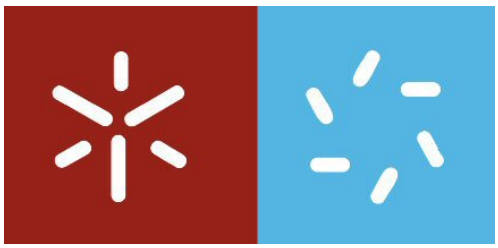

Universidade do Minho

Licenciatura em Ciências da Computação

Sistemas Operativos

Trabalho Prático

Grupo 15

Hugo Costa	Nuno Costa	Sara Fontes
(A96059)	(A97610)	(A92666)

May 13, 2023

Contents

Chapter 1

Introdução

Foi-nos proposto, no âmbito da unidade curricular de Sistemas Operativos, a criação de um serviço de execução de comandos(tarefas), no qual um cliente (*tracer*) consegue enviar várias tarefas a um servidor (*monitor*). Além disso, o servidor também é capaz de consultar tarefas em execução e/ou executadas anteriormente, gera um output para cada tarefa, e funcionalidades avançadas como a manipulação de informação sobre os processos já terminados.

O cliente e o servidor comunicam através de dois pipes com nome, um encarrega-se de enviar os comandos do cliente para o servidor, enquanto o outro envia o output dos comandos do servidor para o cliente.

Chapter 2

Funcionalidades Básicas e Avançadas

2.1 Funcionalidades Básicas

2.1.1 Execute -u

Submetendo um comando do tipo $./tracer\ execute\ -u$, seguido de uma string constituída pelo nome de um programa e respetivos argumentos, caso existam, irá ser escrito no output o PID do processo, o output do programa executado e o tempo total da execução em milissegundos.

2.1.2 Status

Caso o comando inserido seja do tipo $./tracer\ status$, o cliente recebe do servidor uma lista com todos os programas que estão em execução, obtendo assim o seu respetivo PID, o nome do programa e o tempo de execução.

2.2 Funcionalidades Avançadas

2.2.1 Execute -p

Este comando funciona de forma semelhante ao comando $./tracer\ execute\ -u$, no entanto, neste comando são inseridos vários programas com os seus argumentos separados por "|", e executando os programas inseridos na string simultaneamente.

2.2.2 Stats-time

Juntamente com o comando $./tracer\ stats-time$ é passada uma lista de $PIDs$ e o programa irá calcular e escrever no *standard output* a soma dos tempos de execução de todos os programas associados aos $PIDs$.

2.2.3 Stats-command

Aliado ao comando $./tracer\ stats-command$ é passado o nome de um programa e uma lista de $PIDs$, em que o programa deverá indicar quantas vezes o programa passado foi executado.

2.2.4 Stats-uniq

Com o comando *./tracer stats − uniq* é passada uma lista de $PIDs$, em que o programa irá escrever no *standard output* os nomes dos programas, sem repetições, associados aos $PIDs$ passados como argumento.

Chapter 3

Implementação

3.1 Cliente (tracer.c)

Adotamos uma implementação de uma estrutura de código que permite ler o argumento na posição um do que é passado no *standard input*. Caso o comando nessa posição seja:

- "*execute*" vamos analisar qual o argumento vem em seguida, e caso seja:
  - " $-u$" - chamamos uma função auxiliar $parser()$ que retorna uma struct $Program$ contendo informações do programa a ser executado (nome do programa e argumentos). Em seguida, um pipe é criado para comunicação entre processos. É criado um processo filho através do processo pai, em que o filho recebe o pipe e fecha o descritor de leitura, obtendo também o seu PID e envia-o para o pai. Em seguida, o filho executa o programa através da função $execvp()$. Por sua vez, o processo pai fecha o descritor de escrita e lê o PID enviado pelo filho, atualizando a estrutura $Program$ com o PID do filho e regista o status de inicio de execução do programa. No fim, escreve uma mensagem com o PID do filho e espera a finalização do mesmo, escrevendo o status de término do programa acabado de executar.
    Quando é passado um outro argumento no *standard input*, sendo ele um caminho para uma pasta, irá ser criado um ficheiro *.txt* com o PID, o nome do programa e o seu tempo de execução.
  - " $-p$" - percorre os comandos passados como argumento e criar pipes entre eles. Para cada comando, cria-se um processo filho e executa-se o comando. O código também faz o controle dos pipes e redireciona a entrada e saída padrão dos processos de acordo com a sequência de pipes. Por fim, espera-se que os processos filhos terminem a sua execução.
- "*status*" - utilizamos o pipe criado anteriormente, na comunicação entre processos, estabelecendo uma relação entre o cliente e o servidor. O cliente irá enviar uma flag que distinguirá este comando dos restantes, e o PID do processo. De seguida criamos o pipe que estabelece a relação entre o servidor e o cliente, o servidor envia os processos em execução, recebendo um número de processos ($num\_processes$) e, será criado um ciclo for para escrever os programas, individualmente, no *standard output*.
- "$stats-time$" - recebendo este comando seguido de uma lista de $PIDs$, o cliente inicializará um array onde guardará esses $PIDs$, através de uma função $pidsParser$ que fará a divisão dos argumentos da linha de comandos. Por fim, guarda o tamanho da lista dos $PIDs$ e invoca a função $stats\_time$.
- "$stats - command$" - guarda o nome passado no *standard input* numa string e faz o parsing da lista de $PIDs$ que vem em seguida recorrendo à função $pidsParser$. Por fim, guarda o tamanho da lista de $PIDs$ e invoca a função $stats\_command$.

- "$stats - uniq$" - faz o parsing da lista de $PIDs$ passada no *standard input*, recorrendo à função $pidsParser$ e guarda o tamanho da lista resultante. No fim, invoca a função $stats_uniq$.

3.1.1 Funções auxiliares

- "$parser$" - converte uma lista de argumentos numa estrutura $Program$ contendo o nome do programa e uma lista de argumentos que podem ser passados para esse mesmo programa.
- "$pidsParser$" - faz o parsing dos argumentos do programa separando os $PIDs$ e colocando-os num array.
- "$sendInitialStatus$" - envia informações - PID, nome do programa e timestamp - para o servidor. A função abre o pipe para escrita, envia um sinalizador, o PID, o nome do programa e um timestamp para o servidor e fecha a pipe. O timestamp é calculado a partir do tempo atual usando a função $gettimeofday()$. Por fim, a função retorna o timestamp enviado para o servidor.
- "$sendFinalStatus$" - envia para o servidor uma mensagem que indica o programa, e o respetivo PID, que terminou sua execução. Em seguida, a função obtém o timestamp atual e o envia para o servidor. No fim, retorna o timestamp em que a mensagem foi enviada.
- "$stats\_time$" - o cliente envia através do pipe $client\_server$ uma flag que irá distinguir o comando em questão. Em seguida, é enviado o tamanho do array com os $PIDs$ e percorrendo esse array é enviado um PID de cada vez para o servidor. Por fim, recebe do servidor o resultado final que corresponde à soma do tempo de execução dos $PIDs$ que foram enviados.
- "$stats\_command$" - envia para o servidor a flag que identifica o comando $stats - command$. Em seguida envia o tamanho da string que contém o nome do programa e o próprio nome e, envia os $PIDs$ da lista um de cada vez.
- "$stats\_uniq$" - tal como nos restantes comandos, envia para o servidor a flag que o identifica. Depois, envia o tamanho da lista dos $PIDs$ e envia um PID de cada vez.
- "$commandParser$" - separa o *standard input* em diferentes comandos através do caracter "—" e coloca esses comandos num array de comandos.
- "$exec\_command$" - divide a string em substrings usando o delimitador "espaço em branco" e armazena essas substrings num vetor de strings. O vetor de strings é, então, passado para a função *execvp* juntamente com o primeiro elemento do vetor como o nome do programa a ser executado. Retorna 0 em caso de sucesso e -1 em caso de falha na execução do comando.
- "$ficheiroTxt$" - cria um arquivo de texto com informações sobre a execução de um programa, informações essas que são o PID do processo, o nome do programa, o tempo de execução e a extensão "$.txt$".

3.2 Servidor (monitor.c)

Para o servidor desenvolvemos uma função principal $main(intargc, char **argv)$ que começa por criar o pipe $client\_server\_fifo$. A partir de aqui, o servidor lê do $client\_server$ a flag. Caso:

- $flag == 1$: está a ser recebido um novo programa. Inicializa-se um processo do tipo $Process$, que representa uma estrutura que contém o PID do processo, o nome do programa, o timestamp incial e final e o tempo de execução do processo. Em seguida, lemos o PID, o tamanho do nome do programa, o nome do programa e o timestamp inicial enviados pelo cliente, guardando estes dados no processo inicializado anteriormente. Por fim, o programa acrescenta o processo atual a uma lista de processos ativos, ou seja, uma lista que contém todos os processos em execução no momento.

- $flag == 2$: está a ser recebido o fim do programa. Em seguida, é lido do cliente o PID do processo que está a terminar e percorre a lista de processos ativos até encontrar o processo que possui o PID lido. Depois, lê o timestamp de finalização do processo e calcula o seu tempo de execução. Posto isto, adiciona o processo à lista de processos terminados e remove-o da lista de processos ativos.

- $flag == 3$: lê o PID do cliente e escreve, para o cliente, o número de processos na lista de processos terminados. Em seguida, itera sobre cada processo na lista de processos ativos e concatena ao buffer o PID e o tempo de execução do processo em formato de string. Cada buffer é então enviado juntamente com seu tamanho. No final, todos os recursos alocados são libertados.

- $flag == 4$: lê o tamanho da lista de $PIDs$ que será enviada pelo cliente. Em seguida, soma o tempo de execução de cada processo correspondente na lista de processos terminados. Após calcular a soma, é enviado para o cliente o número total de processos na lista de processos ativos, seguido por uma mensagem contendo o tempo total de execução dos processos terminados.

- $flag == 5$: lê do cliente o tamanho do nome e o nome do programa. Vai receber um PID da lista de $PIDs$ do *standard input* de cada vez, e para cada PID que recebe, verifica se esse PID corresponde a um PID de um processo que já tenha terminado e se o nome do programa é igual ao desse PID. Caso isto se verifique, então incrementa uma variável que contém a soma de vezes que um programa é executado numa lista de $PIDs$.

- $flag == 6$: recebe do cliente o tamanho do array com nomes de programas e para cada PID que recebe individualmente, verifica se esse PID corresponde a um PID de um processo que já tenha terminado e, caso seja e o seu respetivo nome do programa não esteja na lista de nomes de programas acrescenta-o a essa lista.

3.2.1 Funções auxiliares

- "$numNums$" - número de dígitos de um certo inteiro para auxiliar na alocação de memória.

- "$remove\_process\_by\_pid$" - remove um processo da lista de processos ativos, através do seu PID.

- "$remove\_duplicate\_strings$" - remove todas as strings repetidas de um array.

Chapter 4

Conclusão

Como é possível verificar, o nosso trabalho executa corretamente quase todas as funcionalidades pedidas. Logo, acreditamos que o nosso trabalho foi bem conseguido. Apesar de alguns desafios, tais como o resultado do envio de informações através dos pipes, não coincidia com o pretendido, a dificuldade em interpretar algumas funcionalidades, nomeadamente no comando *status*, etc. Uma peça fundamental que nos ajudou a superar os desafios, foram as resoluções dos guiões práticos apresentados pela equipa de docente.

Este trabalho permitiu-nos consolidar o que fomos aprendendo, ao longo deste semestre, na unidade curricular de Sistemas Operativos de uma maneira prática e cativante.