Universidade do Minho
Escola de Engenharia

Sistemas Operativos
Rastreamento e Monitorização da Execução de Programas - Grupo 10
2022/2023

Ana Filipa da Cunha Rebelo a90234
Tomás Cardoso Francisco a93193
Simão Paulo da Gama Castel-Branco e Brito a89482

Maio 2023

1 Introdução

O presente relatório enquadra-se na unidade curricular de Sistemas Operativos, na qual nos foi proposta a implementação de um serviço de monitorização dos programas executados numa máquina. Os utilizadores devem ser capazes de executar programas, através do cliente (tracer) e obter o seu tempo de execução. Um administrador de sistemas deve conseguir consultar, através do servidor (monitor), todos os programas que se encontram atualmente em execução, incluindo o tempo dispendido pelos mesmos. Finalmente, o servidor deve também permitir a consulta de estatísticas sobre programas já terminados.

Em seguida, iremos abordar com mais detalhe a estrutura dos programas e as funcionalidades implementadas, bem como as estratégias utilizadas para as implementar.

2 Programas Cliente e Servidor

A comunicação entre o cliente e o servidor é feita utilizando um pipe com nome "fifo". Para distinguir este pipe com nome de outros criados em certos casos no programa cliente (de modo a atender a funcionalidades específicas), vamos denominar este primeiro pipe com nome de pipe com nome "geral".

Primeiramente, o pipe com nome é criado através da função mkfifo, utilizada no início da main do programa monitor. Após a criação do pipe, o servidor abre-o para leitura e para escrita. Apesar de o servidor utilizar o fifo "geral" principalmente para leitura, ao abri-lo também para escrita não só evitamos situações de end-of-file em momentos em que não haja nenhum cliente conectado à extremidade de escrita do pipe, mas também permitimos que eventuais processos-filho criados pelo servidor possam escrever mensagens de fim da própria execução no pipe "geral" de modo a que o processo "pai" consiga ler.

```
if (mkfifo("fifo",0666) == -1) {
     if (errno != EEXIST) { // Só deu erro se o fifo ainda não existir
          perror("Erro ao criar o pipe com nome.");
          exit(1);
     }
}
if ((fifofdr = open("fifo", O_RDONLY)) == -1) {
     perror("Erro ao abrir o fifo para leitura.");
     exit(1);
}
if ((fifofdw = open("fifo", O_WRONLY)) == -1) {
     perror("Erro ao abrir o fifo para escrita.");
     exit(1);
}
```

Figure 1: Criação e abertura do fifo "geral"

Do lado do programa tracer, acede-se ao extremo de escrita do fifo "geral" sempre que for necessário enviar uma mensagem para o servidor. Após a escrita da mensagem no pipe, fecha-se o desritor associado.

```c
int fifofd;
if ((fifofd = open("fifo", O_WRONLY)) == -1) {
     perror("Erro ao abrir o fifo para escrita.");
     exit(1);
}

if (write(fifofd, buffer, sizeof(buffer)) == -1) {
     perror("Erro ao escrever no fifo.");
     close(fifofd);
     exit(1);
}

close(fifofd);
```

Figure 2: Exemplo de envio de uma mensagem por parte do cliente

Exatamente antes do fim da execução do programa monitor, são fechados os descritores de leitura e escrita do fifo "geral" através da função *close()*.

Quanto às estruturas dos programas em si, o tracer começa por fazer uma verificação do nº de argumentos passado e verifica qual das funcionalidades foi pedida. Para cada funcionalidade, e após verificar se foram passados todos os argumentos necessários e se estes são válidos, chama uma função que vai executar o pretendido.

```c
int main(int argc, char* argv[]) {
    // É necessário, no mínimo, um argumento para além do executável
    if (argc < 2) {
        fprintf(stderr, "Erro: faltou especificar uma opção.\n");
        fprintf(stderr, "Opções disponíveis: execute, status, stats-command, stats-uniq.\n");
        return 1;
    }

    // Tratamento e execução das funcionalidades
    if(!strcmp(argv[1],"execute")) {
        // A funcionalidade "execute" tem, pelo menos, 3 argumentos após o executável:
        // "execute"; flag (-u ou -p); nome do programa a executar + possíveis argumentos do programa
        if (argc < 4) {
            fprintf(stderr, "Erro: Invocação inválida da opção \"execute\". Formato correto:\n");
            fprintf(stderr, "./tracer execute -[u/p] \"(progs e args)\"\n");
            return 1;
        }

        //Verificação da flag utilizada
        if (!strcmp(argv[2], "-u")) {
            execute_program(&argv[3], argc-3);
        } else if (!strcmp(argv[2], "-p")) {
            execute_program2(argv[3]);
        } else {
            fprintf(stderr, "Erro: flag inválida.\n");
        }
```

Figure 3: Exemplo do tratamento dos argumentos para as funcionalidades de execução de programas

Já o monitor, após guardar o nome da pasta passado como argumento e criá-la caso ainda não exista, e após a criação e abertura de ambas as extremidades do fifo "geral" já vistas anteriormente, entra num loop (*while(1)*) no qual cada iteração vai começar com a leitura de 1 byte correspondente ao tipo de execução feita pelo tracer e ao tipo de pedido que o tracer pretende do monitor. Os tipos de pedidos são os seguintes:

- '1' - Início da execução de um programa
- '2' - Fim da execução de um programa
- '3' - Pedido de *status*
- '4' - Pedido de *stats-time*
- '5' - Pedido de *stats-uniq*
- 'F' - Informação de término de execução de um processo-filho que estava a processar um pedido de *status*
- 'G' - Informação de término de execução de um processo-filho que criou um ficheiro com informações de término de execução de um programa
- 'S' - Informação de término de execução de um processo-filho que estava a processar um pedido de *stats-time*
- 'T' - Informação de término de execução de um processo-filho que estava a processar um pedido de *stats-uniq*

3 Funcionalidades Implementadas

O desenvolvimento do projeto foi dividido em dois tipos de Funcionalidades: Básicas e Avançadas. Inicialmente começamos por implementar as funcionalidades básicas e de seguida passamos para as avançadas. Em seguida, iremos abordar em mais detalhe cada uma das funcionalidades implementadas.

3.1 Funcionalidades Básicas

No que diz respeito ás funcionalidades básicas foram implementadas as seguintes funcionalidades:

3.1.1 Execução de programas do utilizador

Numa pequena nota inicial, nós desenvolvemos esta funcionalidade com a ideia de que ao executar o tracer, o programa enviado como argumento para a funcionalidade "execute -u" não estaria "dentro de" aspas, só nos apercebendo disso aquando do início da construção da funcionalidade avançada "execute -p", estando esta última preparada para o uso de aspas.

Para armazenar as informações relativas a programas em execução e programas que terminaram de executar, foram criadas 2 structs. Ambas vão conter o pid do programa e uma variável "time" que para uma delas vai guardar o *timestamp* de início de execução e para a outra de fim de execução. A principal diferença entre as structs é que a primeira guarda ainda o nome do programa, enquanto que a struct usada para informações de fim de execução não. Assim, evitamos ter instâncias de estrutura com uma variável inutilizada que ocuparia espaço em memória desnecessário.

```
typedef struct program_info {
     pid_t pid; // guarda o PID do programa.
     char name[MAX_NAME_LENGTH]; // guarda o nome do programa.
     long int time; // guarda o start_time
} program_info;

typedef struct end_info {
     pid_t pid; // guarda o PID do programa.
     long int time; // guarda o end_time
} end_info;
```

Figure 4: Structs utilizadas para armazenamente de informações de programas

Quanto à funcionalidade em si, desenvolvemos a função *execute_program*. Esta função recebe um apontador para o 4ª posição do array de argumentos passado na execução do programa. Isto significa que a variável *program_args* vai ser um apontador para o nome do programa a executar, permitindo-nos usá-la para a execução do programa em si. No início da função criamos uma instância da struct program_info que guarda o pid do programa, o nome e o start_time (obtido recorrendo à struct *timeval* e à função *gettimeofday()* definidas na biblioteca ¡sys/time.h¿ que é importada) e escrevemos essa mesma instância (à qual juntamos no início o caracter '1' que indica o tipo de mensagem) no fifo "geral" de modo a notificar o monitor que houve um programa a iniciar a sua execução. Notificamos ainda o cliente, escrevendo no STDOUT uma mensagem que indica o PID associado ao programa que começou a executar. O monitor, que tem um array *program_info* run_progs* que guarda as informações relativas aos programas em execução, vai ler essa instância do fifo "geral" e adicioná-la ao array.

```c
void execute_program(char** program_args, int argc) {
    program_args[argc] = NULL;

    // Instância da struct program_info que guarda o pid do programa, o nome e o start_time
    program_info info;
    info.pid = getpid();
    //info.name = malloc(strlen(program_args[0]) + 1);
    strcpy(info.name,program_args[0]);
    info.time = getCurTime();

    char buffer[sizeof(program_info)+1]; // buffer a ser enviado na 1ª notificação - início da execução do programa
    buffer[0] = '1'; // o 1º byte contém o tipo de mensagem. Neste caso 1 é uma mensagem de início da execução do programa
    memcpy(buffer+1, &info, sizeof(program_info));

    // Notificação ao servidor do start_time associado ao pid e ao nome do programa
    int fifofd;
    if ((fifofd = open("fifo", O_WRONLY)) == -1) {
        perror("Erro ao abrir o fifo para escrita.");
        exit(1);
    }

    if (write(fifofd, buffer, sizeof(buffer)) == -1) {
        perror("Erro ao escrever no fifo.");
        close(fifofd);
        exit(1);
    }

    close(fifofd);

    printf("Running PID %d\n", info.pid);
```

Figure 5: Notificação do início da execução do programa por parte do tracer ao monitor e ao cliente

```c
case '1': // mensagem de início da execução de um programa
    program_info tmp_info;

    // lê do fifo e guarda na variável tmp_info os dados do programa cuja execução começou
    if(read(fifofdr,&tmp_info,sizeof(program_info)) == -1) {
        perror("Erro ao ler a mensagem do fifo.");
        exit(1);
    }

    // adiciona a variável tmp_info ao array dos programas em execução
    if (num_run_progs < max_progs) {
        run_progs[num_run_progs] = tmp_info;
        num_run_progs++;
    } else {
        max_progs *= 2;
        run_progs = realloc(run_progs, max_progs * sizeof(program_info));
        run_progs[num_run_progs] = tmp_info;
        num_run_progs++;
    }

    printf("[EXEC START] %d;%s;%ld\n", tmp_info.pid, tmp_info.name, tmp_info.time);
    break;
```

Figure 6: Leitura da notificação por parte do monitor

Após as notificações, o tracer executa o programa, criando um processo-filho para o efeito e recorrendo à chamada de sistema *execvp()*. O processo-pai, após esperar pelo término do processo-filho, cria uma instância da struct *end_info* na qual guarda o tempo *timestamp* atual. Volta a enviar uma notificação ao monitor através do pipe com nome "geral", desta vez do tipo '2' que representa o fim de execução do programa. Por fim, o tracer informa o cliente do tempo de execução total do programa. Do lado do monitor, é recebida a instância de fim de execução do programa e remove-se o a instância com o PID correspondente do array de programas em execução. É ainda realizado o armazenamento de informação sobre o programa terminado, que vamos ver em maior detalhe na secção das Funcionalidades Avançadas.

```
// Execução do programa
pid_t child;
int status;
if ((child = fork()) == 0) {
     execvp(program_args[0],program_args);
     _exit(64); // encerrar o processo filho em caso de erro no execvp
} else {
     wait(&status);
     if (WIFEXITED(status) && WEXITSTATUS(status) != 0) {
          fprintf(stderr, "Erro: não foi possível executar o programa. Código de saída: %d.\n", WEXITSTATUS(status));
     }
}

// Notificação ao servidor do end_time associado ao pid do programa
end_info end;
end.pid = info.pid;
end.time = getCurTime();

char buffer2[sizeof(end_info)+1]; // buffer a ser enviado na 2ª notificação - fim da execução do programa
buffer2[0] = '2'; //o 1º byte contém o tipo de mensagem. Neste caso 2 é uma mensagem de fim da execução do programa
memcpy(buffer2+1, &end, sizeof(end_info));

if ((fifofd = open("fifo", O_WRONLY)) == -1) {
     perror("Erro ao abrir o fifo para escrita.");
     exit(1);
}

if (write(fifofd, buffer2, sizeof(buffer2)) == -1) {
     perror("Erro ao escrever no fifo.");
     close(fifofd);
     exit(1);
}

close(fifofd);

// Notificação ao utilizador do tempo de execução utilizado pelo programa
printf("Ended in %ld ms\n", end.time-info.time);
```

Figure 7: Execução do programa e envio das notificações de término de execução ao cliente e ao monitor

```
case '2': // mensagem de fim da execução de um programa
     end_info tmp_end;

     if(read(fifofdr,&tmp_end,sizeof(end_info)) == -1) {
          perror("Erro ao ler a mensagem do fifo.");
          exit(1);
     }
     (...)
     // remove o programa do array de programas em execução
     int i, j;
     for(i=0; i < num_run_progs; i++) {
          if(run_progs[i].pid == tmp_end.pid) {
               for (j=i; j < num_run_progs-1; j++) {
                    run_progs[j] = run_progs[j+1];
               }
               num_run_progs--;
               break;
          }
     }
     printf("[EXEC END] %d;%ld\n", tmp_end.pid, tmp_end.time);
     break;
```

Figure 8: Leitura da notificação por parte do monitor e remoção do programa do array de programas em execução

3.1.2 Consulta de programas em execução

A funcionalidade *status*, que lista os programas em execução no momento, foi implementada através da função *getCurrentStatus()*. Esta função cria um pipe com nome, utilizando o PID (convertido em *string*) juntamente com o prefixo "own_" para o nome do fifo. Na continuação do relatório iremos referir-nos a este tipo de fifos criados pelo tracer como fifos (ou pipes com nome) "próprios". Após a criação do pipe com nome, o tracer comunica ao servidor - através do fifo "geral" - que pretende obter informações sobre os programas em execução e informa-o do nome do seu fifo "próprio" no qual o monitor deve escrever a mensagem de *status*. De seguida, abre o pipe com nome "próprio" do qual vai ler a mensagem enviada pelo monitor, imprimindo-a, posteriormente, no STDOUT.

O monitor, por sua vez, vai ler do fifo "geral" o nome do fifo "próprio" do tracer e criar um processo-filho para tratar do pedido de *status*, de modo a que o próprio monitor possa atender outros pedidos de outros clientes. O processo-filho constrói a mensagem de *status* e escreve-a no fifo "próprio" do cliente em questão. No fim, o processo-filho envia o caracter 'F' para o fifo "geral" para que o pai o leia e saiba que o filho está a terminar a sua execução com sucesso.

```c
void getCurrentStatus() {
    // Uso do pid para obter um nome único para criação do pipe com nome "próprio"
    pid_t pid = getpid();
    char pid_str[20];
    sprintf(pid_str, "%d", pid);
    char fifo_name[25] = "own_";
    strcat(fifo_name,pid_str);

    // Criação do pipe com nome "próprio" no qual o monitor vai escrever o conteúdo do status e do qual o programa tracer vai ler
    if (mkfifo(fifo_name,0666) == -1) {
        if (errno != EEXIST) { // Só deu erro se o fifo ainda não existir
            perror("Erro ao criar o pipe com nome.");
            exit(1);
        }
    }

    // Envio do nome do fifo "próprio" para o monitor via fifo "geral" ("fifo")
    int fifo_fd;
    if ((fifo_fd = open("fifo", O_WRONLY)) == -1) {
        perror("Erro ao abrir o fifo para escrita.");
        exit(1);
    }
    char send_msg[strlen(fifo_name)+1]; // mensagem a ser enviada (tipo de mensagem (1 byte) e nome do fifo (strlen bytes))
    send_msg[0] = '3'; // o 1º byte contém o tipo de mensagem. Neste caso 3 é uma mensagem de pedido de status
    memcpy(send_msg+1, fifo_name, strlen(fifo_name));
    if (write(fifo_fd, send_msg, sizeof(send_msg)) == -1) {
        perror("Erro ao escrever no fifo.");
        close(fifo_fd);
        exit(1);
    }
    close(fifo_fd);

    // abrir o próprio pipe com nome para leitura
    int ownfd;
    if ((ownfd = open(fifo_name, O_RDONLY)) == -1) {
        perror("Erro ao abrir o fifo para leitura.");
        exit(1);
    }

    // Leitura da mensagem do monitor no fifo "próprio" e escrita no stdout
    size_t msg_size = 0;
    size_t max_msg_size = 100;
    char *rcv_msg = malloc(max_msg_size * sizeof(char));
    ssize_t bytes_read = 0;
    while((bytes_read = read(ownfd,rcv_msg + msg_size, 100)) > 0) {
        msg_size += bytes_read;

        if (msg_size >= max_msg_size) {
            max_msg_size *= 2;
            rcv_msg = realloc(rcv_msg, max_msg_size);
        }
    }
    rcv_msg[msg_size] = '\0';
    if (bytes_read == -1) {
        perror("Erro ao ler do fifo \"próprio\".");
        exit(1);
    }
    write(STDOUT_FILENO, rcv_msg, strlen(rcv_msg));
    free(rcv_msg);
    close(ownfd);
    unlink(fifo_name);
}
```

Figure 9: Função *getCurrentStatus()* do programa tracer

```c
case '3': // mensagem que pede o status dos programas em execução
    // Leitura via fifo "geral" do nome do fifo "próprio" do cliente
    char st_name[25];
    if(read(fifofdr,&st_name,sizeof(st_name)) == -1) {
        perror("Erro ao ler do fifo o nome do fifo \"próprio\" do cliente.");
        exit(1);
    }

    // Criação de um novo processo para lidar com a tarefa, enquanto o pai pode atender outros clientes
    if (fork() == 0) {
        close(fifofdr); // O filho não precisa (nem deve) ler do fifo "geral"

        // Abertura do fifo "próprio" do cliente para escrita
        int st_fd;
        if ((st_fd = open(st_name, O_WRONLY)) == -1) {
            perror("Erro ao abrir o fifo \"próprio\" do cliente para escrita.");
            exit(1);
        }

        // Construção da mensagem de status a enviar para o cliente que fez o pedido
        long int cur_time = getCurTime();
        int l;
        char* status_msg = NULL;
        for(l=0; l < num_run_progs; l++) {
            int lsize = snprintf(NULL, 0, "%d %s %ld ms\n", run_progs[l].pid, run_progs[l].name, cur_time-run_progs[l].time);
            char* tmp = malloc(lsize+1);
            snprintf(tmp, lsize+1, "%d %s %ld ms\n", run_progs[l].pid, run_progs[l].name, cur_time-run_progs[l].time);

            if (l == 0) status_msg = calloc(lsize+1,sizeof(char));
            else status_msg = realloc(status_msg, strlen(status_msg) + lsize + 1);

            strcat(status_msg,tmp);
            free(tmp);
        }

        // Envio da mensagem de status para o cliente, caso haja programas em execução
        if (status_msg != NULL) {
            write(st_fd, status_msg, strlen(status_msg));
        } else {
            write(st_fd, "Não existem programas em execução.\n", 38);
        }

        free(status_msg);
        close(st_fd);

        write(fifofdw,"F",1);
        close(fifofdw);
        _exit(0);
    }

    break;
```

Figure 10: Tratamento do pedido de *status* por parte do monitor

3.1.3 Servidor

A última funcionalidade básica pretendia que o servidor suportasse sempre que possível, e citamos o enunciado, "o processamento concorrente de pedidos, evitando que clientes a realizar pedidos que obriguem a um maior tempo de processamento possam bloquear a interação de outros clientes com o servidor". Isto foi assegurado ao fazer com que pedidos de consultas (por exemplo pedidos de *status*) feitos ao servidor por parte de clientes sejam tratados por um processo-filho criado no servidor, permitindo ao processo-pai continuar a ler do fifo "geral" outros pedidos feitos por outros clientes sem esperar pelo término da execução do processo-filho. De modo a que o processo-pai assegure o término da execução do processo-filho, o último envia um caracter via fifo "geral", o qual o processo-pai vai ler e perceber que o processo-filho terminou a sua execução com êxito.

3.2 Funcionalidades Avançadas

3.2.1 Armazenamento de informação sobre programas terminados

A primeira funcionalidade avançada que desenvolvemos é relativa ao armazenamento de informação sobre programas terminados. Isto foi feito recorrendo à criação de um processo-filho dentro do case '2' do switch no código do programa monitor, que corresponde à receção das informações de término de um programa enviadas por um cliente. Após receber essas informações (e antes de remover o programa do array de programas em execução) é então criado um processo-filho que cria um ficheiro cujo nome é o PID do programa cuja execução terminou. Este ficheiro é criado dentro do *folder* passado como argumento na execução do monitor. Caso o ficheiro seja criado com sucesso, é então escrita no ficheiro uma linha que contém o nome do programa, 1 caractér ';', o tempo de execução total do programa e um ponto final ('.'). Por fim, antes de "fazer" *_exit(0)*, o processo-filho escreve no fifo "geral" o caractér 'G' que informa o pai do término da sua própria execução.

Figure 11: Armazenamento das informações do programa terminado num novo ficheiro

3.2.2 Consulta de programas terminados

A funcionalidade de consulta de programas terminados tem 3 comandos possíveis:

- *stats-time* - Imprime no *standard output* o tempo total (em milisegundos) utilizado por um dado conjunto de programas identificados por uma lista de PIDs passada como argumento.
- *stats-command* - Imprime no *standard output* o número de vezes que foi executado um certo programa, cujo nome é passado como argumento, para um dado conjunto de PIDs, também passado como argumento.
- *stats-uniq* - Imprime no *standard output* a lista de nomes de programas diferentes (únicos), um por linha, contidos na lista de PIDs passada como argumento.

Os comandos *stats-time* e *stats-uniq* foram implementados no programa tracer recorrendo, ambos, à função *statsTimeUniq()* que recebe como argumentos o tipo de pedido/comando ('4' para *stats-time* e '5' para *stats-uniq*), a lista de PIDS para a qual pretendos obter o tempo total de execução dos programas e o número de PIDS presente nessa lista.

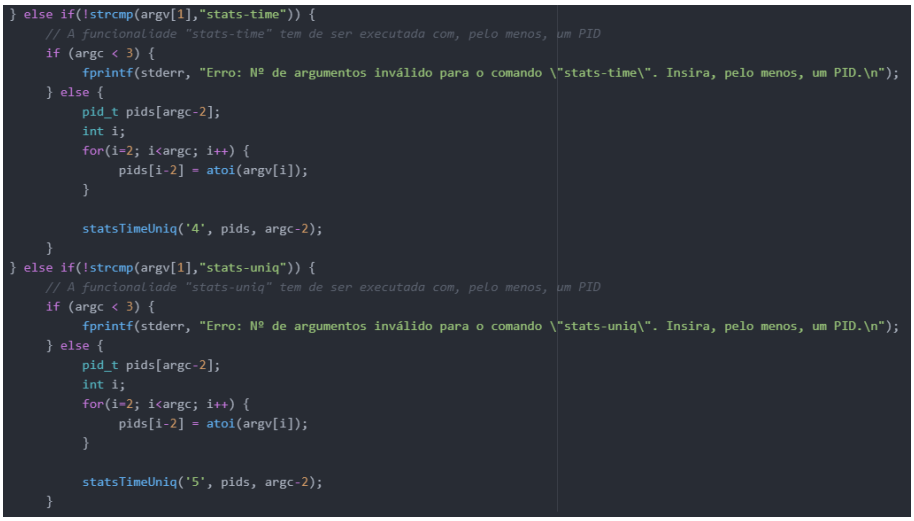

Figure 12: Tratamento de pedidos *stats-time* e *stats-uniq* por parte do tracer

A função *statsTimeUniq()* em si funciona de maneira parecida à função *getCurrentStatus()* na medida em que:

1. Cria um pipe com nome "próprio", cujo nome é composto pelo pid (convertido em string) e um prefixo "stats_"
2. Cria uma mensagem com o tipo de mensagem/comando ('4' ou '5'), tamanho do nome do fifo "próprio", nome do fifo "próprio", nº de pids no array e o array com os pids
3. Envia a mensagem pelo fifo "geral"
4. Abre o fifo "próprio" e lê dele ou um *int* ou um array de *strings* dependendo do tipo de comando, '4' ou '5', respetivamente.

Já o programa monitor, recebe o tipo de pedido/comando e processa o pedido. No caso de pedidos do tipo *stats-time*, o tratamento do pedido é feito da seguinte forma:

1. Lê do fifo "geral" e armazena internamente os dados necessários para tratamento do pedido
2. Cria um novo processo para lidar com o pedido, permitindo ao pai processar outros pedidos de outros clientes
3. O novo processo criado por sua vez cria *n filhos*, em que *n* é o número de PIDS recebido. Cada um destes novos processos-filho vai tratar de obter o tempo de execução de um dos PIDS (abrindo e lendo do ficheiro cujo nome é esse mesmo PID) e enviá-lo via um pipe anónimo para o pai.

4. O processo-pai vai somar o inteiro lido do pipe anónimo a uma variável, também ela do tipo *int*, que armazena o tempo total de execução dos programas. O pai espera ainda pelo término de todos os filhos e no fim, escreve no fifo "próprio" do cliente que efetuou o pedido o tempo total de execução dos programas, enviando ainda um caractér 'S' pelo fifo "geral" informando o término da sua execução ao seu próprio pai.

Os pedidos de *stats-uniq* são processados no servidor de um modo semelhante ao processamento de pedidos de *stats-time*, com a diferença que em vez de ter uma variável inteira que vai guardar a soma de todos os tempos de execução dos PIDS contidos no array, vai guardar os nomes dos programas num array de apontadores para string, tratando de eliminar os repetidos e enviando esse mesmo array para o cliente.

Por último, o comando *stats-command* ainda não está desenvolvido e por isso aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

3.2.3 Execução encadeada de programas

Falta isto. e por isso aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

4 Testes

5 Conclusão

Dado por concluído o trabalho prático , consideramos importante realizar uma análise crítica, e ainda, realçar os pontos positivos e negativos do trabalho realizado.

A implementação da totalidade das funcionalidades básicas e algumas das Avançadas constitui um ponto positivo do nosso trabalho.

Por fim, o grupo considera que o trabalho realizado é bastante positivo,pois cumpre quase todos os requisitos propostos no enunciado.