Relatório do trabalho prático nº 1

Primeiro tópico: Implementação

O objectivo deste trabalho era criar um servidor que fosse capaz de simular as acções de uma impressora e receber pedidos de clientes para as concluir. Para permitir a transferência de informação entre o servidor e o cliente foram utilizados os ficheiros *"mysocks.h"* e *"mysocks.c"*, recorrendo assim, ao uso de chamadas ao sistema operativo que permitissem o uso de *sockets*. Segue abaixo uma descrição dos dois programas.

Cliente

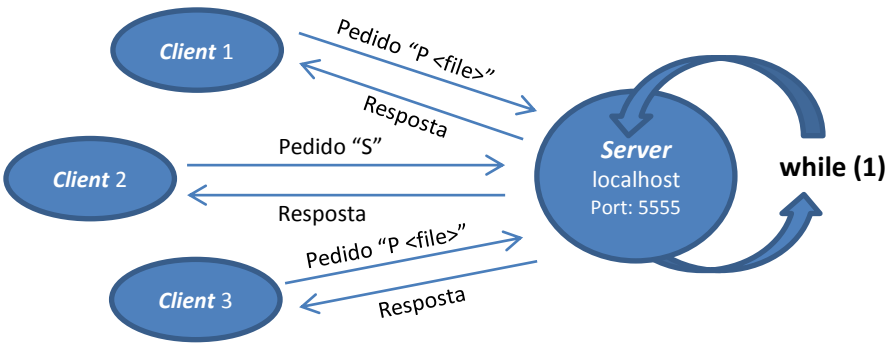
O funcionamento do cliente foi baseado nos exercícios desenvolvidos nas aulas práticas e a sua implementação consiste num sistema de interpretação contínua de comandos inseridos pelo utilizador (semelhante a uma *shell*). Possui, desta forma, a capacidade de reconhecer um conjunto de comandos específicos - com o auxílio de uma função *makeargv* - que podem ser transmitidos ao servidor.

Sequência de execução do cliente:

1. Entra no ciclo de leitura de comandos do utilizador;
2. Interpreta o comando inserido utilizador:
   a. Se o comando for "status", coloca num vector denominado *request* a mensagem "S", para que o servidor devolva o número de ficheiros em lista de espera para impressão;
   b. Se o comando for "print <file>", coloca no vector referido no ponto anterior a mensagem "P <file>", de modo a que o servidor imprima o ficheiro com o nome especificado;
   c. Se o comando for "quit", termina o programa;
   d. Se o comando não for nenhum dos referidos anteriormente, é colocado no vector *request* para o servidor o processar (o servidor irá responder que não reconhece o comando);
3. Tenta estabelecer ligação com o servidor e, se for bem sucedido, envia o pedido contido no vector *request*;
4. Espera pela resposta do servidor;
5. Imprime a resposta do servidor após recepção e fecha a ligação com o mesmo;
6. Volta a entrar no ciclo de modo a ler mais comandos do utilizador (ponto 1);

Servidor

O funcionamento do servidor consiste no uso de múltiplas *threads*, aplicando assim, o conceito de um servidor concorrente. Ao utilizar uma *thread* principal para interpretar os pedidos dos clientes e possibilitar-lhe a capacidade de lançar uma *thread* auxiliar sempre que algum pedido implique a impressão de um ficheiro, é conferida ao servidor a capacidade de poder interpretar e concluir os pedidos de vários clientes em simultâneo. Para aplicar esta abordagem, a *thread* principal encontra-se em ciclo infinito durante a maior parte da sua execução.



Exemplo da capacidade do servidor de processar pedidos de múltiplos clientes

Sendo o servidor concorrente e o objectivo do trabalho criar um simulador de impressões, o mesmo necessita da capacidade de impressão de múltiplos ficheiros. É preciso executar a impressão de um ficheiro de cada vez, de modo a que seja possível garantir que durante a mesma, mais nenhum ficheiro possa ser imprimido, criando assim uma ordem sequencial de impressões – uma lista de espera. A estrutura implementada para servir este propósito foi um *buffer* circular (e as funções respectivas), implementado através de uma *struct*, fornecido pelo ficheiro "*producer_consumer.c"* na página da cadeira e três variáveis essenciais para lidar com a concorrência entre as *threads* no método de impressão do ficheiro *print_file*. O *buffer* circular ao longo da execução do programa vai contendo os descritores dos ficheiros a imprimir e as variáveis *ready*, *lock* e *printing* (variável de condição, variável de *mutex*, e um inteiro, respectivamente) permitem com que a impressão de ficheiros seja sequencial, impedindo com que múltiplas *threads* tentem imprimir o seu próprio ficheiro ao mesmo tempo. O servidor possui ainda uma variável *queued (inteiro)* para ser possível determinar a quantidade de ficheiros que estão em lista de espera.

Sequência de execução da *thread* principal:

1. Inicializa o *buffer* circular e as variáveis *ready*, *lock*, *printing e queued.* As variáveis *printing* e *queued* são inicializadas com o valor '0';
2. Abre o pseudo-terminal especificado para escrita, caso ocorra algum erro termina o programa;
3. Cria a *socket* do servidor na porta fornecida pelo utilizador, caso ocorra algum erro termina o programa;
4. Entra no ciclo infinito à espera de ligações de clientes;
5. Aceita a ligação de um cliente;
6. Interpreta o pedido do cliente:
    a. Se for "S", responde ao cliente com uma mensagem contendo o número de ficheiros que estão em lista de espera para serem imprimidos, ou seja, o valor da variável *queued*;
    b. Se for "P <file>", abre o ficheiro referido, coloca o descritor do mesmo no *buffer* circular, incrementa a variável *queued* e lança uma *thread* para executar a função *print_file* que irá tratar da sua impressão. Por fim, envia uma mensagem ao cliente informando que o ficheiro especificado será colocado na lista de espera para impressão;
    c. Se for uma mensagem diferente das referidas anteriormente, responde ao cliente com uma mensagem informando que não reconhece o pedido;
7. Volta ao ínicio do ciclo e fica à espera de mais ligações (ponto 4).

Sequência de execução da *thread* auxiliar - impressão de ficheiros:

1. Cria um *buffer* de leitura do ficheiro;
2. Adquire o *lock* do *mutex,* permitindo com que seja a única *thread* a executar o trecho de código seguinte;
3. Verifica se uma *thread* já está a realizar a impressão de um ficheiro:
    a. Se *printing* = 0, não há nenhuma impressão a decorrer e continua;
    b. Se printing = 1, espera pela conclusão da impressão a decorrer;
4. Coloca a variável *printing* a '1' de modo a sinalizar que está uma impressão a decorrer;
5. Obtem o descritor do ficheiro a imprimir através do *buffer* circular;
6. Perde o *lock* do *mutex*;
7. Entra no ciclo de leitura do ficheiro enquanto este tiver *bytes* para ler;
8. Lê 1024 *bytes* do ficheiro em questão, ou menos se o conteúdo do ficheiro for inferior a tal;
9. Entra noutro ciclo que realiza a escrita *byte* a *byte* 1024 *bytes* já lidos.
10. Após terminado o ciclo de escrita volta a entrar no ciclo de leitura (ponto 7) ou caso a condição do mesmo não se verifique passa para o ponto 11.;
11. Fecha o canal de entrada/saída para o ficheiro ;
12. Adquire o *lock* do *mutex*, coloca a variável *printing* a '0' e decrementa *queued*;
13. Perde o *lock* do *mutex*;
14. Sinaliza que terminou a impressão e termina a sua execução. Qualquer *thread* que estiver em espera para impressão no ponto 3 pode agora executar o seu código correspondente;

Segundo tópico: Suposições feitas

1. Visto que no enunciado não é referida a necessidade de criar algum tipo de comando ou condição para o servidor terminar a sua execução, considerámos que a mesma dependeria de um ciclo infinito, sendo forçar o fecho do processo a única maneira de a terminar. Isto teve como consequência a limitação enunciada no ponto 2 do tópico seguinte;

2. Considerámos que quando um cliente executa o comando "*quit*" após enviar quaisquer pedidos de impressão para o servidor, o programa cliente termina, mas as impressões pedidas serão executadas pelo servidor de qualquer forma;

3. Quando um cliente executa o comando "*status*", considerámos que o servidor deveria devolver todos os ficheiros na lista de espera, incluindo o ficheiro que estiver a ser actualmente imprimido. Isto é, a impressão de um ficheiro só está realmente acabada quando a totalidade do conteúdo deste for escrita no pseudo-terminal, e só então é que o mesmo deixa de ser contabilizado na lista de espera.

Terceiro tópico: Possíveis limitações

1. O servidor invoca uma grande quantidade de chamadas ao sistema de modo a conseguir realizar as operações necessárias sobre ficheiros, ou seja, é bastante intensivo em operações de entrada e saída (é um processo IO-bound), algo que acaba por prejudicar o seu desempenho.

   As chamadas ao sistema que podem ser consideradas como a causa principal desta situação são as de leitura do conteúdo do ficheiro e as de escrita do conteúdo lido no pseudo-terminal, pois são invocadas repetidamente. As chamadas de abertura e fecho de canais de entrada/saída para o ficheiro pretendido também consomem algum tempo de execução no CPU mas não tanto como as anteriormente referidas, pois a sua invocação é menos frequente.

   A chamada de leitura do ficheiro é invocada a cada 1024 *bytes* do conteúdo do ficheiro e a chamada de escrita é invocada a cada *byte* lido pela anterior. Isto implica que por cada 1024 *bytes* do ficheiro e por cada *byte* escrito no pseudo-terminal, o sistema operativo tem que interromper a execução do programa e realizar a operações pretendidas resultantes das chamadas ao sistema.

2. Antes da partilha de informação entre servidor e cliente é necessário criar-se uma *socket* associada a uma determinada porta. Esta deveria ficar dísponivel no final da execução do servidor para que no futuro fosse possível estabelecer-se algum outro tipo de conexão nessa mesma porta por qualquer outro processo. No entanto, tal não se verifica no nosso programa, pois, o servidor desenvolvido não tem nenhum comando ou condição para terminar a sua execução, mantendo-se num ciclo constante. A única forma de terminar o processo é forçar a terminação do mesmo, impedindo que este consiga efectuar algum mecanismo de fecho da *socket* utilizada.

3. O *buffer* circular implementado tem um tamanho máximo pré-definido de números inteiros que consegue conter. Definimos esse valor como 16, i.e. o *buffer* tem capacidade apenas para conter 16 descritores de ficheiros para impressão, sendo que se se tentar adicionar um valor superior, o servidor deixa de interpretar comandos até terminar pelo menos uma impressão e libertar uma posição do *buffer*.

Relatório realizado pelos alunos Francisco Godinho (nº 41611) e Pedro Carolina (nº 41665) do turno prático 5.