Universidade do Minho
Mestrado Integrado em Engenharia Informática

Sistemas Operativos

GRUPO:

Ana Teresa Gião Gomes - A89536
Maria Quintas Barros - A89325
Maria Beatriz Araújo Lacerda - A89535

14 de Junho 2020

Figure 1: A89536

Figure 2: A89525

Figure 3: A89535

1 Introdução

No âmbito da UC de Sistemas Operativos, foi-nos proposto a elaboração de um projeto de monitorizaçao de execução e de comunicação entre processos. O programa terá que ser capaz de interpretar diversas tarefas submetidas pelo utilizador tais como definir tempo de inatividade, tempo de execução, executar e terminar uma tarefa, listar tarefas em execução e o histórico das já terminadas e ainda oferecer ajuda ao utilizador.
Para além destas funcionalidades, foi-nos sugerido uma outra adicional que consiste em consultar o standard output de uma tarefa executada.
Naturalmente, sendo um dos objetivos do trabalho usarmos um baixo nível de abstração, este foi desenvolvido em C.

2 Cliente

No cliente começamos por criar um fifo que irá ler as informações do servidor (fifo_resposta em modo O_RDONLY) e um fifo que irá enviar os comandos para o servidor para serem executados(fifo_fd em modo O_WRONLY)
Se o cliente inserir apenas um argumento (ex: ./argus) então irá iniciar o argus, onde é possível inserir vários comandos. Para isto fizemos um ciclo que escreve no fifoo que o utlizador escreveu no buffer criado com standard stdin (teclado).
Em seguida, de maneira a apresentar os resultados no terminal do cliente, vamos ler aquilo que nos é ennviado pelo fifo_resposta e escrever no standard output (ecrã).
Se o cliente introduzir o comando diretamente (ex: ./argus -r) o processo em cliente.c é semalhante(escrever o comando para o fifo_fd e escrever no ecrã as respostas dadas por fifo_resposta)

3 Servidor

No servidor começamos por criar:

1. Um fifo_resposta em modo O_WRONLY que irá escrever o resultado dos comando para enviar para o terminal do cliente.

2. O ficheiro log_fd, onde vamos escrever os resultados dos nossos comandos.

3. Um ficheiro log_idx

Vamos, de seguida, abrir o fifo_fd em dod de leitura O_RONLY para ler o comando enviado pelo cliente. Em seguida vamos "dissecar" o que lemos no fifo, separando o tipo do comando (comando) e o que a função tem que executar (ex: no comando executar -ls, comando será exexutar e valor será -ls).

- **Caso 1: Definir o tempo máximo de inatividade**
  Define a variável global max-inat com o valor passado pelo cliente (valor), que é passado se string para int através da função atoi. Em seguida, invocamos a função (write (fifo_resposta,"",1);), visto que a função read é bloqueante e portanto é necessário para manter o fifo aberto. Isto é repetido em todos os nossos outros casos.

- **Caso 2: definir o tempo maximo de execução**
  Fizemos um processo muito semelhante ao caso anterior, desta vez alterando a variável global max-exec.

- **Caso 3: Executar uma tarefa**

  Começamos por eliminar as plicas através de um shift à esquerda. Depois contamos o número de pipes através da função isPipe que conta quantos " — " estao na string valor.
  Se o comando não tiver pipes é chamada a função mysystem que isola e executa o comando.
  Se, por outro lado, o comando apresentar um ou mais pipes, será primeiramente invocada a função "dividePipes" que vai popular o array pipes com as diferentes funções a serem executadas. Em seguida passamos esse array para a função "executa" que fará precisamente isso. Ao contrário da comunicação entre o servidor e o cliente que é feita atráves de pipes com nome, para executar os comandos introduzidos pelo cliente criamos pipes anónimos. A nossa função executa irá variar, dependendo do número de pipes presentes no comando.
  A nossa função cobre os 3 casos possíveis: pipe inicial(se i=0),

pipe final(i=n-1) e pipes intermédios. No pipe inicial, se for filho, fechamos a leitura, redirecionamos a extremidade da escrita, fechamos esta e executamos o comando. Se for pai, espera pelo filho, e adiciona os pids dos filhos a um array. No pipe final, fechamos a extremidade de leitura do pipe e redirecionamos a extrimidade da escrita para o log_fd. Nos pipes intermédios, mantemos ambas as extremidades dos pipes abertas, redirecionamos estas e executamos os comandos.
Finalmente, adicionamos esta tarefa à nossa lista ligada que contem as tarefas em execução.
Neste código, implementamos dois tipos de alarmes. Um alarme de tempo de execução, implementado na função mysystem e executa e o outro alarme, ou seja de inatividade apenas na função executa. Estes sinais servem para evitar casos de infinita execução. Assim é estabelecido um valor máximo dos mesmos.

1. **Alarme de inatividade** - Este termino apenas ocorre quando o programa fica eternamente à espera de um processo-pai. Posteriormente é colocado 3 alarmes, um em cada caso da execução. No primeiro caso, quando é o primeiro comando a ser executado, o alarme é inicializado. No segundo caso, ou seja quando é o último comando o alarme é redefinido para 0. E finalmente o último caso que ocorre quando os comandos intermédios, onde o alarme é redefinido para alarm(0). Sempre que o programa fica à espera de um processo-filho, ele adiciona a um array de pids, o pid correspondente. No caso do alarme tocar, o sinal é ativado e o mesmo mata todos os processos que estão no arrays de pids.

2. **Alarme de execução** - Este termino apenas ocorre quando o tempo de execução do comando é superiror ao tempo definido para tempo de execução. O mesmo é ativado antes do último fork para a execução de comandos.

- **Caso 4: Executar uma tarefa** Este caso executa o mesmo que o nosso caso 3, mas sem a entrada no cliente (ex: ./argus -e). A razão de criarmos um caso separado é o de nesta segunda situação o comando não ter plicas, e logo, eliminamos o ciclo que tratava disto.

- **Caso 5: Listar tarefas em execução**
  Neste caso invocamos a função "linkedToString" que imprime a nossa lista ligada de tarefas em execução.

- **Caso 6: Terminar uma tarefa em execução**
  Começamos por invocar a função "elementoNaOrdem" que se encontra em slist.c, entre as outras funções feitas para a implementação. Esta função devolve-nos o pid da função de ordem escolhida.
  Em seguida terminamos o processo com esse pid através da função "kill".

- **Caso 7: Listar registo historico de tarefas terminadas**
  Para este objetivo invocamos a função "linkedToStringHistorico" que vai imprimir o conteúdo da nossa lista ligada com as tarefas que já não se encontram em execução. Antes de imprimir a função vai verificar o código de terminação do processo (um dos dados da nossa lista ligada que nos é oferecido pelos exit() dos sinais).

- **Caso 8: Apresentar ajuda a sua utilização**
  Para isto, imprimimos a função "ajudaMenu", composta exclusivavemnte por snprintf.
  Utilizamos o snprintf visto que esta função insere a nossa informação num buffer, que é posteriormente returnado pela nossa função e mais tarde escrito no terminal do cliente.

- **Caso 9: Funcionalidade Adicional**
  Para este tópico, criamos um ficheiro log.idx onde adicionamos informação sempre que executamos um comando. Posteriormente, através de uma chamada ao sistema lseek, conseguimos obter os tamanhos da informação desejada. Nesta situação existem dois casos:

  1. **Deseja a informação de ordem 1** - Aqui, colocamos o apontador tanto no ficheiro *idx* como no ficheiro *txt*. Posteriormente lemos a informação dada no ficheiro *idx*, isto é, o número de caracteres da informação pretendida e lemos também a informação no ficheiro *txt* para um buffer do tamanho obtido pela leitura do ficheiro anterior. Finalmente escrevemos no fifo_resposta o buffer obtido.

2. **Deseja a informação de ordem superior a 1** - Desta vez, colocamos o apontador do *idx* no final da anteriror à ordem desejada. Posteriormente, lemos os endereços do início da ordem desejada e do fim da anterior. Assim obtemos o tamanho da informação que futuramente iremos ler. De seguida, colocamos o apontador do ficheiro *txt* do valor início obtido pela leitura do ficheiro *idx* e iremos ler a informação de tamanho (fim-inicio+1) para o buffer. Finalmente, escrevemos toda a informação do buffer no fifo_resposta.

4 Conclusão

Em primeiro lugar, o desenvolvimento deste projeto ajudou-nos a desenvolver e pôr em prática os conhecimentos adquiridos durante as aulas práticas de Sistemas Operativos durante este segundo semestre. Por outro lado, pudemos observar que a programação neste nível de abstração é bastante exigente na guestão de erros e validação de inputs.
Apesar da dificuladade deste trabalho, consideramos que a sua realização foi elaborada com sucesso, uma vez que conseguimos responder a todas as funcionalidades pedidas.