Sistemas Operativos

Serviço de Execução de Tarefas

João Alves 54741 && André Pimenta 54745

Índice

1 Introdução

No âmbito na unidade curricular de Sistemas Operativos, do segundo ano da Licenciatura em Engenharia Informática, foi proposto a realização de um trabalho prático, de modo a aplicar os nossos conhecimentos adquiridos não só na própria unidade curricular de Sistemas Operativos mas também em outra unidades curriculares, como por exemplo Programação Imperativa e Algoritmos e Complexidade.

Este relatório tem como objectivo ilustrar e comentar os principais problemas enfrentados e as soluções adoptadas, as estruturas utilizadas e alguns métodos que são basilares para a concretização dos objectivos propostos no enunciado deste trabalho prático.

Este trabalho propunha aos alunos a realização de um Serviço de Execução de Tarefas que possuísse determinadas características:

- Composto por dois programas distintos, um programa que gere tarefas e outro que as submete;
- Suporte para uma sintaxe especifica e modular de submição de tarefas;
- Detecta a existência de dependencias entre tarefas distintas;
- Executa, quando possível, tarefas mesmo que existam ainda outras tarefas a executar;

Tal como já foi referido o Serviço de Execução de Tarefas é realizado à custa de dois programas. O programa responsável pela submição das tarefas é o mais simples possível e recebe a codificção da tarefa a executar através do seu primeiro argumento que depois passa para o programa, que se deverá encontrar a correr em *background*, responsável pela *parsing*, calendarização e execução da tarefa essa codificação.

O programa reponsável pela gestão das tarefas chama-se *Gestor* e o programa que submete as tarefas denomina-se *Submeter*.

2 Estruturas de Dados

De modo a estruturar a informação que o nosso programa necessita de gerir e também de modo a permitir que o ataque a certos problemas seja mais fácil e directo, o grupo considerou muito importante idealizar e despender algum do seu tempo a estudar quais as estruturas de dados necessárias para a realização do trabalho.

Desde logo tornou-se que claro que seria necessário estruturas de dados para guardar a informação:

- De um processo;
- De uma submição (tarefa);
- Do próprio gestor de tarefas;

Nas próximas subsecções iremos dedicar-nos à explicão das estruturas de dados que utilizamos para guardar essa informação.

2.1 Processo

A estrutura responsável por guardas as informações afectas a um processo denomina-se *PROC* e é definida pelas seguintes linhas de código:

```
typedef struct processo {
        char *nome;
        char *list_arg[MAX_CMD];
        pid_t pid;
        int pipeInput;
        int pipeOutput;
        struct processo* next;
} *PROC;
```

Segue-se a descrição detalhada de cada um dos campos:

Campo	Descrição
nome	Nome do processo a executar
list_arg	Lista dos argumentos do processo
pid	Identificação do processo
pipeInput	*File descriptor* de input
pipeOutput	*File descriptor* de output
next	Proximo processo na lista ligada

O list_arg é um *array* que tem a característica da sua primeira posição conter o nome do próprio processo e a ultima posição conter o valor NULL. No nosso serviço de execução de tarefas o valor de MAX_CMD é 30 logo, o número máximo de argumentos por processo é 30 sendo os argumentos posteriores ignorados.

Este estrutura de dados permite que a execução de algumas funções seja muito simples, por exemplo caso eu queira iniciar um processo basta:

```
execvp(processo->nome, processo->list_arg);
```

A utilização de um apontador para um outro processo de modo a criar uma estrutura de dados é propósitada e está relacionada com a natureza do projecto proposto pelo enunciado. O nosso grupo conceptualiza cada tarefa como possuidora de uma lista de processos (que por vezes possui apenas um elemento) e portanto torna-se muito natural criar uma lista ligada na estrutura de dados relativa aos processos.

2.2 Tarefa

As estruturas que armazenam a informação relativas às tarefas são nucleares para o bom funcionamento de todo o projecto. No nosso programa, idealizamos uma tarefa como sendo uma submição para o programa gestor. A estrutura de dados que guarda a informação dessa submição é definida como:

```
typedef struct submicao {
        PROC proc;
        char* input;
        int id_input;
        char* output;
        int id_output;
        int numProcessos;
        int pid_proc_final;
        int pid_proc_inicial;
} *SUBM;
```

Segue-se a descrição detalhada de cada um dos campos:

Campo	Descrição
proc	Lista de processos afectos à submição
input	Input da submição
id_input	File descriptor do input da submição
output	Output da submição
id_output	File descriptor do output da submição
numProcessos	Numero de processos afectos à submição
pid_proc_final	PID do processo final
pid_proc_inicial	PID do processo inicial

Caso a submição não possua um *input* ou um *output* especificado os valores correspondentes na estrutura de dados são inicializados a *NULL*.

No entanto pode ocorrer o caso em que o programa gestor necessita de gerir mais que uma submição ao mesmo tempo e para isso criamos uma estrutura que é pouco mais que uma lista ligada de submições. Segue-se o código da sua definição:

```
typedef struct lista_submicao {
        SUBM sub;
        int id;
        int dependencia[2];
        struct lista_submicao* next;
} *LISTA_SUB;
```

Esta estrutura já traduz uma visão de mais alto nível daquilo que é na verdade uma submição para o nosso gestor. O campo sub contem a submição correspondente desta entrada da lista e o campo id armazena um inteiro único correspondente a esta submição. Este id criado com base nas submições já atendidas.

No entanto a parte mais importa desta estrutura de dados refere-se ao facto de armazenar as depêndencias de uma determinada submição. Tal como é explicado no enunciado uma submição possui no máximo duas depências (uma relativa ao ficheiro de input inicial e outra relativa ao ficheiro de output final). Caso ambas as entradas neste array estejam a zero, então é seguro assumir que este submição ou está pronta para ser executada, ou está a ser executada ou ja terminou a sua execução e necessita de ser apagada. Este array contem o valor do pid do processo que esta a ocupar um dos recursos que necessita.

Imaginemos o seguinte caso `ls >out.txt` em que o ls tem de pid 1003 caso uma nova submição dependa do ficheiro *out.txt* entao uma das entradas do array dependencia vai possui o valor 1003.

2.3 Gestor

Existe na nossa aplicações que gere as tarefas uma estrutura responsável por guardar os dados de estado da sua execução. Essa estrutura é definida por:

```
typedef struct gest_strc {
        LISTA_SUB pedidos;
        int numSubmicoesActivas;
        int numSubmicoesInactivas;
        int numSubmicoesAtendidas;
} GESTOR;
```

Os campo pedidos armazena o apontador para a lista de pedidos de gestão submetidos para o nosso programa, os restantes campos são bastante auto explicativos.

O numero de submições activas indica o número de submições que se encontram em execução enquanto que o número de submições inactivas indicam o número de submições que se encontram em espera para serem executadas (provavelmente por causa de dependências com outros processos).

O numero de submições atendidas diz respeito ao numero de submições total passadas para o gestor, o id de cada submição é dado com base neste número.

Pelo facto de esta ser a estrutura de dados de mais alto nível do nosso trabalho decidímos nomeá-la simbólicamente gestor. A única variável global do nosso programa é definida como `GESTOR gestor;`. É necessáia a existência de uma variável global principalmente porque nos *handlers* dos sinais necessitamos de ter acesso a toda a informação sobre o estado do nosso gestor.

3 Comunicação entre Gestor e Submeter

Um dos problemas que enfrentamos na criação deste projecto relaciona-se com o facto de necessitarmos de passar informação entre dois programas cujo antecessor comum nós não conseguimos controlar.

Esta passagem de informação é necessária pois o programa Submeter necessita de passar o valor que recebeu como argumento para o programa Gestor. Por exemplo:

```
./Submeter ''ls > out.txt''
```

Neste caso o programa Submeter necessita de passar a *string* que recebeu para o programa Gestor.

A resolução que encontramos para este problema passa pela a utilização de uma *fifo*. Uma *fifo* é um ficheiro *first-in first-out* especial também denominado como PIPE nomeada[1].

Este ficheiro especial pode ser aberto por vários processos para escrita e para leitura. Neste caso a *fifo* que vamos utilizar vai ser aberta em modo escrita pelo programa Submeter e aberta em modo leitura pelo programa Gestor.

A *fifo* no nosso caso vai ser criada pelo programa Gestor ao iniciar e removida pelo programa Gestor sempre que termina a sua execução. Dentro do programa gestor possuimos a função *criaEntrada* que cria a fifo que vai ser utilizada para comunicação entre os dois programas:

```
int criaEntrada() {
        (void)remove("/tmp/gestorFIFO");
        return (mkfifo("/tmp/gestorFIFO",0644));
}
```

Esta função é muito simples, consiste apenas em duas linhas, e faz:

1. Tenta remover uma fifo com o mesmo nome da fifo a ser criada;
2. Cria um fifo utilizando a função *mkfifo*.

É de notar que nós não podemos garantir, sem verificação, que uma fifo com o mesmo nome da que nós vamos criar já exista portanto tentamos sempre remover, no caso dessa fifo existir.

Como podemos observar a fifo é denominada *gestorFIFO* e é guardada na directoria de ficheiros temporários */tmp/*.

[1] *Named Pipe*

Utilizar esta fifo é bastante simples basta utilizar a funções *write* e *read*.No entanto, no programa que lê da fifo existe o problema em saber qual o tamanho que deve ler. Assim, a solução por nós adoptada para este problema passa por passarmos sempre um inteiro antes de mandar-mos a *string* que contem a tarefa. Este inteiro contem o tamanho que devemos ler no nosso próximo acesso à fifo.

```
read(FIFO, &tamanhoParaLer, sizeof(int));
read(FIFO, buffer, tamanhoParaLer);
```

A nossa metodologia de acesso á fifo é bastante fácil de perceber atráves do código posto em cima. O tamanho para ler corresponde ao tamanho da submição que vai ser lida e buffer é o local para onde essa submição vai ser carregada.

De notar que para cada submição não pode possuir mais do que 1024 caracteres que corresponde ao número máximo de caracteres que o buffer suporta.

4 Parse de Submição

Cada submição que é passada como string pelo programa Submeter para o programa Gestor necessita de ser convertida para a estrutura de dados *SUBM*[2]. Para realizar esta conversão é necessário fazer o parse desta string. A função responsável inteiramente por este parsing denomina-se *parseSubmicao*.

A primeira coisa que é feita à string recebida é trim, de modo a eliminar os espaços em branco no seu fim e no seu início. De seguida corremos a funções *getInputOutput* que alem de ir preencher os campos input e output da struct submicao também devolve um inteiro que nos permite saber em que local relativo na string se encontram os processos.

São considerados três casos genéricos de strings, na função *getInputOutput*, cada letra corresponde a uma substring:

1. Caso em que existem dois '>'; (Ex: "a > b > c")
2. Caso em que não existem '>'; (Ex: "a")
3. Caso em que existe um '>'; (Ex: "a > b")

O primeiro caso é simples de resolver, o input é representado pela substring a, o output pela substring c e os processos pela substrinng b. O segundo caso é também simples de resolver, não existe input e output definidos e os processos são representados pela substring a.

O terceiro caso é muito mais difícil de resolver. Poe não conseguimos facilmente saber onde é representada a substring dos executáveis, se em a se em b. Seguem os nossos critérios para decidir este caso, por ordem de prioridade:

1. Se a ou b contiver '—' entao corresponde à substring que representa os processos;
2. Se a ou b contiver a substring ".txt" então corresponde à substring que representa o input (no caso do a) ou o output (no caso do b);
3. Se a ou b fôr a um executável então corresponde à substring que representa os processos;
4. Se tudo falhar, a representa o input e o b representa os processos;

É de salientar, a maneira como verificamos se determinada string corresponde a um processo. Para esta verificação usamos a seguinte função:

[2]Ler capítulo relativo às estruturas de dados.

```
access(CaminhoExecutavel,X_OK);
```

Os locais testados são a directoria do programa Gestor e as directorias do sistema */bin/* , */usr/bin/*, */sbin/* e */usr/sbin/*.

O parsing da sublista dos processos é agora muito mais facil pois ja conseguimos localizar a substring que diz apenas respeito aos processos. Esse parsing é efectuado pela função *separaComandos*, que devolve a cabeça da lista ligada de processos que cada submição contem.

5 Submição

5.1 Submição de uma Tarefa

Após a tarefa recebida pelo programa Submeter ser convertida numa submicao, esta é adicionada à lista de submiçoes. O id dá sua entrada na lista de submições corresponde ao número de submições já atendidas ao longo da corrente execução do programa Gestor.

É verificado se o campo input da submição esta a nulo, se estiver então não vão ser verificadas colisões com o input desta submição. O mesmo acontece com o campo input da submição.

Seguidamente a essa verificação, percorremos todas as submições. E caso encontremos uma submição que esteja a utilizar um dos recursos que a nova tarefa também necessita necessitamos de estabelecer uma relação de depêndencia entre as duas submições.

Na verdade é estabelecido uma ligação entre a submição dependente e o processo responsável pelo recurso.

```
./Submeter "sleep 100 | ls > out.txt" -> Submição 1
./Submeter "out.txt > wc" -> Submição 2
```

A segunda submição não está dependente da primeira submição! Está sim dependente do processo *ls*. Pois é o processo *ls* o responsável pelo recurso *out.txt* do qual a segunda submição depende. Assim, no campo dependencia é inserido o pid do *ls*. Deste modo, mal o *ls* acabe, a submição 2 pode arrancar.

Para entre duas submições existem quatro tipo de conflitos possíveis:

- Recurso a ser utilizado como input duplamente;
- Recurso a ser utilizado como output duplamente;
- Recurso a ser utilizado como input do novo pedido esta a ser utilizado como output em outra submição; 1item Recurso a ser utilizado como output deste pedido esta a ser utilizado como input em outra submição;

Em todos estes tipo de conflitos a única solução é adiar a execução da nova tarefa até os recursos serem libertados. Assim, no fim de procurarmos conflitos numa nova tarefa, se estes não forem encontrados executamo-la, se forem encontrados colocamos a tarefa em espera.

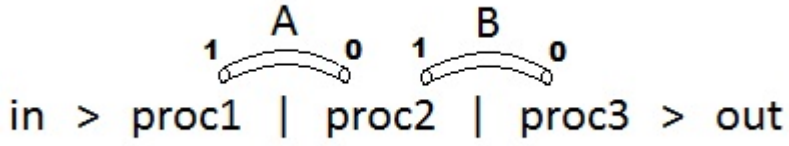

Figura 1: Esquema de uma Tarefa

5.2 Execução de uma Submição

A rotina que executa uma submição é talvez a mais extensa do nosso projecto, isto é um reflexo da complexidade e da generalidade inerente a esta rotina.

O objectivo da rotina responsável pela a execução de uma tarefa não passa só por pôr a correr todos os seus processos. É necessário direccionar o input e o output de todos os processos manualmente. Por exemplo, seguindo a imagem, o output do proc1 deve ser o input do proc2 ou entao o output do proc3 deve ser redireccionado para o ficheiro out.

Ao atender um processo existem três tipos distintos de casos:

- Caso em que é o primeiro processo (Input dado pelo utilizador ou não);
- Processo no meio da submição (Temos de ser nós a canalizar o input e o output);
- Caso em que é o último processo (Output dado pelo utilizador ou não).

Vamos analisar o atendimento de uma tarefa igual à representada na imagem. O proc1 cai sobre o primeiro caso, assim em primeiro lugar fazemos uma pipe (neste caso chama-se A) e verificamos se existe um ficheiro de input especificado. Como existe, abrimos esse ficheiro depois, fazemos *fork*, e no processo filho:

1. Redireccionamos esse ficheiro para o input do proc1 (usando a função dup2);
2. Fechamos o lado 0 da pipe A;
3. Redirecionamos o output do processo para o lado 1 da pipe A;
4. Fazemos exec do proc1;

No processo pai:

1. Fechamos o ficheiro de input do proc1;

2. Fechamos o lado de input do pipe A;

3. Actualizamos o campo do pid do processo que agora iniciou;

De seguida passamos para o proc2, este processo cai sobre o caso 2. Fazemos uma nova pipe (chamada B) e de seguida fork. No processo filho:

1. Ligamos o lado 0 do pipe A ao input do proc2;

2. Fechamos o lado 0 da pipe B;

3. Redireccionamos o output do proc2 para o lado 1 da pipe B;

4. Fazemos exec do proc2;

O processo pai faz o equivalente ao que eu já descrevi anteriormente mas desta vez é relativo ao processo proc2 e à pipe B. Estamos finalmente no processo proc3. Desta vez não necessitamos de fazer uma pipe, necessitamos apenas de abrir o ficheiro de output (neste caso é o ficheiro out) e de seguida mais um fork. No processo filho:

1. Ligamos o lado 0 do cano B ao input do proc3;

2. Redireccionamos o output do proc3 para o ficheiro out;

No processo pai, fechamos o lado de input do cabo B e fechamos também o ficheiro de output. De notar que sempre que fazíamos fork mandávamos logo de seguida (no processo pai) um sinal para parar o processo filho. Assim, no final de fazer todos os redirecionamentos e forks fico com um monte de processos filhos parados. É portanto, necessário percorrer de novo a lista ligada de processos e mandar-lhes um sinal para retomarem execução.

O sinal mandado para um processo para a sua execução é o SIGSTOP e para resumir é o sinal SIGCONT. Nós decidimos parar os processos logo no inicio da sua execução e depois resumi-los todos muito rapidamente, como maneira de evitar problemas como SIGPIPES ou afins.

5.3 Terminar uma Submição

Sempre que um processo de uma submição morre o contador do número de processos associados a essa submição é decrementado. Quando este contador chegar a zero, isso significa que a tarefa está completa e pode ser apagada.

A função responsável por apagar uma submição chama-se *rotinaSubmicaoApagada* esta função recebe como argumentos um apontador para a estrutura anterior da lista de submições e um outro apontador para a estrutura da lista de submições que deve ser apagada.

Em primeiro lugar é necessário percorrer todos os processos da tarefa que terminou e fazer *free* às informações as informações guardadas sobre eles.

```
while (proc) {
        aux = proc;
        proc = proc-> next;
        free(aux->nome);
        i = 0;
        while(aux->list_arg[i]){
                free(aux->list_arg[i]);
                i++;
        }

        free(aux);
}
```

Como se pode ver fazemos não só free ao nome mas também a todos os argumentos de cada processo.

6 Signal Handlers

6.1 Sinais de terminação

Existe uma variedade de sinais que quanda lançados, se não forem apanhados causam a terminação do programa. No nosso programa Gestor tentamos apanhar esses sinais de modo a poder terminar todos os processos filhos e libertar as dependências deste nosso programa, em particular a named fifo.

```
signal(SIGTERM, rotinaFecho);
signal(SIGABRT, rotinaFecho);
signal(SIGINT, rotinaFecho);
signal(SIGQUIT, rotinaFecho);
signal(SIGUSR1, rotinaFecho);
signal(SIGUSR2, rotinaFecho);
signal(SIGHUP, rotinaFecho);
signal(SIGPIPE, handlerPipe);
```

De notar que a função *handlerPipe* não é nada mais que uma mensagem de erro seguida de uma chamada para a função *rotinaFecho*.

Tal como foi dito o handler deste tipo de sinais deve ser responsável por terminar os processos filhos e libertar as dependências do programa. É exactamente isso que esta rotina faz.

A primeira coisa que o handler dos sinais de terminação realiza é fazer com que os sinais SIGCHLD passem a ser ignorados. Isto é necessário para podermos facilmente terminar todos os processos filhos.

Para terminar os filhos a função *rotinaFechar* percorre todos os processos filhos e manda-lhes o sinal SIGTERM, utilizando a função *kill*.

```
kill(processoFilho,SIGKILL);
```

É de salientar que o sinal SIGTKILL faz com que o processo filho termine imediatamente e não pode ser ignorado nem apanhado.

Após terminar todos os processos filhos, resta apenas remover a fifo.

```
remove("/tmp/gestorFIFO");
```

Tarefas Com a fifo removida a *rotinaFechar* acaba com a execução do programa utilizando a função *exit*.

6.2 Processo Filho Termina

Quando um processo termina é mandado um sinal SIGCHLD ou processo pai. A acção por defeito dos programas é ignorar este sinal. No entanto no espectro particular deste problema isso seria obviamente um erro. No nosso

caso, quando o nosso processo recebe um SIGCHLD, é chamada a função *rotinaFilhoMorreu*.

É necessário o programa Gestor ser alertado quando um processo filho termina para saber quando uma tarefa terminou ou se um determinado recurso já não se encontra em uso.

A primeira coisa que o *handler* deste sinal faz é passar a ignorar os sinais SIGCHLD de modo a tentar garantir a sua atomicidade. De seguida tenta apanhar o pid do processo filho que lhe enviou o sinal através da função:

```
pid_filho = waitpid(-1,&status,WNOHANG);
```

Esta função, por causa da opção WNOHANG, não fica à espera que um processo filho morra e retorna um pid inválido caso não exista nenhum processo filho cujo pid não tenha sido recuperado. A variável status vai também indicar-nos se o processo filho terminou ou não correctamente.

Como já referi, logo no inicio da *rotinaFilhoMorto* nós passamos a ignorar os sinais que nos indicam que um filho terminou, e para não deixar-mos de tratar filhos que tenham terminado usamos o seguinte ciclo:

```
void rotinaFilhoMorto (int val) {
        signal(SIGCHLD,SIG_IGN);
        ...(declaracoes de algumas variaveis)...
        pid_filho = waitpid(-1,&status,WNOHANG);

        while(pid_filho > 0) {
                ...(Codigo)..
                pid_filho = waitpid(-1,&status,WNOHANG);
        }
        signal(SIGCHLD,rotinaFilhoMorreu);
}
```

Assim, enquanto o *waitpid* apanhar pid válidas continuamos a correr o código da rotina.

Após sabermos qual o pid do processo que terminou, procuramos qual a submição que possui esse processo. Sabendo qual a submição que diz respeito a um determinado processo, podemos saber se esse processo esta responsável por um qualquer determinado recurso.

Se estiver responsável por algum recurso, o programa procura por submições que estejam dependentes do recurso libertado. Quando encontra um processo nessas condições põe-no responsável pelo recurso. Se a subição referente ao processo que ficou responsável pelo recurso tiver resolvido os seus problemas de dependências então é executada.

Na submição referente ao processo que morreu é diminuído o numero

de processos. Caso o numero de processos fique a zero, isso significa que a submição não tem nenhum processo a correr e que pode ser apagada.

Este rotina pode também ser chamada caso se detecte que um processo filho terminou sem ter sido tratado por esta rotina. Para ser chamada desta maneira é necessário passar o valor negativo correspondente ao pid do processo filho que terminou. Por exemplo, se for detectado que o processo filho 564 tiver terminado e este não tenha sido tratado por este handler então é só executar : `rotinaFilhoMorreu(-564)`.

6.3 Alarme

Uma das protecções que decidimos implementar no nosso Gestor de Tarefas passa por sempre que uma nova tarefa é atendida pôr um alarme a disparar passados 3 segundos. Caso passem 3 segundos sem nenhuma nova tarefa ser atendida o nosso programa vai verificar se existem processos filhos que já terminaram, mas não foram tratados pelo nosso Gestor. Isto acontece raras vezes, mas a implementação desta protecção dá uma maior robustez à nossa aplicação que fica portanto menos vulnerável.

Caso o alarme dispare é executada a função rogueCleaner, que tal como já foi explicado tem como objectivo limpar os processos que conseguiram escapar à jurisdição da rotinaFilhoMorreu. A função rogueCleaner verifica se existem submições activas, se não existirem põe o alarme para disparar passados 15 segundos.

Se existirem submições activas no nosso gestor, esta função verifica um a um todos os processos que são considerados em execução e verifica se eles ainda estão a correr.

A função para verificar se um processo se encontra em execução chama-se *processExists* e encontra-se definida no ficheiro *aux.h*.

```
int processExists (int pid) {
        return (kill(pid,0) == 0);
}
```

Esta função verifica se é possível mandar um sinal inofensivo ao processo com a pid recebido. É uma especie de fazer um *poke* ao processoa ver se ele reage. Se não reagir assumimos que ele não existe, e que portanto não se encontra em execução.

7 Conclusao

Este enunciado trouxe-nos muitos desafios novos e engraçados com que os elementos do grupo nunca se tinham deparado. A realização que pipes entre processos em cadeia através de um código que seja genérico para permitir diferentes casos foi algo bastante mentalmente desafiador, e que necessitou de mais papel e lapis do que propriamente bater teclas. A única figura que incluímos neste relatório foi alias criada com a intenção de nos ajudar a perceber como deviamos resolver este problema.

O projecto proposto pelos docentes da disciplina parecia, inicialmente, algo bastante complicado mas com um bocado mais de estudo e paciência começamos achar cada vez mais que esta era o projecto mais desafiante e engraçado que nos tinham proposto até hoje. Este projecto não possui a dimensão inerente a um trabalho de Laboratórios de Informática, mas à nivel de diversificação dos conceitos a utilizar não lhes fica nada atrás.

A resolução de muitos dos problemas de este enunciada exigiam muito mais tempo de investigação e pesquisa do que programação propriamente. Um exemplo perfeito de um problema cuja a programação é trivial mas que nos demorou algum tempo até conseguirmos descobrir a solução foi a resolução do problema de como estabelecer uma comunicação entre dois processos. A maneira como enfrentamos e resolvemos este problema encontra-se devidamente relatada num capítulo deste relatório.

Apesar de esta exigência adicional ao nivel de investigação e pesquisa, foi bastante divertido andar a trabalhar com forks e processo pai e processos filhos e saber que sinais existem e como funcionavam e depois no fim ver todos este conhecimentos serem utilizados.

Nem sempre, no entanto, esta experiência foi agradável.Depois de termos sido introduzidos à linguagem java este semestre e de comodamente nos habituarmos a trabalhar com strings em Java. Trabalhar com strings em C tornou-se numa experiência bastante aborrecida e frustante em que é necessário varios truques para poder fazer o trim a uma string completamente por exemplo.

No final, nós achamos que esta foi uma experiência muito positiva e que nos deu novas ferramentas e mecanismos para resolver os diversos problemas que ainda teremos de enfrentar no futuro. Mas talvez, mais que as novas ferramentas obtidas ao nível da programação, fora os conhecimentos relativos ao funcionamento da nossa máquina a um nível bastante baixo que nos marcaram mais e que iremos reter não só deste trabalho prático como da disciplina.