Universidade do Minho

Escola de Engenharia

Mestrado Integrado em Engenharia Telecomunicações e Informática

Serviço de Agendamento de Tarefas

David Alves, A79625

Gilberto Morim, A65214

Dezembro de 2017

Índice

1.Introdução

Para este projeto de Sistemas Operativos foi-nos pedido a elaboração de um programa de agendamento de tarefas no qual existe uma interação entre um cliente e um servidor. Para a concretização do mesmo foi utilizada linguagem C, recorrendo assim aos conteúdos aprendidos ao longo das aulas teóricas e práticas.

2.Desenvolvimento

2.1 Contextualização

Este projeto tem como objetivo o agendamento de tarefas armazenando o seu resultado num servidor, além disso tem os seguintes objetivos:

- Agendar a execução de uma tarefa para uma data e hora específica (agendar AAAAMMDD hhmm tarefa:);
- Listar os agendamentos previamente realizados, consoante o estado especificado
- Consultar informação detalhada de relativa a uma tarefa previamente agendada;
- Cancelar a execução da tarefa especificada

2.2. Estrutura do Programa

- **Cliente**
  O programa cliente é o programa que transmite ao servidor os pedidos que o cliente efetua. A comunicação entre cliente e servidor é efetuada utilizando dois *pipes* com nome: um para onde o cliente escreve e o servidor lê (por outras palavras, o servidor verifica os pedidos do cliente), e outro para o qual o servidor escreve e o cliente lê. O programa do cliente funciona com dois processos: um que lê do *standard input* os pedidos do cliente e os escreve para o *pipe*; e outro que lê do *pipe* a resposta do servidor e a apresenta no *standard output*.

- **Servidor**
  O servidor é responsável por processar os vários pedidos do cliente. Conforme o pedido, o servidor executa um determinado conjunto de instruções. É formado por dois processos base: um que apenas cria o *pipe* com nome de leitura, e que cria um segundo processo (este sim ativamente lê e processa pedidos do cliente); o primeiro processo depois espera pela morte do segundo processo, para posteriormente remover os ficheiros dos *pipes* de comunicação.
  Este segundo processo armadilha o sinal SIGCHLD, para poder fazer wait() aos filhos que terminam (sem precisar de esperar ativamente pela morte do filho e potencialmente ignorando pedidos de clientes). Assim evitamos a existência de processos zombies.
  Ao receber um pedido de agendamento de tarefa, o servidor guarda em memória, numa *struct*, os dados referentes ao pedido: tarefa pedida, tempo, estado, ID, e processo onde foi executado. É efetuado um *fork()*, para criar um novo processo (ou conjunto de processos) onde serão executados os programas pedidos pelo cliente. Este processo, antes de executar, armadilha o sinal SIGALRM, e agenda um sinal SIGALRM correspondente ao tempo pedido pelo cliente. O processo, após isso, fica adormecido. Após este intervalo de tempo decorrer, a tarefa é executada. Caso a tarefa seja composta por vários

programas, são efetuados vários *fork()* correspondentes ao número de programas a executar. São abertos também dois pipes anónimos (um de "escrita para o processo seguinte", outro de "leitura do processo anterior"). Cada processo tem então o seu *standard input* e *output* redirecionados via *dup2* para o respetivo descritor do *pipe* anónimo (ou para o *pipe* com nome do *output*, caso seja o último processo da cadeia, para enviar o resultado ao cliente).

Quando recebe um pedido de "listar", o servidor apenas percorre a estrutura de dados que contém as informações de todas as tarefas, e lista todas se o cliente assim pediu, ou então as tarefas com um determinado estado (*"scheduled", "cancelled", "terminated"…*).

Da mesma forma, para um pedido de "consultar", a estrutura de dados é percorrida para procurar a tarefa com ID requisitada pelo cliente.

Para cancelar, após encontrar a tarefa com ID indicado pelo cliente, é enviado um SIGTERM para o processo responsável por essa tarefa. Este SIGTERM causa o *_exit()* do processo adormecido, mas faz com que ele termine com um status diferente, que permite alterar o estado do processo para "*cancelled*".

2.3. Limitações e falhas na implementação

O programa implementado, apesar de tudo, apresenta certas limitações, que não deveriam existir de modo a garantir a eficiência e bom funcionamento do programa:

- **Guardar em memória os dados das tarefas todas:** As únicas tarefas que deveriam ser guardadas em memória são as tarefas agendadas e em execução. As tarefas terminadas e canceladas deveriam ter os dados e output enviados para respetivos ficheiros em disco, cujo conteúdo seria lido e transmitido ao cliente caso ele solicitasse. Como a memória do disco é bem maior que a memória RAM, teríamos mais eficiência neste aspeto.
- **Bug na introdução dos dados do cliente:** Caso um cliente requisite, por exemplo, "ps -a | wc" o servidor não reconhece o pedido. Apenas reconhece na forma "ps -a | wc |". Isto ocorre devido à função de *parsing* apenas acrescentar o *null terminator* à lista de argumentos para o *execvp* quando encontra um caracter "|". Este foi um bug que não conseguimos solucionar até à data de entrega.
- **Servidor apenas encadeia dois processos:** Devido à falta de completo domínio desta matéria, não conseguimos criar efetivamente um ciclo que criasse vários processos, e redirecionasse adequadamente os *outputs* e *inputs;* ficámos apenas pelo encadeamento de dois processos.
- **Data no comando que o cliente envia diferente do enunciado:** O cliente não indica a data em específico da execução da tarefa. Indica sim, "daqui a quantos segundos a tarefa deverá executar" - na forma "agendar 10 ps -a | wc |", em que deve executar daqui a 10 segundos. Isto ocorre devido à nossa dificuldade em fazer *parsing* das *strings* no formato de data.
- **Tarefa nem sempre fica com estado "*cancelled*":** Devido a conflito entre os sinais enviados no término de um processo, por vezes uma tarefa cancelada não atualiza corretamente o estado.
- **Falta de gestão de concorrência entre clientes:** Como apenas existem dois *pipes* com nome de comunicação, não existe controlo de acessos de cada cliente. Para resolver este problema, a solução passaria por cada cliente criar o seu próprio *pipe* de leitura, e o servidor guardar a informação de qual descritor corresponde a qual cliente. Assim enviaria os pedidos de cada cliente para o descritor associado.

3.Conclusões

Com este projeto conseguimos aplicar os conteúdos lecionados na UC de Sistemas Operativos a uma simulação de uma interação de um cliente e servidor no mundo real. Conseguimos observar os parâmetros a considerar e preocupações a ter quando estamos perante o desenvolvimento de um sistema de serviço a vários clientes.

Também conseguimos ganhar a noção de quais conteúdos ficaram menos bem consolidados das aulas (nomeadamente execução encadeada de processos), bem como certas bases de programação em C (nomeadamente trabalho com *strings*).

No entanto, percebemos em que situações necessitamos de vários processos para resolver um pedido, e como o sistema operativo age para responder a esse pedido, de como se processa leitura e escrita para ficheiros, e da importância de processos comunicarem entre eles através de *pipes* e sinais.