FCTUC FACULDADE DE CIÊNCIAS
E TECNOLOGIA
UNIVERSIDADE DE COIMBRA

Practical Assignment Nº1 - Processes, Files, Pipes, FIFOs, I/0 Redirection, Unix Sockets and Signals

Relatório realizador por:

David Oliveira, nº2021221929

João Cunha, nº2020252382

Pl1, Grupo 1

Unidade Curricular de Sistemas Operativos
Licenciatura em Engenharia Eletrotécnica e de Computadores – 2023/2024

Problema 1:

O principal objetivo deste problema era conseguirmos executar a seguinte linha de código shell em C “find . -type f -ls | cut -c 2- | sort -n -k 7 >file.txt ; less <file.txt”

Para a primeira pipeline, entre o “find” e o “cut” usamos um socket e para a segunda pipeline entre o “cut” e o “sort” usamos um pipe, depois usamos o “fork()” para criar um processo filho que executa o respetivo comando de cada processo filho. Em cada processo filho, as entradas e saídas são redirecionadas conforme necessário usando “dup2()”, por fim os comandos “find”, “cut” e “sort” são executados em seus respetivos processos filhos. O processo pai espera que todos os processos filhos terminem usando “waitpid()” e o socket criada pelo processo “find” é removido após o fim do programa.

O processo filho do “find” encontra todos os arquivos regulares a partir do diretório atual, utiliza um socket para redirecionar a sua saída para o processo pai, que depois passa para a próxima parte do pipeline, ou seja, usamos “socket(AF_UNIX , SOCK_STREAM, 0)” para criar um socket, depois para configurar o socket com um endereço e vinculado ao mesmo usando “bind()”, que por sua vez usamos “listen()” para que o processo ouça as conexões do socket. Quando uma conexão é estabelecida pelo processo pai, o processo filho aceita a usando “accept()” e o descritor do socket é finalmente usado para redirecionar a saída do “find”. A saída do “find” é redirecionada para o socket usando a função “dup2()” e por fim executamos o “find”.

O processo filho do “cut” é responsável por receber a saída do processo “find”, de seguida filtrar a mesma e enviá-la ao processo pai através de um pipe. Em primeiro lugar o processo estabelece uma conexão com o socket criado pelo processo “find”, depois o stdin do processo filho é redirecionado para o socket através da função “dup2()”, o stdout é redirecionado para o pipe através também da função “dup2()”, finalmente fechamos o pipe e executamos o “cut”.

O processo filho do “sort” recebe a saída do “cut” através do pipe e antes de redirecionar para o “file.txt” ordena. Em primeiro lugar, o stdin do processo é redirecionado para o pipe usando “dup2()” e fecha o pipe de seguida, depois o arquivo “file.txt” é criado ou aberto usando a função “open()”, após a criação ou abertura do arquivo vamos redirecionar o stdout para o arquivo usando a função “dup2()”, de seguida fechamos o descritor do arquivo e por fim executamos o sort usando “execlp()”.

O pai deve fechar os seus descritores do pipe para não interferir nas comunicações entre os filhos. Após os três processos filhos, o processo pai espera que os três processos filho terminem através da função “waitpid()”, de seguida remove o socket através do “unlink()” e por fim executamos o comando “less” com redireccionamento de entrada do arquivo “file.txt”.

Concluindo, cada processo filho desempenha um papel específico na execução da linha de comando referida inicialmente, eles são responsáveis por executar os comandos, redirecionar a entrada e a saída conforme necessário e garantir a comunicação adequada entre si e com o processo pai.

Problema 2:

Neste problema tínhamos que reutilizar o código do problema 1 mas desta vez tínhamos que usar FIFOs e modificar o programa para que o Pipeline 2 fosse implementado usando named pipe através do mknod(), mkfifo() em vez de pipe.

Em primeiro lugar, criamos um FIFO usando a função “mkfifo()” para que este FIFO seja usado na segunda pipeline entre os processos “cut” e “sort”, depois criamos três processos filhos a partir da função “fork()”, um para cada comando na linha original de comando referida no início do problema 1 (“find”, “cut” e “sort”) e explicado anteriormente no problema 1 cada processo.

A saída do processo filho do “find” vai ser redirecionada para o processo pai através do socket. O processo filho do “cut” estabelece uma conexão com o socket criado pelo processo “find” e redireciona o seu stdin para o socket, abre o FIFO para escrever, também redireciona o stdout para o FIFO através da função “dup2()” e executa o "cut". O processo filho do “sort” abre o FIFO para leitura e redireciona o stdin para o FIFO, abre o arquivo “file.txt”, também redireciona o stdout para o arquivo “file.txt” e finalmente executa o “sort”.

Após os três processos filhos, o processo pai espera que os três processos filho terminem através da função “waitpid()”, de seguida remove o socket e o FIFO através do “unlink()” e por fim executamos o comando “less” com redireccionamento de entrada do arquivo “file.txt”.

Concluindo, a solução mostrada anteriormente implementa com sucesso a execução da linha de comando original, na parte da segunda pipeline utilizamos o named pipe (FIFO) para a comunicação entre os processos “cut” e “sort”. Cada processo redireciona corretamente a sua entrada e saída garantido um bom funcionamento da shell command line.

Problema 3:

Comunicação entre dois programas “agente.c” e “controller.c” através de Sinais.

O programa “agente.c” começa por criar um ficheiro “agente.pid” para guardar o seu “Process ID”, de forma que outros processos lhe possam enviar sinais. Depois abre o ficheiro “text.in” e escreve o seu conteúdo para o terminal (stdout).

No próximo passo o programa prepara-se para apanhar os sinais “SIGHUP” e “SIGTERM” através da função “signal()” e redireciona-os para as funções “catch_HUP” e “catch_TERM” respetivamente.

Por fim fica à espera de receber um sinal, “pause()” num loop infinito.

catch_HUP: abre o ficheiro “text.in” e escreve o seu conteúdo para o terminal (stdout).

catch_TERM: escreve “Process is terminating…” no terminal usando “printf()”, elimina “agente.pid” com “unlink()” e termina o programa através de “exit()”.

O programa “controller.c” abre o ficheiro “agente.pid”, lê o PID lá guardado e verifica se o processo “agente.c” está vivo ao enviar um sinal 0 “kill(pid_agent, 0)”, se a função “kill()” retornar 0 significa que o processo existe.

No final é criado um loop para perguntar ao utilizador que sinal deseja enviar para o “agente.c”, “SIGHUP”(1) ou “SIGTERM”(15). Caso o utilizador escolha (1), o sinal “SIGHUP” é enviado e volta para o inicio do loop. Caso o utilizador escolha (15), o sinal “SIGTERM” é enviado é terminado o processo através de “exit(0)”.

Antes do programa entrar no loop do menu é necessário interromper o sinal “Ctrl+C” e redireciona-lo para o função “catch_INT()”, onde irá enviar o sinal “SIGTERM” para o processo “agent.c” através de “kill()” e terminar o programa com “exit(0)”.

Problema 4:

Transporte de uma tabela de inteiros entre Pai e Filho através de um ficheiro binário “filetable.bin”.

É definido o tamanho da tabela para 10 usando “#define MAXBUF 10”.

Preenchemos a tabela com um ciclo “for()” de 10 iterações com “tabela[i] = i +1;”.

É aberto, ou criado caso não exista, o ficheiro para guardar a tabela em modo escrita através de “open(“filetable.bin”, O_WRONLY | O_CREAT, 0666)”. Usamos outro ciclo “for()” para escrever a tabela no ficheiro, inteiro a inteiro. Após concluído fecha o ficheiro com “close()”.

Utilizamos um “fork()” para criar um filho que irá ler o ficheiro binário. Abrimos o ficheiro em modo leitura “open(“filetable.bin”, O_RDONLY)” e lemos para uma tabela os 10 inteiro através de um ciclo “for()”. É usado novamente um ciclo “for()” para escrever a tabela do terminal (stdout).

Para o Filho descobrir se o Pai terminou é verificado através de “getppid()” se o PID do Pai muda. Logo é necessário registar numa variável “pid_t” o PID do Pai originalmente, para comparar com o PID do Pai atualmente. Quando o Pai termina, o Filho deve também terminar e escrever o terminal “The child process is terminating.”.

Apos criar o Filho, o Pai deve terminar e escrever no terminal “The parent process is terminating.”.