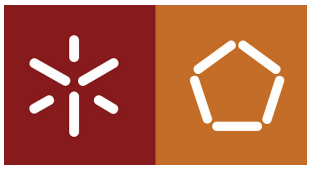
Universidade do Minho

Departamento de Informática



licenciatura em engenharia informática

Sistemas Operativos - Orquestrador de Tarefas

Grupo 8

Luís de Castro Rodrigues Caetano A100893
Simão Pedro Ferreira Antunes A100597
Tiago Esteves Rodrigues A84276

Maio de 2024

Conteúdo

1 Introdução

O seguinte relatório descreve a nossa implementação de um serviço de orquestração de tarefas num computador, realizado no âmbito da UC de Sistemas Operativos.

O trabalho consiste em dois programas, o *Client* e o *Orchestrator*. Os utilizadores usam o *Client* para submeter ao *Orchestrator* uma tarefa que desejam executar, dando uma indicação da duração em milissegundos que necessitam para a mesma. A cada tarefa é atribuído um identificador único que é transmitido ao cliente mal o servidor recebe a mesma. O *Orchestrator* é responsável por escalonar e executar as tarefas dos vários utilizadores, sendo o seu output armazenado num ficheiro cujo nome corresponde ao identificador da tarefa. Através do programa *Client*, os utilizadores podem ainda consultar o servidor para saberem quais as tarefas em execução ou em espera.

2 Util

2.1 Struct Task

A estrutura *task* foi criada para armazenar a informação de uma tarefa e para facilitar a comunicação entre o *Client* e o *Orchestrator*. É composta por:

- Um pid t, equivalente a um signed int, que serve para guardar o pid do processo que submeteu o pedido para ser usado como identificador único.
- Um int que representa o tempo de execução da tarefa em milissegundos.
- Um array de chars de tamanho fixo que achamos adequado para guardar os nomes dos comandos (execute -u, execute -p, status).
- Um array de chars de tamanho fixo que achamos adequado para guardar os nomes dos programas a serem executados.

2.2 Struct Entry

A estrutura *entry* foi criada para simplificar o armazenamento de programas já executados num ficheiro binário. É composta por:

- Um pid t, equivalente a um signed int, que serve para guardar o pid do processo que submeteu o pedido para ser usado como identificador único.
- Um long int que representa o tempo de execução real da tarefa em milissegundos.
- Um array de chars de tamanho fixo que achamos adequado para guardar os nomes dos programas a serem executados.

2.3 Funções auxiliares

Para além destas estrtuturas, o módulo util.h implementa também duas funções cruciais para o funcionamento do *Orchestrator*, a *execute_u* e a *execute_p*.
A função *execute_u* é responsável pela execução de programas únicos. Recebe como argumento um char* que é o programa a executar e outro char* que é o nome do ficheiro onde vai ser guardado o output. Primeiramente ela faz uma cópia do programa original para não alterar o char* original e a seguir divide-o num array de strings, usando o carater como divisor. Após dividir o programa, ela redireciona o standard output e standard error para o ficheiro passado como argumento através de um *dup2* e cria um processo filho através de um *fork()* para executar o mesmo com um *execvp()*.

```
int execute_u(char* prog, char* file) {
    char **array = malloc(MAXARGS * sizeof(char*));
    char *prog_copy = strdup(prog);

    (...)

    while ((temp = strsep(&prog_copy, " ")) != NULL) {
        array[i++] = temp;
    }

    array[i] = NULL;
    free(prog_copy);

    (...)

    dup2(fd, STDOUT_FILENO);
```

```
        dup2(fd, STDERR_FILENO);

        if (fork() == 0) {
            execvp(array[0], array);
            _exit(1);
        }

        (...)
    }
```

A função *execute_p* é responsável pela execução de programas em pipeline. Tal como a *execute_u* recebe como argumento um char* que é o programa a executar e outro char* que é o nome do ficheiro onde vai ser guardado o output. Contudo, em vez de dividir o programa num array de strings ela divide numa matriz. Assim, numa primeira vez divide o programa num array de strings, usando o carater "como divisor, e para cada array resultante divide o mesmo usando o carater como divisor. Ao contrário da *execute_u* que basta apenas chamar um *execvp()* para executar o programa, é necessário percorrer todas as linhas da matriz e para cada uma delas chamar a *execvp()*. Para isso é necessário criar N - 1 pipes (sendo N o número de linhas da matriz) para poder redirecionar o standard input para a linha anterior da matriz e o standard ouput para a próxima linha da matriz. No caso do primeiro do primeiro programa não é necessário redirecionar o standard input e no caso do último é necessário redirecionar o standard ouput para o ficheiro passado como argumento.

```
    int execute_p(char *prog, char* file) {
        char **array = parsePipes(prog);
        char ***matrix = parseArgs(array);

        (...)

        int pipes[len - 1][2];

        (...)

        dup2(fd, STDOUT_FILENO);
        dup2(fd, STDERR_FILENO);

        for (int i = 0; i < len; i++) {
            if (i == 0) {
                pipe(pipes[i]);
                if (fork() == 0) {
                    close(pipes[i][0]);
                    dup2(pipes[i][1], STDOUT_FILENO);
                    close(pipes[i][1]);
                    execvp(matrix[i][0], matrix[i]);
                    _exit(1);
                } else {
                    close(pipes[i][1]);
                    wait(&res);
                    if (WEXITSTATUS(res)) {
                        return -1;
                    }
                }
            } else if (i == len - 1) {
                if (fork() == 0) {
                    dup2(pipes[i - 1][0], STDIN_FILENO);
                    close(pipes[i - 1][0]);
                    execvp(matrix[i][0], matrix[i]);
                    _exit(1);
                } else {
```

```
                close(pipes[i - 1][0]);
                wait(&res);
                if (WEXITSTATUS(res)) {
                    return -1;
                }
            }
        } else {
            pipe(pipes[i]);
            if (fork() == 0) {
                close(pipes[i][0]);
                dup2(pipes[i - 1][0], STDIN_FILENO);
                close(pipes[i - 1][0]);
                dup2(pipes[i][1], STDOUT_FILENO);
                close(pipes[i][1]);
                execvp(matrix[i][0], matrix[i]);
                _exit(1);
            } else {
                close(pipes[i - 1][0]);
                close(pipes[i][1]);
                wait(&res);
                if (WEXITSTATUS(res)) {
                    return -1;
                }
            }
        }
    }

    (...)
}
```

3 Client

Para os utilizadores serem capazes de submeter tarefas ao servidor para ele executar, é necessário que o *Client* e o *Orchestrator* comuniquem entre si. Para tal decidimos usar um *pipe com nome* (stats), o qual é criado tanto no *Client* como no *Orchestrator* caso ele já não exista.

3.1 Envio de tarefas

Como foi referido anteriormente, decidimos criar uma estrutura *task* para facilitar a comunicação entre os dois programas, sendo esta utilizada para o envio de uma tarefa. Assim, criamos uma função *execute()* que, dado um int e dois char* passados como argumento, preenche a estrutura *task* com a informação relevante para a execução da tarefa. Primeiramente abre o pipe stats no modo de escrita e após preencher a estrutura escreve-a no mesmo. A seguir, através do seu identificador, cria um novo pipe com nome para recber a resposta do servidor, imprimindo-a através de um *write()*.

```
void execute(int time, char* prog, char* cmd) {
    int fd1 = open("tmp/stats", O_WRONLY);

    (...)

    Task t;
    t.pid = pid;
    t.time = time;
    strcpy(t.cmd, cmd);
    strcpy(t.prog, prog);

    (...)

    if (mkfifo(s_pid, 0777) == -1) {
        perror("Could not create FIFO\n");
        return;
    }

    write(fd1, &t, sizeof(t));
    close(fd1);

    int fd2 = open(s_pid, O_RDONLY);

    (...)

    read(fd2, &task_id, sizeof(int));

    (...)

    write(STDOUT_FILENO, s_task, bytes);

    close(fd2);
    unlink(s_pid);
}
```

3.2 Consulta de tarefas

A consulta de tarefas terminadas e em espera segue o mesmo molde da função acima, com pequenas alterações. É também enviada uma estrutura *task* só que esta agora possui um tempo de execução igual a 0 e o comando deixa de ser um "execute -u/-p", mas sim um "status". Como resposta a este pedido, o servidor envia uma série de estruturas *entry*, onde as que possuem tempo de execução igual a 0 significa que ainda não foram realizadas.

4 Orchestrator

O programa *Orchestrator* foi feito com o intuito de permitir o máximo de concorrência sem gerar casos de deadlock ou causar comportamento incorreto das diferentes funções do programa devido a essa concorrência.

Para tal o *Orchestrator* começa, tal como o *Client*, por criar um pipe com nome caso já não exista, para receber as tarefas submetidas por processos a correr o programa *Client*. De seguida abrimos e guardamos o descritor do pipe em modo leitura. Repetimos o procediemnto para o modo escrita de garantir que o *Orchestrator* bloqueie e por isso não necessita uma espera ative ou de ter um *Client* já em execução.

Com isto pronto, cria-se um pipe anónimo e utilizamos a função *fork()* para criar um filho dedicado ao escalonamento de tarefas.

4.1 Escalonamento de tarefas

Através do pipe anónimo criado anteriormente, o processo recebe as várias tarefas que tem para executar, atuando como uma fila de espera. De maneira a utilizar o tempo de execução indicado pelo utilizador, tomamos a decisão de implementar uma adaptação do algoritmo *Round-Robin*.

Deste modo, definimos um *Quantum* que é subtraído ao tempo de execução de cada tarefa até este ser 0. Enquanto for diferente de 0 a tarefa é escrita de novo no pipe anónimo de maneira a ir para o fim da fila de espera.

Posto isto, o processo filho cria N processos filhos (sendo N o número máximo de tarefas em paralelo) que partilham o mesmo pipe anónimo, de maneira a atuar como outra fila de espera. Assim, quando o tempo de execução atinge 0, o processo escreve a tarefa no pipe anónimo partilhado e o primeiro processo filho a ler a mesma executa-a.

```
void handler(int pipes[2], char* folder, int max_parallel_tasks) {

    (...)

    int queue_pipe[2];

    (...)

    for (int i = 0; i < max_parallel_tasks; i++) {
        if (fork() == 0) {
            close(queue_pipe[1]);
            executer(queue_pipe[0], writer_pipe[1], folder);
        }
    }

    close(queue_pipe[0]);

    (...)

    int res;
    Task t;

    while ((res = read(pipes[0], &t, sizeof(t))) > 0) {
        t.time -= QUANTUM;
        if (t.time <= 0) write(queue_pipe[1], &t, sizeof(t));
        else write(pipes[1], &t, sizeof(t));
    }
}
```

4.2 Tarefas terminadas

Para armazenar as tarefas que já foram executadas criamos um processo filho responsável pela escrita das mesmas num ficheiro binário. Optamos por esta abordagem para evitar estar sempre a abrir e fechar o ficheiro.

```
void fWriter(int pipe) {
    int fd = open("history.bin", O_CREAT | O_APPEND | O_WRONLY, 0777);

    (...)

    int res;
    Entry e;

    while ((res = read(pipe, &e, sizeof(e))) > 0) {
        write(fd, &e, res);
    }
}
```

4.3 Tarefas em espera

Para lidar com a resposta das tarefas em espera no pedido status, fizemos uso da biblioteca *glib* e implementamos uma hashtable. Assim, quando chega uma tarefa ao *Orchestrator* ele verifica se a mesma já se encontra na hashtable. Caso esta não se encontre lá, significa que foi um *Client* a escrevê-la no pipe com nome e armazena uma estrutura *entry* usando o identificador único da tarefa como chave. Caso já se encontre lá, significa que foi o processo filho que a executou a escrevê-la no pipe com nome e remove-a. Deste modo, quando um utilizador faz um pedido de status basta apenas percorrer a hashtbale e enviar todas as entradas lá presentes.

```
void executer(int pipe1, int pipe2, char* folder) {
    int res;
    Task t;

    while ((res = read(pipe1, &t, sizeof(t))) > 0) {

        (...)

        gettimeofday(&start, NULL);
        if (!strcmp(t.cmd, "execute -u")) execute_u(t.prog, out_file);
        else execute_p(t.prog, out_file);
        gettimeofday(&end, NULL);

        (...)

        int fd = open("tmp/stats", O_WRONLY);

        (...)

        write(fd, &t, sizeof(t));
    }
}
```

```
int main(int argc, char ** argv) {

    (...)

    GHashTable *pending_tasks = g_hash_table_new(g_direct_hash, g_direct_equal);

    int res;
    Task t;

    while ((res = read(fd1, &t, sizeof(t))) > 0) {

        if (g_hash_table_contains(pending_tasks, GINT_TO_POINTER(t.pid))) {
            g_hash_table_remove(pending_tasks, GINT_TO_POINTER(t.pid));
            continue;
        }

        (...)
    }

    (...)
}
```

5 Conclusão

Em conclusão, com a realização deste trabalho foram aprofundados vários conceitos de sistemas operativos, tais como: pipes com nome, pipes anónimos, dups e outros temas abordados nas aulas da disciplina.