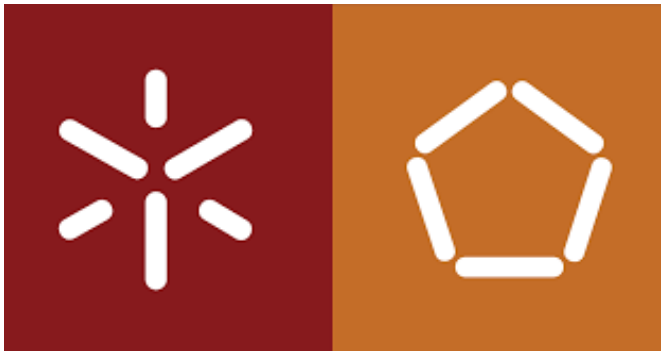


Universidade Do Minho
Engenharia Informática

Trabalho Prático

Sistemas Operativos
2023/2024

Carlos Alberto Fernandes Dias Da Silva a93199

Tiago Alexandre Ferreira Silva a93182

Tiago Pinheiro da Silva a93285

Relatório Grupo11 data de entrega: **07/05/2024**

Índice

Introdução

Este relatório descreve a nossa implementação de um Orquestrador de Tarefas no âmbito da Unidade Curricular de Sistemas Operativos do curso de Engenharia Informática da Universidade do Minho.

Formato das mensagens

```c
typedef struct msg{
    char program[300];
    int pid;
    int time;
    int id;
    int type;
    struct timeval start_time;
    struct timeval end_time;
} Msg;
```

Toda a comunicação que se dá na nossa solução passa pelo uso desta struct *msg*. Usar sempre a mesma estrutura para comunicar permite ter um controlo mais fácil no momento de ler e escrever num pipe com nome, que foi o mecanismo de comunicação usado no nosso programa.

Na maior parte das vezes a mensagem representa um pedido de execução em que:

program: O programa que se pretende executar;
pid: O pid do processo que faz um pedido, seja ele de execução ou status, com o qual será possível aceder ao *FIFO* dedicado para enviar uma resposta;
time: Parâmetro "tempo estimado de execução" que será usado na política de escalonamento;
id: O id do pedido;
type: Parâmetro que permite identificar o tipo de mensagem e agir consoante o mesmo;

start_time e ***end_time***: Campos que a permitirão obter o tempo real de execução de um pedido.

Ao longo deste relatório, entrar-se-á em mais detalhe sobre a composição das mensagens a cada momento do programa.

Estrutura geral

Os processos comunicam entre si através de pipes com nome. O servidor quando inicia cria o seu pipe *fifo_server*. Este nome é de conhecimento geral e assume-se que qualquer cliente que pretenda efetuar um pedido saiba que é através desse canal que pode contactar o servidor.

No momento de efetuar um pedido, o cliente começa por criar um FIFO destinado a receber resposta do servidor. O nome desse FIFO é derivado do PID do processo do cliente, o que garante a unicidade do nome. Como já vimos, o PID em questão é enviado na mensagem para que o servidor saiba para onde é que a resposta deve ser enviada, seja ela apenas o ID da task ou a resposta a um *status*.

Cliente

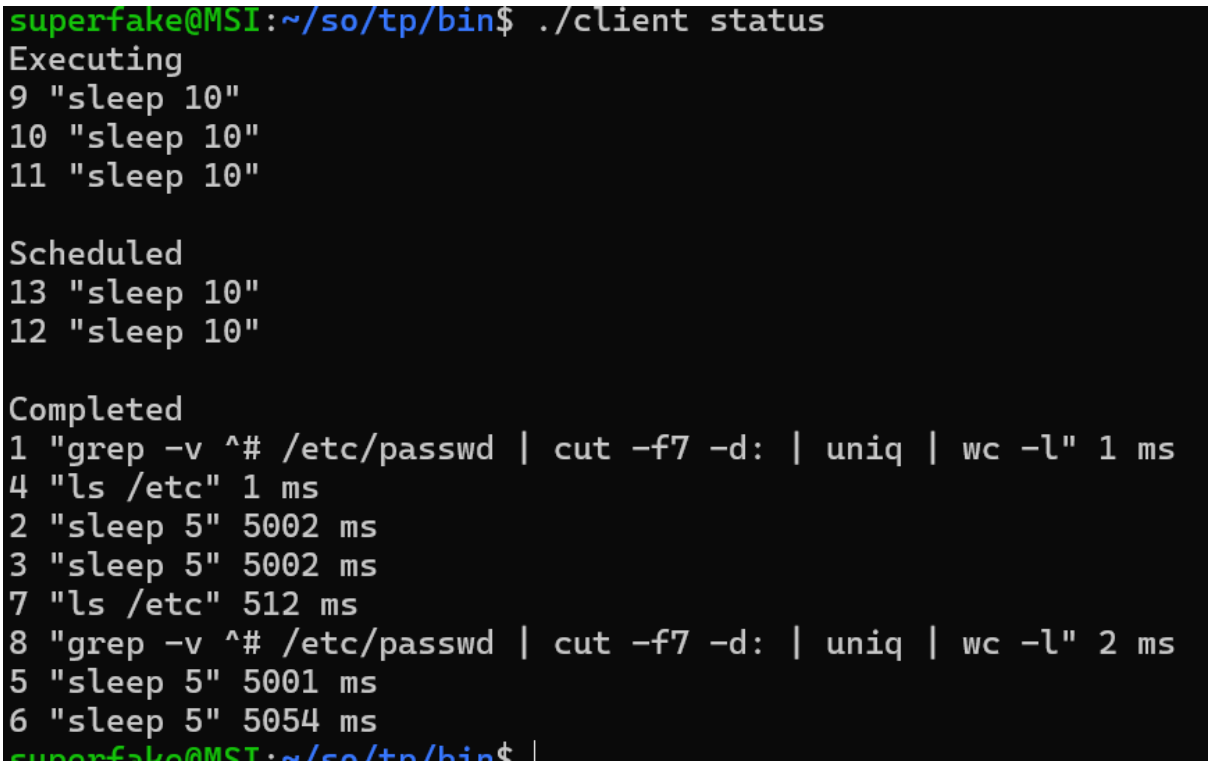
O programa cliente pode ser executado das seguintes maneiras:

```
superfake@MSI:~/so/tp/bin$ ./client execute 3 -u "ls -l -a -h"
TASK 1 Received
```

```
superfake@MSI:~/so/tp/bin$ ./client execute 4 -p "grep -v ^# /et
c/passwd | cut -f7 -d: | uniq | wc -l"
TASK 2 Received
```

Nestes dois casos, o servidor devolve de imediato o ID atribuído à task e será executado quando for possível. São diferenciados com um *type*

diferente na mensagem, já que o servidor não pode executar *pipelines* da mesma forma como executaria outro programa. Definimos *type* 0 para "-u" e 1 para "-p".



Quando se trata de um pedido de *status*, a resposta que o cliente deve receber é algo deste tipo. O *type* da mensagem neste caso é 3.

Orchestrator (servidor)

Inicialização

Tal como referido anteriormente, a primeira coisa que o servidor faz é criar o seu FIFO (*fifo_server*). De seguida, são feitos N forks de acordo com o número dado de tarefas que podem ser executadas em paralelo. Demos o nome de "workers" a esses processos que serão os responsáveis por executarem as tarefas. Cada um deles começa por criar um FIFO no qual receberão processos para executar. O nome dos pipes segue o formato *fifo_workerI* onde I é um número de 0 a N que lhes é atribuído sequencialmente no momento em que o *Orchestrator* efetua os forks.

Pedidos de execução

À medida que os pedidos de execução vão chegando, o servidor reencaminha-os para um dos workers caso estes estejam disponíveis. A verificação de disponibilidade é feita através de um array de inteiros auxiliar simples com zeros e uns. Caso todos os workers estejam ocupados, o servidor adiciona a tarefa a uma lista de espera (a gestão da fila de espera será explicada posteriormente).

Quando um worker acaba de executar uma tarefa, este envia uma mensagem para o servidor com *type* 2 através do *fifo_server* a declarar o fim da tarefa. O worker substitui o campo *pid* da mensagem original pelo *i* referente ao seu identificador, já que é importante para o servidor saber quem executou a tarefa.

Nesse momento, o servidor regista num ficheiro, ao qual demos o nome de "log.txt", o id, programa e tempo real de execução da tarefa em questão. Caso a fila de espera não esteja vazia, o *Orchestrator* atribui de imediato uma nova tarefa ao worker em questão, caso contrário apenas atualiza o array auxiliar para indicar que o worker *i* está livre.

Pedidos *status*

Quando se trata de um pedido *status*, é o servidor que tem de dar resposta imediata. A resolução deste pedido divide-se em 3 partes, sendo elas as tarefas em execução, em fila de espera e terminadas.

Quando se determina que uma tarefa irá ser executada, o servidor guarda num array auxiliar de tamanho *Nworkers* a mensagem relativa a esse pedido. Deste modo, apenas é preciso fazer uma simples consulta ao array para saber quais as tarefas em execução.

As tarefas em fila de espera são obtidas com uma passagem pela estrutura que representa a fila de espera em si.

Já os pedidos completados são dados através do ficheiro "log.txt" no qual havíamos escrito todas as tarefas previamente resolvidas.

Gestão da fila de espera

A política de escalonamento que utilizamos foi a *Shortest Job First* (SJF). Foi a política que nos saltou à vista desde logo dado que há um tempo estimado de execução associado a cada task.

Usamos uma *binomial heap* para gerir a fila de espera, já que se comporta como uma *priority queue*, sendo o campo ***time*** o critério de comparação para inserção na *heap*. Dessa forma, garantimos que quando efetuamos *pop*, obtemos a tarefa com menor tempo previsto o que contribui para uma redução do tempo médio de execução.

A implementação da *binomial heap* não foi feita por nós e tudo o que diz respeito a direitos de autor em relação ao seu uso foi respeitado.

Execução de tarefas

Quando uma tarefa chega a um worker para ser executada é necessário efetuar alguns procedimentos antes do "exec".

É primeiramente criado um ficheiro de nome "TASK*i*.txt" onde *i* é o ID da tarefa. Como se pretende que o *standard output* e o *standard error* da tarefa sejam redirecionados para esse ficheiro, efetuam-se 2 *dup2* do descritor do ficheiro para 1 (*output*) e para 2 (*error*).

Execute -u

Definido em "*mysystem.c*", é feito um parse à string que corresponde ao programa, colocando cada argumento num array. Cria-se então um processo-filho no qual se usa o *execvp* para se proceder à execução do programa.

Execute -p

Definido em "*execPipeline.c*", começamos por dividir o programa e subprogramas através do delimitador " | ", colocando-os num array. Cada subprograma é executado num processo-filho diferente.

A estratégia passa por usar pipes sem nome e ir fazendo redirecionamentos do *standard input* e *standard output*, de modo a que o output do primeiro subprograma seja o input do segundo, o output do segundo seja o input do terceiro e assim sucessivamente.

Conclusão

Em suma, consideramos que o trabalho desenvolvido foi um sucesso na medida que completamos com êxito as várias funcionalidades pedidas no enunciado. No entanto, não efetuamos testes extensivos de avaliação de políticas de escalonamento, já que apenas implementámos uma.

A frequência das aulas práticas da disciplina foi sem dúvida essencial para uma implementação suave e atempada do trabalho prático, visto que os guiões davam boas bases para desenvolver as estratégias que acabamos por implementar.