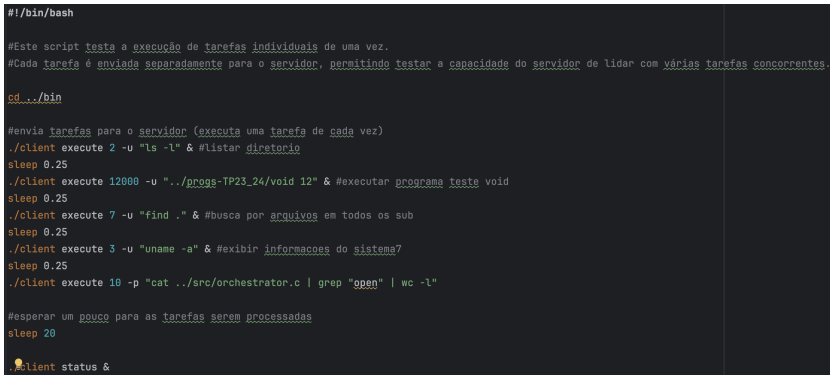
8.1 Testes de interação

Os testes de interação com o servidor foram realizados pelos membros do grupo ao longo do desenvolvimento das diferentes funcionalidades e no final do desenvolvimento do trabalho. Para isso, testámos funcionalidades como o pedido de status, help e quit.

Graças a estes testes, obtivemos conclusões acerca da facilidade de utilização do nosso programa, na ótica do utilizador. O resultados deste testes pode ser visto nas figuras 3 e 4.

8.2 Testes automáticos

No que toca aos testes automáticos, foi criado um script para facilitar o processo de envio de tarefas do cliente para o servidor. O script em questão é o seguinte:



```
#!/bin/bash

#Este script testa a execução de tarefas individuais de uma vez.
#Cada tarefa é enviada separadamente para o servidor, permitindo testar a capacidade do servidor de lidar com várias tarefas concorrentes.

cd ../bin

#envia tarefas para o servidor (executa uma tarefa de cada vez)
./client execute 2 -u "ls -l" & #listar diretorio
sleep 0.25
./client execute 12000 -u "../progs-TP23_24/void 12" & #executar programa teste void
sleep 0.25
./client execute 7 -u "find ." & #busca por arquivos em todos os sub
sleep 0.25
./client execute 3 -u "uname -a" & #exibir informacoes do sistema7
sleep 0.25
./client execute 10 -p "cat ../src/orchestrator.c | grep "open" | wc -l"

#esperar um pouco para as tarefas serem processadas
sleep 20

./client status &
```

Figura 8: Script usado nos testes automáticos

Este script envia a cada 250ms um pedido de tarefa para o servidor. São enviados pedidos de tarefa quer unitários, quer em pipeline, para testar o comportamento do servidor perante o envio das mesmas. Após o envio do pedido, o script espera 20 segundos e, após esse tempo, envia um pedido de status.

Para a execução deste script, primeiro é necessário iniciar o servidor e depois correr, noutra janela do terminal na diretoria scripts, o script test1.sh através do comando ./test1.sh

8.3 Resultado dos testes

Através dos testes referidos nos pontos anteriores, foi possível a deteção de erros durante o desenvolvimento do projeto, que foram solucionados após a sua deteção. Quando estes testes correram todos como esperado, demos o projeto como conluido.

9 Anexos

9.1 Anexo I. Notas adicionais acerca do funcionamento do servidor

- **Nota 1.** O servidor não suporta a receção de pedidos em simultâneo. É possível o envio de pedidos num curto espaço de tempo mas não o envio de pedidos simultaneamente. É por este motivo que o script de testes envia pedidos espaçados por um intervalo de tempo

9.2 Anexo II. Funções da engine

9.2.1 `__engine__execute_task`

```c
Task __engine_execute_task(Task task_executing,int logFile_fd){
    char *aux = strdup( s1: task_executing.program);
    char *token = strtok( str: aux,  sep: " ");
    char *programa = token;
    char *argumentos[11];

    int i = 0;
    while (token != NULL && i < 10) {
        argumentos[i] = token;
        token = strtok( str: NULL,  sep: " ");
        i++;
    }
    argumentos[i] = NULL;

    struct timeval start, end;
    gettimeofday(&start, NULL);

    int status;
    pid_t pid = fork();
    if(pid == 0){
        dup2(logFile_fd, STDOUT_FILENO);
        dup2(logFile_fd,STDERR_FILENO);
        execvp( file: programa,  argv: argumentos);
        perror("Erro ao executar o programa");
        _exit(1);
    }
    waitpid(pid,&status,0);

    gettimeofday(&end, NULL);
    long runtime = (end.tv_sec - start.tv_sec) * 1000 + (end.tv_usec - start.tv_usec) / 1000; // em ms

    task_executing.time = (int) runtime;
    task_executing.pid = getpid();

    char output_message[1024];
    snprintf(output_message, sizeof(output_message), "\nPid: %d (LocalID: %d);Time: %d ms;Arguments: %s\n-----\n\n",
             task_executing.pid,task_executing.id,
             task_executing.time, task_executing.program);

    ssize_t bytes_written = write( fd: logFile_fd,  buf: &output_message,  nbyte: strlen( s: output_message));
    if(bytes_written <= 0){
        perror("Error writing on log file");
        _exit(0);
    }

    free(aux);
    return task_executing;
}
```

Figura 9: Função de execução de uma tarefa unitária

9.2.2 `__engine__execute_pipeline`

```
Task __engine_execute_pipeline(Task task_executing, int logFile_fd) {
    char* pipe_programs[MAX_PIPELINES];
    char* task_program_cpy = strdup( s: task_executing.programa);
    int num_pipelines = 0;

    fillArray( array: pipe_programs, command: task_program_cpy,&num_pipelines);

    int fd[num_pipelines - 1][2];

    struct timeval start, end;
    gettimeofday(&start, NULL);

    for (int i = 0; i < num_pipelines; ++i) {
        char *aux = strdup( s: pipe_programs[i]);
        char *token = strtok( str: aux,  sep: " ");
        char *programa = token;
        char *argumentos[11];

        int j = 0;
        while (token != NULL && j < 10) {
            argumentos[j] = token;
            token = strtok( str: NULL,  sep: " ");
            j++;
        }
        argumentos[j] = NULL;

        if (i == 0) {
            pipe(fd[0]);
            if(fork() == 0) {

                dup2(fd[0][1], STDOUT_FILENO);
                dup2(logFile_fd,STDERR_FILENO);
                close(fd[0][0]);
                close(fd[0][1]);

                execvp( file: programa,  argv: argumentos);

                perror("Exec failed");
                _exit(1);
            }
            close(fd[0][1]);

        } else if (i < num_pipelines - 1) { // meio
            pipe(fd[i]);
            if(fork() == 0) {

                dup2(fd[i - 1][0], STDIN_FILENO);
                dup2(logFile_fd,STDERR_FILENO);
                close(fd[i - 1][0]);

                dup2(fd[i][1], STDOUT_FILENO);
                close(fd[i][0]);
                close(fd[i][1]);

                execvp( file: programa,  argv: argumentos);

                perror("Exec failed");
                _exit(1);
            }
            close(fd[i - 1][0]);
            close(fd[i][1]);


        } else if(i == num_pipelines - 1){ // ultimo
            if(fork() == 0) {

                dup2(fd[i - 1][0], STDIN_FILENO);
                dup2(logFile_fd,STDOUT_FILENO);
                dup2(logFile_fd,STDERR_FILENO);
                close(fd[i - 1][1]);
                close(fd[i - 1][0]);

                execvp( file: programa,  argv: argumentos);

                perror("Exec failed");
                _exit(1);
            }
            close(fd[i - 1][0]);
        }
    }

    for (int i = 0; i < num_pipelines; i++) {
        wait(NULL);
    }

    gettimeofday(&end, NULL);
    long runtime = (end.tv_sec - start.tv_sec) * 1000 + (end.tv_usec - start.tv_usec) / 1000; // em ms

    task_executing.time = (int) runtime;
    task_executing.pid = getpid();

    char output_message[1024];
    snprintf(output_message, sizeof(output_message), "\nPid: %d (LocalID: %d);Time: %d ms;Arguments: %s\n--------------------\n\n",
            task_executing.pid, task_executing.id,
            task_executing.time, task_executing.programa);

    ssize_t bytes_written = write( fd: logFile_fd,  buf: &output_message,  nbyte: strlen( s: output_message));
    if (bytes_written <= 0) {
        perror("Error writing on log file");
        _exit(0);
    }

    return task_executing;
}
```

Figura 10: Função de execução de uma tarefa em pipeline