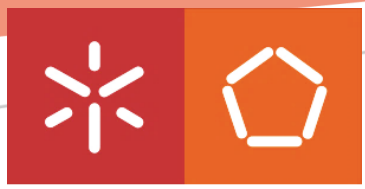


Universidade do Minho
Escola de Engenharia

SISTEMAS OPERATIVOS

RELATÓRIO TRABALHO PRÁTICO

Controlo e Monitorização de Processos e Comunicação

Grupo 71

Ana Luísa Carneiro A89533
Ana Rita Peixoto A89612
Luís Miguel Pinto A89506

ÍNDICE

INTRODUÇÃO

O trabalho prático "Controlo e Monitorização de Processos e Comunicação" propõe a implementação de um serviço de monitorização de execução e de comunicação entre processos.

Nesse sentido, utilizaram-se princípios fundamentais de Sistemas Operativos como o acesso a ficheiros, gestão processos, execução de programas, redireccionamento de descritores de ficheiros, pipes anónimos, pipes com nome, sinais, entre outros…

No decorrer do desenvolvimento deste projeto, foram surgindo alguns desafios, os quais passamos a enumerar:

- O primeiro desafio consistiu em perceber como é que iria funcionar a comunicação entre o cliente e o servidor, já que a comunicação entre FIFOS só funciona num sentido. Para além disso era também necessário que toda a informação gerada no servidor fosse redirecionada para o cliente, e vice-versa.
- Surgiu um outro desafio, em particular para funcionalidades como o tempo de inatividade e o tempo de execução, que foi descobrir uma estratégia para que conseguíssemos, durante a execução de uma tarefa, fazer uso de dois alarmes em simultâneo, já que um reiniciava o do outro.
- Também consideramos que foi desafiante construir este projeto em função da execução em paralelo, assim enquanto que uma tarefa corre em background é possível que o servidor fique disponível para executar outras funcionalidades.

FUNCIONALIDADES IMPLEMENTADAS

Cliente e Servidor

No nosso programa, tal como era pedido, foram implementadas duas entidades: o Cliente – **argus** – e o Servidor – **argusd**. O cliente é a entidade responsável pela comunicação entre o utilizador e o programa criado, ou seja, serve como intermediário entre o utilizador e o servidor. No servidor encontra-se toda a camada de dados implementada para dar resposta às funcionalidades do programa. Assim é possível que, enquanto o servidor corre em background, o argus recebe o *input* do utilizador que será enviado para o argusd que irá executar e enviar o seu *output* de volta ao cliente, criando assim a possibilidade de o servidor executar várias tarefas e funcionalidades em simultâneo.

A comunicação entre o cliente e o servidor é feita com o auxílio a FIFOS (*Named Pipes*). Para isso, foram criados dois FIFOS. O primeiro, **fifoCS**, faz a comunicação entre o Cliente e o Servidor, ou seja, envia a informação do argus para argusd de forma a este executá-la. O segundo FIFO, **fifoSC**, faz a comunicação inversa, isto é, envia toda a informação do argusd para o argus que será posteriormente enviada ao utilizador.

Diagrama Servidor Cliente

Funcionalidades Mínimas

Uma das primeiras funcionalidades implementadas foi a interface de comunicação entre o utilizador e o cliente, que foi feita de duas formas: via linha de comandos e via Shell (interpretação textual). Em primeiro lugar, foi estabelecida a interface via linha de comandos. Mais tarde, aquando da criação do método via Shell, adaptou-se esse método ao já existente. Desta forma, o comando "tempo-inatividade 10" foi transformado em "-i 10", podendo assim aproveitar os métodos de interpretação de comandos já implementados no servidor e no cliente.

Tempo-Inatividade

Esta funcionalidade permite ao utilizador definir o tempo máximo de inatividade de comunicação de um pipe anónimo.

A estratégia utilizada, foi implementada na função *execPipe*, função que executa uma tarefa recebida pelo comando "-e". Deste modo, quando a tarefa recebida é um comando ou vários, a função *execPipe* executa cada comando num *fork()*. Assim, o processo pai envia a cada processo filho um sinal *SIGUSR1* (através da chamada *kill)*, cujo handler executa um *alarm* com o tempo de inatividade definido. Assim, quando o *alarm* termina e é enviado o SIGALRM, são terminados todos os processos associados a essa tarefa.

O modo de teste desta tarefa, baseou-se na execução encadeada de programas cujo tempo de execução de cada tarefa individual demorasse mais do que o tempo de inatividade. Por exemplo, ao executar o comando *./argus -e "./sleep2 | ./sleep10"*, em que a chamada

./sleep2 e a chamada *./sleep10* demoram 2 e 10 segundos, respetivamente, e o tempo de inatividade está definido para 3, o processo é terminado após 5 segundos de execução.

Tempo-Execução

O tempo de execução permite ao utilizador definir o tempo de máximo de execução de uma tarefa. Por omissão, não existe restrição neste tempo.

Para implementação deste comando foi utilizada a *system call* **alarm**, que ao fim do tempo máximo de execução proposto pelo utilizador, vai lançar um sinal – SIGALRM. Após o programa receber este sinal, este vai "matar" todos os processos que foram criados durante a execução da tarefa e assim parar a execução desta.

Como efeito de testar a funcionalidade, criou-se um programa *sleep* que fica adormecido durante 40 segundos. Assim, se o utilizador definir um tempo máximo de execução de 10 segundos, ao executar o programa *sleep,* o programa termina ao fim de 10 segundos (ao invés dos 40).

Executar

Este comando permite ao utilizador executar uma tarefa, quer com encadeamento de comandos *("ls | wc -l"),* quer com uma única instrução *("ls"),* e devolve o output dessa tarefa assim que esta termine. Em ambos os casos foi utilizado o conceito de **fork** para que o filho fosse responsável pela execução da tarefa deixando o pai livre para executar outras funcionalidades, contribuindo assim para a execução simultânea no servidor.

Ao executar uma determinada tarefa, o processo pai vai colocar essa tarefa num array estático (*tarefas*) responsável por guardar todas as tarefas que foram executadas e/ou que estão em processo de execução. Para além disso, também vai guardar num outro array estático (*estados*), no mesmo índice, o estado da tarefa que inicialmente será igual a 0 (0 – em execução e 1 – executado). O processo filho vai ser responsável pela respetiva execução do comando através da função *execPipe*. Após a execução, o filho vai devolver como código de terminação o número da tarefa que acabou de executar, isto é, o índice nos arrays.

Enquanto o filho está em execução o processo pai vai ainda guardar num array estático (*processos*) o pid do processo filho, para que na opção terminar seja possível saber qual o processo a "matar". Para além disso, logo que o filho termine o pai vai receber um sinal – SIGCHILD – e vai executar a respetiva rotina de tratamento, que será receber o código de terminação do filho, para que este não se torne zombie e mudar o estado do índice que recebeu do filho no array de estados para o valor 1.

Os testes feitos como forma de verificar a correta implementação desta opção, foram simplesmente verificar se o output da tarefa era o correto e se era dado no cliente, verificar se era possível a execução em simultâneo de várias tarefas (e se isso não impedia a execução de outras funcionalidades do programa), e finalmente, se as tarefas terminavam corretamente, ou ao fim de um tempo máximo definido para execução ou inatividade, ou ao fim do tempo por default.

Listar

Nesta opção é possível mostrar ao utilizador todas as tarefas que estejam em execução naquele momento. Para isso, foram utilizados os dois arrays estáticos já mencionados na opção executar – *tarefas* e *estados*. Este comando vai percorrer o array dos estados até encontrar uma tarefa a 0. De seguida, o servidor devolve ao cliente, através do FIFO, a tarefa que está no mesmo índice em que foi encontrada o estado a 0.

Como teste utilizou-se novamente o programa sleep, que está adormecido durante 30 segundos (por exemplo). Se este programa for executado em background e em foreground executarmos a funcionalidade listar, então o programa sleep estará na lista de programas em execução apresentada ao utilizador.

Terminar

A opção terminar permite ao utilizador finalizar a execução de uma determinada tarefa (que ainda esteja por concluir). Para a sua implementação foram utlizados os arrays estáticos já mencionados na opção executar – *processos* e *estados*. Assim, dependendo da tarefa que o utilizador quiser terminar, o programa vai dizer se a opção dada é válida para ser terminada ou não, ou seja, se o utilizador quiser terminar uma tarefa já concluída o servidor devolve uma mensagem de erro (por exemplo). Após o programa verificar se a tarefa é válida, este vai ao índice respetivo no array dos pids e vai com uso da função **kill** terminar o processo pretendido e mudar no array dos estados o valor de 0 para 1.

Para testar esta funcionalidade foi novamente utilizado o programa sleep. Após colocar esse programa a executar em background, executou-se a opção terminar para finalizar a tarefa do sleep. Logo de seguida, verificou-se se o programa tinha sido terminado através do histórico que continha esse programa na lista dos já concluídos e através do listar que não continha na lista dos processos ainda por execução.

Histórico

Neste comando é possível mostrar ao utilizador todas as tarefas que já tenham sido executadas, isto é, o histórico de todas as tarefas executadas e terminadas. Assim, tal como se fez para a opção listar, utilizou-se dois arrays estáticos – *tarefas* e *estados*. Este comando vai percorrer o array dos estados até encontrar uma tarefa a 1. De seguida o servidor devolve ao cliente, através do FIFO, a tarefa que está no mesmo índice em que foi encontrada o estado a 1.

Como forma de testar a funcionalidade executou-se o comando “ls”, pois a sua execução é praticamente imediata. Assim, logo após o fim da execução deste comando, testou-se a funcionalidade histórico que irá listar ao utilizador todas as tarefas concluídas, incluindo o comando “ls”. Contudo, se executarmos novamente o programa sleep em background, este não se encontra na lista de histórico pois ainda não é uma tarefa concluída.

Ajuda

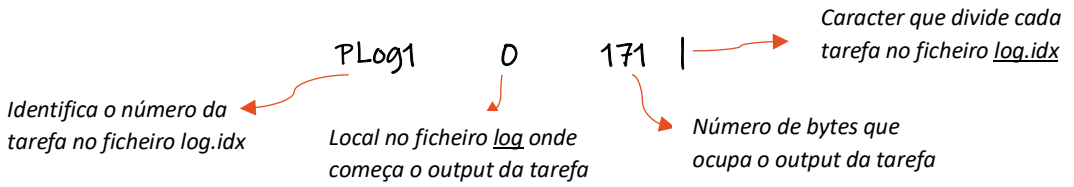
O comando *ajuda* fornece informação sobre as funcionalidades existentes e como executá-las, através de sucessivos prints que formam um pequeno menu informativo.

Funcionalidade Adicional

Output

Esta funcionalidade permite o utilizador consultar o *output* de uma dada tarefa executada anteriormente.

Tal como proposto pelo enunciado, a implementação desta funcionalidade basou-se em guardar num ficheiro *log* o output de cada tarefa, aquando da execução da mesma, e no ficheiro *log.idx* informações sobre a consulta do ficheiro *log.* A título de exemplo, por cada tarefa executada, é guardada uma *string* no ficheiro *log.idx*:



Deste modo, quando é executado o comando *./argus -o 2*, por exemplo, é lido o ficheiro *log.idx* e é feita a procura até encontrar as respetivas informações da tarefa, no estilo especificado anteriormente, ou seja, identificadas por *PLog2* e até ao caracter de divisão. Após encontrar a respetiva *string* da tarefa no *log.idx*, é feita a leitura do ficheiro *log*, de modo a retornar o output correspondente.

De forma a testar a implementação desta funcionalidade, foram executadas várias tarefas com o auxílio do comando "-e", e de seguida, a invocação do comando "-o" da tarefa pretendida.

CONCLUSÃO

Dado por concluído o nosso projeto, apresentamos uma reflexão sobre possíveis melhorias, assim como aspetos a valorizar.

Em primeiro lugar, apresentamos aspetos que podiam ser melhorados, como o fazer e as consequências dessas implementações:

- No caso da opção executar, quando o utilizador introduz um comando inválido, -e "25" (por exemplo), o programa apesar de não ter nenhum output e de não crashar, vê esse comando como uma nova tarefa. Assim, seria mais benéfico se o servidor mandasse uma mensagem de erro "Comando inválido".
- A maioria dos arrays implementados são estáticos de maneira a não ser necessário alocar e libertar memória. Contudo, em alguns casos essa opção pode não ser a mais viável, pois sendo o tamanho limitado teremos problemas se o limite for ultrapassado. Mesmo definindo um limite grande essa não será a melhor implementação, e por isso deveríamos utilizar arrays dinâmicos.

Em segundo lugar, reconhecemos que existem também aspetos positivos, os quais passamos a enunciar:

- Implementação de todas as funcionalidades pretendidas, ou seja, tanto as funcionalidades mínimas como a funcionalidade adicional. Bem como ter duas interfaces operacionais.
- Uso eficiente de rotina de tratamento de sinais para o SIGCHILD, SIGUSR1, SIGALRM, que permitem um bom funcionamento do trabalho.
- Criação de diferentes estruturas capazes de: guardar comandos de execução, o estado de execução dos mesmos, e ainda os pids dos seus processos. Conseguindo assim obter uma solução eficiente para diversas funcionalidades.
- O programa suporta programação concorrente através de conhecimentos obtidos nas aulas de SO principalmente do conceito de fork, ou seja, enquanto um processo filho é responsável pela execução de uma tarefa, o processo pai está disponível para a execução de outras funcionalidades.

De um modo geral, achamos que o trabalho cumpriu com os requisitos propostos e apesar de haver melhorias, achamos que o balanço é positivo.