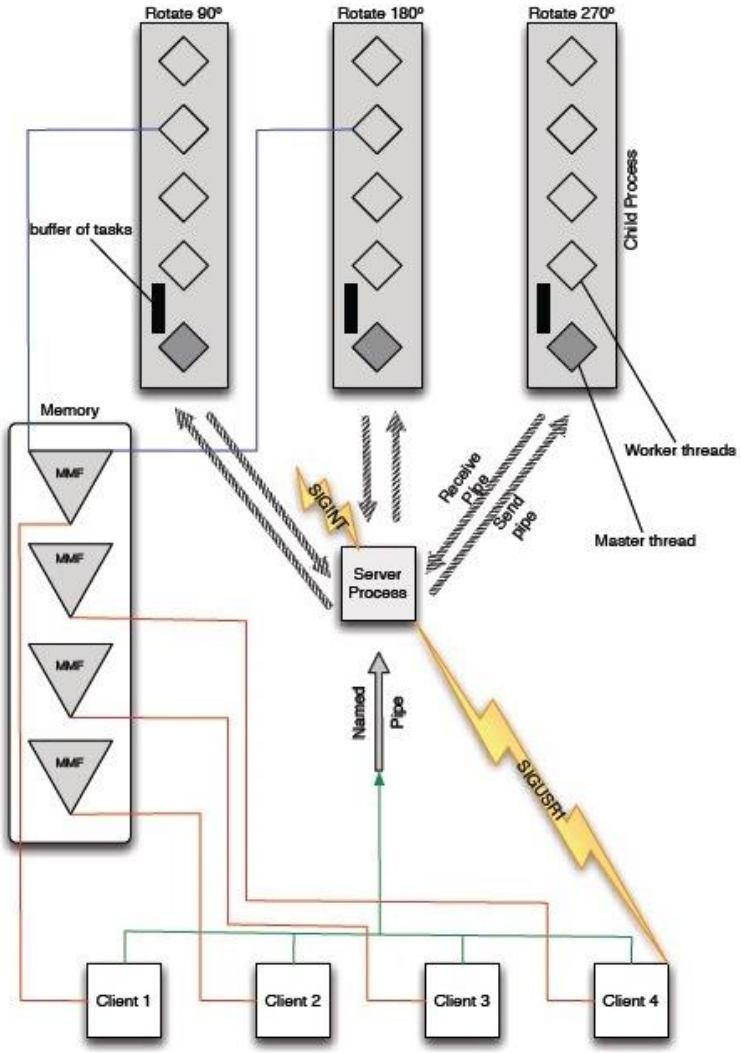
Manual de Programador

Projecto – Rotação de Imagens

Sistemas Operativos



Filipe Mendes

2012139689

António Simões

2012145253

• U C •

FCTUC FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE DE COIMBRA

Manual de Programador

Como podemos ver na esquematização do projecto, podemos dividí-lo em 3 zonas:

➔ Cliente

O cliente é responsável por ler a imagem que é passada como parâmetro na chamada do programa, ou seja, *./client <path imagem> <rotação>*. Este programa é responsável por abrir o *Named Pipe* em modo de escrita, a fim de enviar o seu pedido ao servidor, sendo este pedido uma *struct cliente*.

```
typedef struct{
    char path[1024];
    pid_t pid;
    int rot;
    size_t size;
}cliente;
```

O campo *pid* é obtido pela função *getpid()*, que irá será útil para lhe ser enviado o sinal pelo servidor, a informar acerca do resultado da rotação, e o campo *size* é obtido com recurso à função *get_stat()*, que calcula o tamanho da imagem.

Após a leitura de todos estes dados para a estrutura a enviar e do pedido realizado, o cliente desconecta-se do *Named Pipe*, para ficar disponível para um novo pedido, e espera pelo sinal de sucesso ou insucesso do seu pedido.

É realizado ainda um mapeamento da imagem, com o propósito de ser um *backup*, caso a rotação seja mal realizada e altere a imagem original.

➔ Servidor

O servidor é responsável pela criação de todas as estruturas fundamentais para execução do programa e cria-as *"a priori"*, ou seja, antes de fazer qualquer comunicação ou que receba algum pedido. Estas estruturas referidas tratam-se de:

- um *Named Pipe* para comunicação com todos os clientes que se pretendam ligar e enviar um pedido ao servidor;

- 3 processos-filhos, cada um responsável por um tipo de rotação (90º, 180º ou 270º), ligados ao servidor por 6 *unnamed pipes*, 2 para cada processo, visto tratar-se de um meio de comunicação unidireccional;

- um *select()* que vai ficar a escuta do *Named Pipe* e dos 3 pipes de comunicação processos-filhos-servidor.

Relativamente ao funcionamento, o cliente escreve o seu pedido (uma *struct cliente*) no *Named Pipe*, ao qual o servidor está à escuta. O servidor lê o pedido e direcciona o pedido para o respectivo processo-filho. Assim que o pedido for satisfeito (a imagem rodada) e receber uma mensagem com uma *struct validade*, envia um sinal ao cliente em causa, *SIGUSR1* em caso de sucesso ou *SIGUSR2* em caso de insucesso.

```
typedef struct {
        pid_t pid;
        int rotacaofeita;
}validade;
```

➔ Processos-Filhos

O processo-filho começa o seu funcionamento pela criação de 4 *threads*, passando a funcionar como a *master thread* e as outras 4 novas *threads* como *worker threads*, e também um buffer comum a todas as *threads*. Este buffer é organizado sob o princípio do algoritmo de consumidor-produtor e é do tipo cliente, ou seja, vai ser responsável por armazenar os pedidos por realizar, vindos do servidor. Assim que algo for escrito no buffer, será desbloqueada uma das worker threads, que terá o papel de fazer a rotação da imagem pretendida. Caso a rotação seja concluída com sucesso, será criada uma *struct validade*, que terá, no campo *pid*, o identificador do cliente, e, no campo *rotacaofeita*, o inteiro 0. Caso contrário, inteiro 1. Por fim, envia esta *struct* novamente para o servidor.

Toda a sincronização é realizada com o auxílio de um mutex, responsável por bloquear e desbloquear as *worker threads*, e dois semáforos, empty e full, inicializados a MAXFILA e a 0, respectivamente. Estes últimos são responsáveis pela gestão da fila de espera. Utilizamos ainda dois inteiros, read_pos e write_pos que funcionam como uma espécie de ponteiros para a próxima posição a ler e a escrever, respectivamente, do buffer.

➔ Clean Shutdown

Por fim, o nosso *clean shutdown* é feito com recurso à função *sigint()*, que fecha todos os pipes existentes, mata todos os processos-filhos (armazenados num array de pid_t) e destrói o mutex criado para a sincronização das nossas threads.