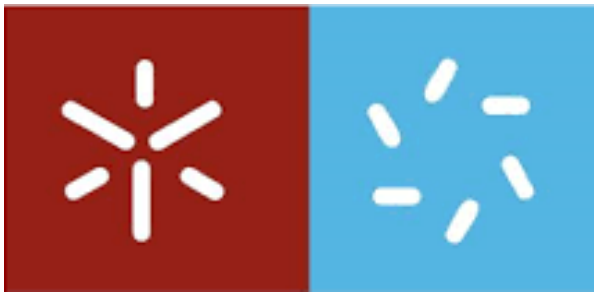

Orquestrador de Tarefas

Nuno Machado A68702 LCC
Tomás Oliveira A100088 LCC

Grupo 40

7 de junho de 2024

Conteúdo

1 Introdução

No âmbito da unidade curricular de Sistemas Operativos do curso de Licenciatura em Ciências da Computação foi-nos proposto como trabalho prático a implementação de um serviço de orquestração (isto é, execução e escalonamento) de tarefas num computador.
Os utilizadores utilizam um programa "cliente"para submeter ao servidor a intenção de executar uma tarefa com informações sobre a duração necessária e qual a tarefa a executar.
Cada tarefa garante um identificador único que é transmitido ao programa cliente mal o servidor recebe a informação da mesma.
O servidor é responsável por escalonar e executar as tarefas solicitadas pelos utilizadores.
A informação produzida pelas tarefas para o standard output ou o devido erro devem ser redireccionados pelo servidor para um ficheiro cujo nome corresponde ao identificador da tarefa.
Através do programa cliente, será possível para os utilizadores a consulta do servidor para saberem quais as tarefas que se encontram em execução, quais estão em espera para executar e quais já terminaram a sua execução.
A seguir são expostas as decisões tomadas pelo grupo, bem como os motivos das mesmas e as suas vantagens/desvantagens. Também se fará uma explicação breve da implementação do projeto, mais centrado nas partes que o grupo considerou necessidade de haver destaque face à importância do seu papel na compreensão do funcionamento do software desenvolvido.

2 Arquitectura

O nosso projeto segue o modelo cliente/servidor, como mencionado no capítulo da introdução, pelo que se pode verificar dois programas:

1. **cliente**: responsável pela execução de comandos de bash e consultas das informações/estados das tarefas;
2. **orchestrator**: responsável por escalonar e executar tarefas solicitadas pelos utilizadores.

2.1 Programa Client

O programa client permite visualizar e executar tarefas propostas por um utilizador, lendo os seus argumentos, convertendo-os posteriormente para um program enviando-o para o servidor. Quando esse FIFO fechar, significa então que o pedido terminou a sua execução e o cliente termina normalmente.
O cliente é um processo independente do servidor, é possível ter vários clientes ligados ao servidor simultaneamnete. O cliente deve receber o seu pedido através de argumentos, processa-los para o formato utilizado pelo servidor, criar um FIFO para o qual o servidor lê desse mesmo canal até ao pedido ser terminado, imprimindo a resposta do cliente e do servidor para um ficheiro .txt.

2.2 Programa Orchestrator

O servidor começa por criar o FIFO. O programa espera conseguir ler algo do FIFO de leitura, isto é, quando o programa client escreve algo no FIFO de escrita). Suponha-se ter 2 comandos recebidos pelo orchestrator do seu FIFO de leitura:

1. execute
2. status

Falaremos deles mais à frente no relatório. O programa utiliza uma estrutura de dados para armazenar informações das tarefas, como o seu identificar único (PID), o nome do programa, o tempo de execução, entre outros. O programa utiliza um loop para aguardar por comandos vindos do pipe de leitura. Quando um comando é recebido, o programa executa a ação correspondente ao comando solicitado.

O servidor é constituída por três componentes, cada uma corresponde a um processo filho do processo principal do servidor. Será explicada cada compenente no capítulo seguinte de "Etapas do Grupo".

3 Dificuldades e Etapas do Grupo

3.1 Etapas do Grupo

3.1.1 Pedidos

De forma a facilitar o processamento do pedido no lado do servidor, foi implementada uma estrutura "Program"que armazena a informação de um pedido, como se pode ver aqui:

```
typedef struct {
    pid_t pid;    // process pid
    char program[300];  // program to execute
    int time;     // execution time
    char type[3];
    long sec;
    long ms;
} program;
```

3.1.2 Escalonamento

Avançamos para o escalonamento que nos gerou dificuldade na forma como íriamos implementar, visto estarmos na dúvida qual a melhor política. Acabamos por implementar FCFS, First-Come-First-Served, ou seja, a primeira tarefa enviada é processada primeiro, depois a segunda e assim por diante, pois acreditamos que seria a mais fácil de entender/implementar. Utilizamos uma fila de tarefas (taskQueue), que é uma estrutura de dados que armaneza tarefas pela ordem que são recebidos do cliente.

3.1.3 Comandos

O pedido de status é outro comando a par do comando execute que implementamos no nosso projeto.

1. **execute**: O pedido de execute permite ao utilizador do cliente enviar um pedido de execução de uma tarefa indicando o respetivo tempo de execução, o nome do programa e os seus argumentos (caso existam). Para realizar esta operação, o cliente envia um pedido ao servidor através de um FIFO. O servidor, por sua vez, recebe este pedido e processa-o. Após o processamento desse pedido, o servidor envia a resposta ao cliente através do pipe com nome. O cliente recebe essa resposta e apresenta-a ao utilizador gerando um ficheiro .txt com as informações da tarefa.

2. **status**: O pedido de status permite ao utilizador do cliente solicitar informações sobre os programas que estão em execução. Á semelhança do comando execute, o cliente envia um pedido ao servidor através de um FIFO. O servidor, por sua vez, recebe este pedido e processa-o. Para obter as informações necessárias, a ideia seria o servidor verificar o estado de cada processo em execução e devolver uma lista com as informações da respetiva tarefa, o PID (identificador único), nome do programa e o seu tempo de execução. Após reunidas essas informações, o servidor envia a resposta ao cliente através do pipe com nome. O cliente recebe essa resposta e apresenta-a ao utilizador gerando um ficheiro .txt com essas mesmas informações.

3.2 Dificuldades

A primeira dificuldade com que o grupo se deparou foi com a comunicação entre o orchestrator e o client. Estavamos a criar o pipe com nome mas apenas existia comunicação do client com o orchestrator mas o contrário não se verifica. Conseguimos perceber que seria necessário inciar 1 pipe com nome, FIFO, uma que comunicava do client para o orchestrator e vice-versa.
A segunda dificuldade com que o grupo se deparou foi que inserindo uma tarefa, o servidor repetia infinitamente a mesma obrigando a abortar a ligação ao servidor. Corrigimos implementando novamente um pedido de leitura do orchestrator ao client ficando o mesmo a aguardar pela próxima tarefa.
Na etapa seguinte decidimos que teríamos de colocar o orchestrator a executar as tarefas guardado as informações das mesmas num ficheiro .txt e a dificuldade com que o grupo se deparou foi com a parte posterior à execução pois não estaríamos a conseguir fazer com que ficasse a informação guardada no ficheiro como era pretendido. Acabamos por resolver essa dificuldade e ficou implementado.
A implementação do status foi algo que não conseguimos completar com sucesso, fica um subcapitulo a falar do que seria pretendido e que tentamos implementar. Avançamos de seguida para a implementação das pipelines que infelizmente também não tivemos sucesso na sua implementação. Fica o código da tentativa no nosso projeto.

4 Avaliação de Políticas de Escalonamento

FCFS (First-Come, First-Served):

Prós É fácil de implementar e garante uma distribuição justa do tempo da CPU entre os processos. Em cenários onde os processos têm tempos de execução semelhantes e não há preocupação com tempo de resposta, FCFS pode funcionar bem.

Contras: Pode causar inanição de processos mais curtos se processos longos ocuparem a CPU por um longo período. Isso pode levar a um aumento no tempo médio de espera e ao fenômeno conhecido como "efeto da fila do supermercado", onde um único processo longo pode atrasar todos os outros processos na fila.

SJF (Shortest Job First):

Prós: Minimiza o tempo médio de espera dos processos na fila de prontos, dando prioridade aos processos mais curtos. Isso pode levar a uma maior eficiência geral do sistema, especialmente em cenários onde há uma mistura de processos longos e curtos.

Contras: Pode causar problemas de preempção, onde processos curtos podem ser interrompidos por processos mais longos que chegam depois. Além disso, para implementar o SJF de forma ideal, o sistema precisa conhecer os tempos de execução de todos os processos com antecedência, o que pode não ser viável em todas as situações.

Exemplo:

o comando ls -l e pode demorar consideravelmente quando há um grande número de arquivos no diretório especificado o que ira atrasar progamas mais curtos o que implica que o nosso escalonamento FCFS não é o mais adequado para estas situações

5 Conclusão

Conseguimos concluir a 1ª etapa do projeto na totalidade, todas as funcionalidades que foram propostas foram concluídas com sucesso à exceção do comando status pelo que neste momento temos o nosso programa client a comunicar com o servidor orchestrator à qual é possível executar tarefas com escalonamento guardando as informações das tarefas executadas, em execução e por executar num ficheiro .txt. Em relação à 2ª etapa, iniciamos o encadeamento de programas, os pipelines, embora não esteja totalmente funcional como pretenderíamos, acreditamos que falta muito pouco para o seu sucesso. Naturalmente há espaço a melhorias, e tarefas do enunciado que ficaram em falta, mas ainda assim, o grupo considera o seu projeto satisfatório.

Referências

A Código Final

A.1 Client

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include "defs.h"

#define server_to_client "/tmp/server_to_client"
#define client_to_server "/tmp/client_to_server"

int main(int argc, char *argv[]) {
    if (argc < 4) {
        printf("Uso: %s execute <time> -u '<prog-a> [args]'
          \n", argv[0]);
    }




    // Popula a struct
    program task;
    task.time = atoi(argv[2]);
    strncpy(task.type, argv[3], sizeof(task.type));
    task.pid = getpid();
    strcpy(task.program, argv[4]);

    // Abrir o FIFO do servidor para escrita
    int fd_client_to_server = open(client_to_server,
    O_WRONLY);
    if (fd_client_to_server < 0) {
        perror("Erro ao abrir o FIFO de escrita do servidor
        ");
        return -1;
    }


    // Escrever a struct no FIFO
    if (write(fd_client_to_server, &task, sizeof(task)) <
    0) {
        perror("Erro ao escrever no FIFO do servidor");
        return -1;
    }
```

```
    // Fechar o FIFO de escrita do servidor
    close(fd_client_to_server);

    int fd_server_to_client= open(server_to_client ,
    O_RDONLY);
    if (fd_server_to_client < 0) {
        perror("Erro ao abrir o FIFO de escrita do servidor
        ");
        return -1;
    }

    char task_id[256];
    if(read(fd_server_to_client , task_id , sizeof(task_id))
    ==-1){
        perror("erro id");
        close(server_to_client);
        return -1;
    }
    close(server_to_client);
    printf("Task ID: %s\n", task_id);

    printf("Tarefa submetida com sucesso.\n");

    return 0;
}
```

A.2 Struct

```c
#include <time.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/time.h>
#include <stdio.h>

typedef struct{
    pid_t pid; // process pid
    char program[300];  //program to execute
    int time;
    char type[3];
    long sec;
    long ms;
} program;
```

A.3 Orchestrator

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <sys/wait.h>
#include "defs.h"
#include <errno.h>

#define server_to_client "/tmp/server_to_client"
#define client_to_server "/tmp/client_to_server"
#define MAX_TASKS 10

typedef struct
{
    program tasks[MAX_TASKS];
    int count;
} TaskQueue;

TaskQueue task_queue; // fila de tarefas

char **parse(char *string)
{
    char *token = strtok(string, " ");
    char **strings = NULL;
    int i = 0, num_strings = 1;
    while (token != NULL)
    {
        strings = (char **)realloc(strings, num_strings *
        sizeof(char *));
        strings[i] = token;
        token = strtok(NULL, " ");
        i++, num_strings++;
    }
    return strings;
}

int parser(char **exec_args, char *str) {
    char *comando = strdup(str);
    if (!comando) {
        perror("Failed to allocate memory for comando");
        exit(EXIT_FAILURE);
    }
    int i = 0;
    char *string = strsep(&comando, " ");
```

```
    while (string) {
        if (strcmp("", string) != 0) {
            exec_args[i++] = string;
        }
        string = strsep(&comando, " ");
    }
    exec_args[i] = NULL;
    return i;
}

int pipeline(char **commands, int nc) {
    int pipefds[2 * (nc - 1)]; // array pra guardar os
     descritores do pai
    int pid;
    int status;

    // Create pipes
    for (int i = 0; i < nc - 1; i++) {
        if (pipe(pipefds + i*2) < 0) {
            perror("Failed to create pipe");
            exit(EXIT_FAILURE);
        }
    }

    for (int i = 0; i < nc; i++) {
        pid = fork();
        if (pid == 0) { // filho
            // se n for o ultimo comando rederecionar o
             input para o proximo pipe
            if (i > 0) {
                if (dup2(pipefds[(i - 1) * 2], STDIN_FILENO
                 ) < 0) {
                    perror("dup2 input");
                    _exit(EXIT_FAILURE);
                }
            }
            // se n for o ultimo comando redirecionar para
             o proximo pipe
            if (i < nc - 1) {
                if (dup2(pipefds[i * 2 + 1], STDOUT_FILENO)
                  < 0) {
                    perror("dup2 output");
                    _exit(EXIT_FAILURE);
                }
            }

            // Cfechar todos os pipes fds
            for (int j = 0; j < 2 * (nc - 1); j++) {
```

```
                close(pipefds[j]);
            }

            // Parse dos argumentos comando
            char *exec_args[20]; // aumentar tamanho se
             necessaro
            parser(exec_args, commands[i]);
            execvp(exec_args[0], exec_args);
            perror("execvp failed");
            _exit(EXIT_FAILURE);
        } else if (pid < 0) {
            perror("Failed to fork");
            exit(EXIT_FAILURE);
        }
    }

    // fechar todos os pipes no pai
    for (int i = 0; i < 2 * (nc - 1); i++) {
        close(pipefds[i]);
    }


    while ((pid = wait(&status)) > 0) {
    }

    return status;
}
int parse_pipeline(char **args, char *cmd_str) {
    char *token;
    token = strtok(cmd_str, "|");
    int i = 0;
    while (token != NULL) {
        args[i] = token;
        token = strtok(NULL, "|");
        i++;
    }
    return i;
}

// Fun   o para adicionar uma tarefa    fila
void add_task(program task)
{
    if (task_queue.count < MAX_TASKS)
    {
        task_queue.tasks[task_queue.count++] = task;
    }
    else
    {
```

```
        printf("A fila de tarefas est  cheia.\n");
    }
}

// Fun  o para remover e retornar a pr xima tarefa da
 fila
program pop_task()
{
    if (task_queue.count > 0)
    {
        program task = task_queue.tasks[0];
        for (int i = 0; i < task_queue.count - 1; i++)
        {
            task_queue.tasks[i] = task_queue.tasks[i + 1];
        }
        task_queue.count--;
        return task;
    }
    else
    {
        program empty_task; // Retorna uma tarefa vazia se
         a fila estiver vazia
        empty_task.time = -1;
        return empty_task;
    }
}

int mysystem(program p)
{

    // Parsing do comando
    char *comando;
    comando = strdup(p.program);
    int i = 0;
    char *string;
    char *exec_args[20];
    string = strsep(&comando, " ");
    while (string != NULL)
    {
        exec_args[i] = string;
        string = strsep(&comando, " ");
        i++;
    }
    exec_args[i] = NULL;

    int status;

    pid_t pid = fork();
```

```
    // Executar o comando
    if (pid == 0)
    {
        execvp(exec_args[0], exec_args);
        perror("Erro ao executar comando");
        _exit(EXIT_FAILURE);
    }
    else
    {
        wait(&status);
        p.pid = pid;
    }
}

int main()
{

    int outfd = open("saida.txt", O_CREAT | O_APPEND|
    O_WRONLY, 0666);
    dup2(outfd, 1); // Redireciona stdout para o arquivo
     saida.txt

    int errfd = open("erros.txt", O_CREAT | O_APPEND |
    O_WRONLY,0666);
    dup2(errfd,2); //aponta o error para a pasta de erro.
     txt



    // Inicializar a fila de tarefas
    task_queue.count = 0;

    program task;

    // Criar o FIFO principal (MainFifo) se n o existir
    if (mkfifo(server_to_client, 0666) == -1 && errno !=
    EEXIST)
    {
        perror("erro no server to client");
        exit(EXIT_FAILURE);
    }

    if (mkfifo(client_to_server, 0666) == -1 && errno !=
    EEXIST)
    {
        perror("erro no client to server");
        exit(EXIT_FAILURE);
    }
```

```
// Abrir o FIFO do cliente para leitura
int fd_client_to_server = open(client_to_server,
O_RDONLY);
if (fd_client_to_server < 0)
{
    perror("Erro ao abrir o FIFO de entrada do cliente"
    );
    exit(EXIT_FAILURE);
}



// Loop para receber e processar as tarefas
size_t bytes_read;
int server_to_client_fd = -1;

while (1)
{
    ssize_t bytes_read = read(fd_client_to_server, &
    task, sizeof(task));
    if(bytes_read >0){
    if (bytes_read < 0)
    {
        perror("Erro ao ler do FIFO do servidor");
        exit(EXIT_FAILURE);
    }

    if (server_to_client_fd == -1)
    {
        server_to_client_fd = open(server_to_client,
        O_WRONLY);
        if (server_to_client_fd == -1)
        {
            perror("erro a abrir o server to client");
            exit(EXIT_FAILURE);
        }
    }
    // Adicionar a tarefa recebida    fila
    add_task(task);

    // Executar as tarefas da fila
    while (task_queue.count > 0)
    {
        program next_task = pop_task();

        if (strcmp(next_task.type, "-u") == 0)
```

```
        {
            struct timeval time;
            gettimeofday(&time, NULL);
            task.ms = time.tv_usec;
            task.sec = time.tv_sec;

            mysystem(next_task);

            gettimeofday(&time, NULL);
            long ms = ((task.sec - time.tv_sec) * 1000)
              + ((task.ms - time.tv_usec) / 1000);
        }
        else if (strcmp(next_task.type, "-p") == 0)
        {
            struct timeval time;
            gettimeofday(&time, NULL);
            task.ms = time.tv_usec;
            task.sec = time.tv_sec;

            char *args[100];

            int tamanho = parse_pipeline(args,
             next_task.program);

            fflush(stdout);
            pipeline(args, tamanho);

            gettimeofday(&time, NULL);
            long ms = ((task.sec - time.tv_sec) * 1000)
              + ((task.ms - time.tv_usec) / 1000);
        }
        // Enviar mensagem de confirma  o para o
         cliente
        char response_message[20];
        sprintf(response_message, "%d", next_task.pid);
        ssize_t bytes_written = write(
         server_to_client_fd, response_message, strlen(
         response_message) + 1);
        if (bytes_written < 0)
        {
            perror("Erro ao escrever no FIFO de sa da
             do servidor");
            exit(EXIT_FAILURE);
        }
    }

    // bytes_read = read(fd_client_to_server, &task,
     sizeof(task));
```

```
            // Fechar o FIFO do cliente ap s processar cada
             tarefa
            // close(fd_server_to_client);
            }
        }

        // Fechar o FIFO do cliente ap s processar todas as
         tarefas
        close(server_to_client_fd);
        close(fd_client_to_server);

        // Remover o FIFO principal (MainFifo)
        unlink(server_to_client);
        unlink(client_to_server);

        return 0;
}
```