

Universidade do Minho

Mestrado Integrado em Engenharia de Telecomunicações e Informática

Sistemas Operativos

Grupo 4



Relatório do Trabalho Prático



Universidade do Minho
Escola de Engenharia

Professor Francisco Moura

Professor Vitor Fonte

	Ana Carolina Monteiro da Silva a81127
	Carlos Jorge Teixeira Machado a81890
	Hugo Daniel da Costa Cunha Machado a80362

21 de dezembro de 2018

Introdução

No âmbito do trabalho prático da unidade curricular de Sistemas Operativos elaboramos este relatório para explicar o nosso raciocínio no desenvolvimento do trabalho até obtermos o resultado final do nosso projeto.

Para fazer este trabalho recorremos às bases dos exercícios dos guiões práticos, ao livro *Advanced Programming in the Unix Environment* de W. Richard Stevens e por fim como header files usamos: `<sys/wait.h>`, `<sys/types.h>`, `<sys/stat.h>`, `<unistd.h>`, `<time.h>`, `<fcntl.h>`, `<stdlib.h>`, `<stdio.h>`, `<signal.h>`, `<string.h>`, `<stdbool.h>`, `<errno.h>`.

Noções iniciais

No nosso programa, a interpretação de comandos no servidor e no cliente é feita da seguinte forma cada comando inserido em cliente é enviado para o servidor antes de qualquer funcionalidade, para tanto o cliente como o servidor estarem na mesma “página”.

É necessário também ter em conta que para este projeto uma tarefa pode assumir três estados, 0- cancelado, 1-ativo e 2-executado, definimos igualmente ‘\$’ como carater de finalização de dados, em algumas funções.

Em cliente temos uma estrutura *taskStruct*, que tem os parâmetros da tarefa. Abrimos em modo de escrita o *fifo clienttoserver* e em modo de leitura o *fifo servertoclient*, ambos os *fifos* são criados em servidor.

Funcionalidade de execução dos programas

O início da execução dos programas começa quando a interrupção de alarme é acionada, que nos leva para a função *sigalrm_handler()*. Nesta função chamamos uma outra função para converter a *string* que contém os argumentos para a execução num duplo apontador do tipo *char*, criamos dois *pipes* e redirecionamo-los para o *stderr* e o *stdout*, anteriormente criamos um processo filho para o *exec* e no processo pai ativamos o sinal para o fim do filho, o pai segue para a *main()* enquanto o sinal não é acionado.

Quando o sinal é acionado é feito um *wait(&status)* para receber o fim do filho e de seguida lemos dos *pipes* do *stderr* e *stdout* para as respetivas variáveis da estrutura, fechamos os *pipes* e é escolhida a próxima tarefa a executar.

[-a] Agendar uma tarefa

Para agendar uma nova tarefa é necessário introduzir a mesma em cliente e após a sua configuração enviamos a sua estrutura para o servidor, e o servidor retorna o id da tarefa.

Este processo funciona da seguinte forma, inicialmente verificamos se o número de argumentos recebidos é maior ou igual a cinco (número de argumentos mínimos do tipo de agenda pedida), após essa verificação, confirmamos também se a data inserida é superior à data atual, posto isto atribuímos os argumentos inseridos à *Taskstruct* a partir da função *newTask()* e escrevemos no *fifo* de escrita essa estrutura.

Caso o número de argumentos ou a data não se admitam, então escrevemos no *stdout* que o comando ou argumento foi invalido.

No servidor incrementamos o tamanho da estrutura e atribuímos um *id*, sequencial, à nova tarefa e guardamos no *array* de estruturas do servidor, de seguida verificamos qual a próxima tarefa a ocorrer e acionamos o alarme para o tempo dessa tarefa.

[-l] Listar as tarefas

Em listar as tarefas o cliente vai receber a informação vinda de servidor e apenas serão listadas as tarefas ativas e as executadas.

No servidor temos a função *printTasks()*, que vai enviar para o cliente o número de tarefas, se este for diferente de 0.

Se essa tarefa apresentar o estado como ativo vamos escrever para o *fifo* de escrita o cabeçalho dessa tarefa. Se não, se o estado estiver como executado escrevemos também para o *fifo* de escrita o cabeçalho dessa tarefa. Já em cliente lemos o conteúdo no *fifo* de leitura e escrevemos no *stdin*.

Se o *size* for igual a zero não existe nenhuma tarefa apenas enviamos o carater de finalização ('\$').

[-c] Cancelar uma tarefa

Em cancelar uma tarefa o cliente envia o *id* da tarefa a ser cancelada, para o servidor através de um *fifo* aberto para escrita.

Já em *servidor* verificamos se o *id* que recebemos do cliente é um número válido e corresponde com um *index* da lista de estrutura em server.

Se a tarefa que pretendemos cancelar não for a próxima a ser executada, apenas colocamos o seu estado.

Se a tarefa que pretendemos cancelar for a próxima a ser executada, colocamos também o seu estado a 0 e ativamos o alarme para a próxima tarefa a ser executada, no caso de não existir uma tarefa seguinte àquela que queremos cancelar, desativamos o alarme (*alarm(0)*).

[-r] Resultado de uma tarefa

Para o comando resultado de uma tarefa o cliente envia para servidor o *id* da tarefa da qual pretendemos ter acesso ao seu resultado e espera que o servidor lhe envie a tarefa associada.

Em servidor se houver correspondência com o *id* e se o seu estado esteja como executado enviamos através do *fifo* para o cliente a estrutura da respetiva tarefa.

Em cliente é recebida a estrutura e a sua informação é disposta como pedido no enunciado, no *stdout*.

Se não houver correspondência com o *id* ou o estado não estiver como executado, é retornada uma estrutura com o *id* igual a -1 e sem conteúdo relevante.

Novamente em cliente se o *id* da estrutura recebida for diferente de -1 colocamos numa variável a informação recebida da tarefa e imprimimos no *stdout* o conteúdo da variável.

Caso o *id* recebido da estrutura recebida seja igual a -1 imprimimos uma mensagem de erro.

[-e] Envio de resultados de execução

No envio de resultados executados o cliente envia para o servidor o mail do destinatário, e no servidor é executado o processo de envio dos mails com os resultados.

Em servidor percorremos a lista de estruturas verificando quais já foram executadas e, para cada executada criamos um processo filho, onde redirecionamos a leitura do *stdin* para o *pipe* anonimo e para fazer o *exec*. O processo pai vai escrever o conteúdo da tarefa para o *pipe* anonimo, que posteriormente será usado pelo processo filho.

[-n] Envio de resultados de execução

Esta funcionalidade não foi implementada por causa de uma implementação de código menos eficiente.

Contudo o nosso raciocínio para esta funcionalidade seria ter um *array* de inteiros para cada índice das tarefas a executar com um inteiro que corresponde ao limite de numero de tarefas concorrentes e da seguinte forma ter também um *array* de *pipes* para o *stdout* e o *stderr*, criar uma nova variável na estrutura para o *id* do processo.

Na função *sigalrm_handler()* teríamos um ciclo *for* para fazer *forks* até ao numero limite de tarefas, e na *sigchld_handler()* teríamos um *wait(&status)* que devolveria o *id* do processo terminado e aí comparávamos com os *ids* dos processos das estruturas de forma a obter o índice da estrutura recebida e assim colocar os respetivos *stdout* e *stderr* na estrutura da tarefa correspondente.

Desta forma em princípio iríamos resolver o problema proposto.

Conclusão

Este trabalho ajudou o grupo a consolidar, entender e a implementar da melhor os assuntos abordados nas aulas, bem como melhorar o nossa capacidade e domínio na linguagem C.

Inicialmente começamos por abordar o código através de listas ligadas, mas depois vimos que a maneira mais eficaz seria utilizar um *array* de estruturas.

Tivemos uma maior dificuldade em tratar do agendamento, e enviar um *mail*. Infelizmente não foi possível implementar o último tópico do enunciado, definir limite do número de tarefas executadas concorrentemente

Consideramos que apesar de todas as dificuldades que tivemos na construção do código e a falta da implementação do último tópico, obtivemos sucesso.