Projeto de Sistemas Operativos

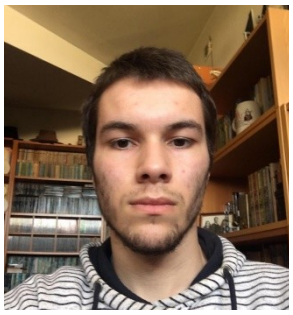
Grupo 51

Carlos Miguel Luzia de Carvalho A89605

Francisco Correia Franco A89458

José Pedro Carvalho Costa A89519

A89605



A89458

A89519

Índice

Introdução

Nesta unidade curricular foi-nos proposta a implementação de um serviço de monotorização e de comunicação de processos.

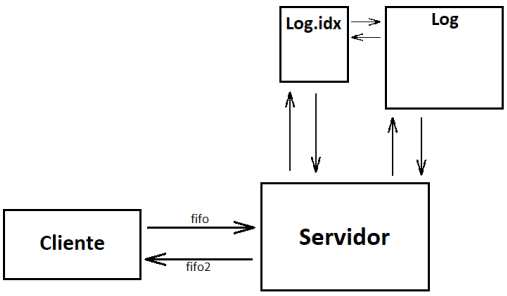
Este serviço é dividido em duas partes, o servidor onde são executados comandos encadeados por pipes anónimos, e a interface do utilizador, pela qual podem ser submetidos os comandos para o servidor. Para além disso, o utilizador pode especificar o tempo máximo de execução de uma tarefa (execução de um comando) e o tempo máximo de inatividade de um pipe, pedir informação sobre as tarefas que estão a decorrer, bem como as que já terminaram, terminar uma tarefa que o servidor está a executar e consultar o output de uma certa tarefa.

A comunicação entre o servidor e o utilizador é feita entre dois pipes com nome, pois cada pipe pode ser apenas usado num sentido. O tipo de informação passado neles será abordado mais a frente.

Desenvolvimento do projeto

1. Ordem de implementação

Numa fase inicial do desenvolvimento deste projeto, o nosso foco foi **deliniar a nossa abrodagem** relativamente à **comunicação** entre o cliente e o servidor, bem como a **implementação dos logs**. A partir daí orientamo-nos pela imagem seguinte.



Implementamos a comunicação no sentido do cliente para servidor, utilizando um **pipe com nome**, ao que chamamos **fifo**, e decidimos qual seria o tipo de informação a ser transmitida pelo pipe, ao que chegamos a esta **estrutura**:

```
#define COMMAND_SIZE 1024
struct estrutura{
    int comando;                //inteiro referente ao comando a executar
    int valori;                 //informação int  a transmitir (ex.: terminar)
    char valorc[COMMAND_SIZE];  //informação string a transmitir (executar)
};
```

Fizemos então a interface do utilizador, ou seja, a passagem de informação referente ao comando que queremos executar e a receção e o processamento da mesma no servidor.

Tendo realizado esta funcionalidade básica, foi possível começar a **desenvolver** as **funcionalidades pedidas** pelo enunciado individualmente. A ordem que achamos mais benéfica foi a execução de um comando, o tempo de execução, o tempo de inatividade (que se provou das mais dificeis e foi deixada para último), terminar, guardar o outupt nos ficheiros log. No **ficheiro log** é guardado todo o **output dos comandos**, sem qualquer divisória entre eles e **log.idx** é gravada a informação referente ao comando executado e a localização do seu output no ficheiro log, através da **estrutura** a baixo.

```
struct logidx{
    int tarefa;                 //numero da tarefa
    char estado[20];            //estado no qual terminou
    char valorc[COMMAND_SIZE];  //comando executado
    int index;                  //indice do output no ficheiro log
    int size;                   //tamanho do output
};
```

Depois instalamos todas as **funcionalidades** que **enviam output** ao utilizador, as funções listar, histórico e output de uma tarefa, sendo necessário **criar** outro **pipe com nome**, ao qual chamamos **fifo2** e definimos que iria passar **strings**.

2. Interface do utilizador

A interface do utilizador, como já foi referido, é dividida em duas partes, a linha de comandos, em que se indica as opções apropriadas, e a interface textual interpretada (shell), onde o comando aceita instruções do utilizador através do standard input.

É utilizada a **shell** caso o programa seja executado com **menos de 2 argumentos**, é executada a shell, que é um **ciclo while** que usa a função **read** para **ler do standard input**. Após a leitura de uma string, esta é validada através da função **validateop**, que retorna um inteiro associado ao comando que se pertende executar ou -1 no caso de erro. É feito um **switch** do valor obtido, **validado o input** consuante o comando que se pertende executar, e se o comando estiver correto, é enviada a estrutura com a informação através do **fifo** e espera um resultado do servidor através do **fifo2** (se for o caso). O while termina quando for enviado um end of file. Exempo de utilização:

```
zepedro@zepedro-X550CC:~/2s/SO/Projeto$ ./argus
argus$ ajuda
tempo-inactividade segs
tempo-execucao segs
executar 'p1 | p2 ... | pn'
listar
terminar n
historico
ajuda
output n
argus$ executar 'cat | cat'
nova tarefa #1
argus$ listar
#1: cat | cat
argus$ terminar 1
argus$ historico
#1, terminada: cat | cat
argus$
```

A segunda opção é passagem de **flags e valores** com a execução do programa, ou seja, quando são recebidos **pelo menos 2 argumentos**. Para o processamento dos argumentos é chamada a função **getflags** que retorna 0 se for bem sucedida. Esta função faz um **parse** sequencial dos argumentos, procurando **flags** e caso necessite, **guarda o valor associado** numa variável global. Exemplo de utilização:

```
zepedro@zepedro-X550CC:~/2s/SO/Projeto$ ./argus -i 3
zepedro@zepedro-X550CC:~/2s/SO/Projeto$ ./argus -e "cat | cat"
nova tarefa #1
zepedro@zepedro-X550CC:~/2s/SO/Projeto$ ./argus -m 2
zepedro@zepedro-X550CC:~/2s/SO/Projeto$ ./argus -e "cat | cat"
nova tarefa #2
zepedro@zepedro-X550CC:~/2s/SO/Projeto$ ./argus -e "ls -l | wc"
nova tarefa #3
zepedro@zepedro-X550CC:~/2s/SO/Projeto$ ./argus -r
#1, max inatividade: cat | cat
#2, max execucao: cat | cat
#3, concluida: ls -l | wc
zepedro@zepedro-X550CC:~/2s/SO/Projeto$ ./argus -o 3
     16     137     881
zepedro@zepedro-X550CC:~/2s/SO/Projeto$
```

3. Servidor

O servidor, como foi dito no primeiro tópico, começou por apenas receber a informação do comando enviado pelo utilizador. Para isso, é criado o fifo e o programa entra num ciclo while(1). Nesse ciclo é aberto o fifo para leitura e feito um ciclo **while**, que vai **ler do fifo**. Caso não exista informação para ler, o programa vai ficar bloqueado até haver. Por isso foi criado um handler para o sinal **SIGINT**, de maneira a poder **terminar o programa**. Inicialmente esse handler apenas removia o fifo e fazia um _exit(0), mas agora também envia um sinal para cada processo que está a decorrer para terminar os processos. Após receber informação através do fifo, esta é processada por um **switch** de acordo com o primeiro campo da estrutura. Saindo do ciclo que lê do fifo é **fechado o descritor** de leitura.

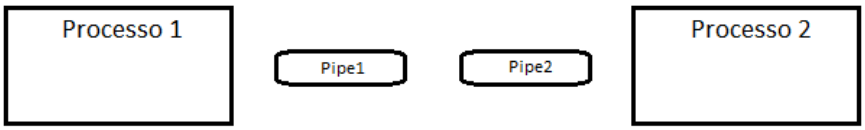
O **id da tarefa** seguinte é guardado na variavel global **tarefa_index** e é obtido pelo numero da última tarefa executada apresentada no log.idx mais um. Caso nao exista nenhuma tarefa no log.idx a variável é inicializada a um.

Como o objetivo do projeto era o **envio simultanea de comandos**, na opção do switch para a **execução de um comando** é realizado um **fork**, é guardado o **pid** do processo criado no array global de inteiros **pids** e o **comando** executado no array global de strings **comandos**. De forma a não ficar a espera que o filho termine, a receção da informação do fim do processo filho é feito no handler **sigchld_handler**, no qual se vai ao array pids, procura-se o **pid do processo que terminou** e coloca-se a -1. O filho remove os handlers herdados do pai e chama a função **executar**. Nesta função o comando é dividido por '|', que indicam os pipes que vão ser criados, e são guardados num array de strings. De seguida é feito um **ciclo for** e para cada iteração exceto a ultima é criado um **pipe anónimo** e redirecionado o input e o output para a interligação de comandos. É feito um **fork** e o filho chama a função **mysystem** que vai dividir o comando individual por espaços, redirecionar input e output caso seja especificado no comando e faz **exec** do comando. A **última iteração** do ciclo for da função executar é utilizada para a **gravação do output** nos ficheiros log. Por fim é feito um **wait** para cada processo filho criado. Para testar esta funcionalidade, criamos um programa auxiliar que lia do ficheiro log.idx, sendo que não podiamos fazer cat do mesmo, por estar escrito em binário. Consuante o que lia, esta ferramenta apresentava o output do comando realizado na tarefa.

A funcionalidade de **tempo máximo de execução** foi relativamente fácil de criar. Quando é enviado pelo utilizador um pedido de alteração do tempo de execução o servidor apenas verifica se o valor é maior ou igual a zero e altera a variável global **te**. Para a implementação, antes do ciclo for da função executar é criado o **sigalrm_execucao_handler** e é chamado um alarme com **te** segundos. Caso seja recebido o **SIGALARM** antes do fim a execução do comando, o handler vai matar todos os processos criados para executar os comandos individuais, dos quais foram gravados os **pids** no array global **pidsexecucao** e no inteiro global **pidsaver**. No handler é também alterado o estado em que terminou a tarefa no log.idx. Tanto nesta funcionalidade como no tempo de inatividade, para testar o seu funcionamento, enviavamos um comando do tipo "cat" ou "cat!cat", pois apenas termina se o tempo máximo chega-se ao fim.

O **tempo máximo de inatividade** foi mais complicado do que pensavamos. O início foi semelhante ao tempo de execução, alterando a variável global **ti** em vez da **te**. Primeiramente a implementação era algo mais parecido a tempo máximo de execução de cada comando individual. Foi implementado um **alarme** e um **handler** na função **mysystem**, de modo a limitar o tempo máximo de execução desse processo, apesar de sabermos que não estava muito correta a implementação. Depois de terminarmos as outras funcionalidades alteramos esta, de

modo a fazer o que é pedido. No for da função **executar**, em vez de ser criado um pipe para interligar dois processos, **são criados dois pipes**, como demonstra a imagem em baixo.



É feito um while que vai ler do primeiro pipe1, que fica **bloqueado** se o **pipe** estiver **inativo**. Antes do while é criado um alarme com **ti segundos** e um handler, que envia um **SIGUSR1** ao processo **pai**, que ao ser recebido, tem um comportamento semelhante ao do tempo de execução. Quando entra no ciclo desliga o alarme, pois o pipe já deixou de estar inativo.

Para **terminar uma tarefa** em específico, é **validado** se o **valor da tarefa** recebido é maior que zero e se o **pid** que esta no indice do valor recebido no array de global de **pids** é maior que zero. Se estas condições se verificarem, é enviado o sinal **SIGUSR2** para o processo que está a executar a função **executar**. Esta função tem um handler para este sinal, semelhante ao do tempo de inatividade e o tempo de execução. Foi testado o terminar de funções do tipo "cat|cat", pois sabiamos que estaria ainda a ser executada, bem como funções do tipo "ls", que sabiamos que já teria terminado quando o servidor recebesse a instrução de terminar. Foi também experimentado um script que enviava o comando para executar e logo de seguida a instrução de terminar a tarefa, para verificar que o programa conseguia responder.

A opção **listar as tarefas em execução** apenas chama a função **writeemexecucao**. Esta função abre o **fifo2** para **escrita**, percorre o array global **pids**, e se o valor for maior que zero, significa que ainda está a ser executado, por isso formata-se uma string com a informação a enviar, e envia-se. Depois **fecha-se** o descritor do **fifo2**. Para verifivar esta opção, foram enviados alguns comandos que terminaram e outros que ainda estavam a executar, e foi verificado se o servidor os tinha identificado corretamente.

O **histórico de tarefas** foi interpretado por nós como mostrar de todas as tarefas que já foram corridas, ou seja, o **conteudo do log.idx**. Desta forma, esta opção chama a função **writehistorico**, que abre o **log.idx** para **leitura** e o **fifo2** para **escrita**. Para cada estrutura lida do log.idx é **formatada** uma **string** e **enviada** pelo **fifo2**. Depois fecham-se os descritores. Caso a função histórico fosse suposto apenas imprimir os comandos executados depois da abertura do servidor, bastava avaliar se o **id da tarefa** é **maior que** a variável global **start_tarefas**, que guarda o valor da primeira tarefa executada pelo servidor desde que ele foi aberto. De modo a testar esta funcionalidade foram enviados todos os tipos de comandos, uns que terminaram com sucesso e têm output a mostrar, outros que terminaram com sucesso mas sem output a mostrar (foi redirecionado para um ficheiro, por exemplo), alguns que ainda estão a executar, e também uns terminados por tempo máximo de execução, inatividade ou por pedido do utilizador.

Por fim, para ver o **output de uma tarefa** específica. Nesta opção é chamada a função **writeoutput**, passando o valor da tarefa da qual se quer ver o output. Esta função vai **abrir** o ficheiro **log.idx** para **leitura** e procurar pelo numero da tarefa, para **obter o indice e o tamanho do output** do comando, e fecha-se o descritor do log.idx. De seguida **abre-se** o **fifo2** para **escrita**. Caso tenha sido encontrada a tarefa no log.idx e o estado dela seja concluido, **abre-se** o ficheiro **log** para **leitura**, faz-se **lseek** para a posição do indice, faz-se um while enquanto se ler do log, lê-se até se chegar ao tamanho do output que se quer, e **escreve-se** no **fifo2**. Depois fecha-se o descritor do log. Fora do if, fecha-se o descritor do fifo2. Esta funcionalidade foi testada de um modo parecido à anterior.

4. Possíveis melhorias

Neste projeto existem duas melhorias nas quais pensamos e que consideramos ser boas funcionalidades a implementar, apesar de não as termos adicionado.

A primeira consiste em **substituir** o **vetor** que guarda os **pids dos processos** criados que vão tratar da execução de um comando (através da função executar), **e** o **vetor** que guarda os **comandos associados** a esses pids **por uma lista ligada** com essas informações. A **vantagem** consiste no **tratamento do fim** desse **processo**, não ter de percorrer o vetor todo, mesmo as tarefas que já existiram e que já terminaram, à **procura** do **pid** do processo que **terminou** agora e enviou o **SIGCHLD**, para colocar a -1. Apesar de sabermos que era uma boa implementação, achamos que não era bem o foco da disciplina a utilização de listas, dando prioridade a outras implementações.

A segunda resume-se a **criação** de um programa que denominamos de "**logger**". Na main iria ser criado um pipe anónimo e um fork para a criação de um processo que iria executar o "logger". Este programa seria o **único que irira mexer nos ficheiros log e log.idx**, de forma a **impedir escritas simultaneas** por diferentes processos. Iria receber, por exemplo, parte do output de uma tarefa e ia guardando no log e no log.idx as atualizações. A estrutura do ficheiro log.idx iria ser um pouco alterada, de forma a conseguir guardar mais que um indice relativo ao ficheiro log, e mais que um tamanho de output. Nós chegamos a começar a codificar o programa, mas devido a falta de tempo, consideramos melhor não nos focarmos nele. O pouco código que criamos não foi enviado como ficheiro, mas pode ser visto no anexo, sendo que estava funcional.

Conclusão

Em retrospetiva, a realizacão deste projeto mostrou-se desafiante, mas enriquecedora. Conseguimos aplicar os conhecimentos obtidos na unidade curricular, como comunicação de processos entre pipes com nome e anónimos, forks, sinais, dups, execs e escrita em ficheiros.

Como já mencionamos em cima, sabemos que poderiamos ter implementado algumas funcionalidades que deixariam o trabalho um pouco melhor. Foram ideias que exploramos, mas devido a falta de tempo não foram incluidas no projeto.

A cima de tudo, consideramos que foi feito um bom trabalho e achamos que foi bastante motivante o facto de obter um produto final visivel e com um propósito.

Anexos

```c
#define LOG "log2"
#define LOGIDX "log.idx2"

#define SIZEVALORC 64
struct logger{
    int tarefa_index;
    int modo;               //0->criar,1->adicionar log,2->alterar estado
    char valor[504];        //504 pois mais dois inteiros da 512, tamanho de escrita que assegura que a escrita no pipe é de um unico write
};
struct logger s;

struct logidx{
    int tarefa;
    char estado[10];
    char valorc[SIZEVALORC];
    int index[100];         //consegue gravar 100*504 bytes de output para cada tarefa
    int size[100];
};
```

```c
void criarlogidx(){
    int flogidx,enc=0;
    if((flogidx=open(LOGIDX,O_CREAT | O_RDWR,  0666))==-1){
        perror("open log.idx");
    }

    struct logidx slogidx;
    slogidx.tarefa=s.tarefa_index;
    strcpy(slogidx.estado,"erro");
    strcpy(slogidx.valorc,s.valor);
    slogidx.size[0]=0;

    if(write(flogidx,&slogidx,sizeof(slogidx))==-1){
        perror("write");
    }

    close(flogidx);
}
```

```c
int adicionarlog(){
    int flogidx,enc=0;
    if((flogidx=open(LOGIDX,O_CREAT |O_APPEND| O_RDWR,  0666))==-1){
        perror("open log.idx");
    }

    struct logidx slogidx;
    while(!enc && (read(flogidx,&slogidx,sizeof(struct logidx)))>0){
        if(slogidx.tarefa==s.tarefa_index) enc=1;
    }
    if(enc) lseek(flogidx,-1*sizeof(struct logidx),SEEK_CUR);

    int i=0,flog;
    while(slogidx.size[i]!=0)i++;

    if((flog=open(LOG,O_CREAT | O_WRONLY , 0666))==-1){
        perror("open log");
        return -1;
    }
    slogidx.index[i]=lseek(flog,0,SEEK_END);
    if(write(flog,s.valor,strlen(s.valor))==-1){
        perror("write");
    }
    slogidx.size[i]=strlen(s.valor);
    slogidx.size[i+1]=strlen(s.valor);
    close(flog);

    if(write(flogidx,&slogidx,sizeof(slogidx))==-1){
        perror("write");
    }

    close(flogidx);
    return 0;
}
```

```c
int main (int argc, char* argv[]){
    int o;
    while((o=read(0,&s,sizeof(struct logger)))>0){
        printf("[LOGGER] %d:modo %d->%s\n", s.tarefa_index,s.modo,s.valor);
        switch(s.modo){
            case 0:
                printf("criar em logidx\n");
                criarlogidx();
                break;
            case 1:
                printf("adicionar no log\n");
                printf("%d\n",adicionarlog());
                break;
            case 2:
                printf("alterar estado\n");
                break;
            default:
                printf("erro\n");
                break;
        }
    }
    return 0;
}
```