Controlo e Monitorização de Processos e Comunicação

Sistemas Operativos

1 Introdução

O projeto a ser desenvolvido visa o despacho e monitorização de processos, nomeadamente é preciso garantir os seguintes pontos. Este sistema possui duas grandes componentes, o cliente e servidor. O cliente é quem emite *jobs* que representam comandos normais, que podem vir a usufruir da capacidade de *piping*, por meio de *pipes* anonimos. O servidor é quem processa e dá resposta a estes *jobs*, armazenando registos adequados para este efeito. É necessário focar em cada um destes sistemas individualmente.

2 Cliente

Por simplicidade, assume-se que o cliente envia mensagem para uma caixa negra, que é o servidor, mas na qual apenas tocaremos de seguida. Por essa razão, assumimos que o cliente pode enviar mensagens, por meio a definir, e, eventualmente, receber respostas de volta.

Na realidade, pasta soluccionar o problema para a linha de comando, com isso encontrar um metódo para a resolução via *shell* torna-se imedita. O cliente tem pouco, ou até nada que se lhe diga, por isso só é de interesse analisar o tipo de *inputs* que este recebe.

Para a execução do enunciado, é necessário que o cliente consiga interagir das seguintes formas com o servidor.

1. **Tarefa 1** Definir o tempo máximo de inactividade de comunicação num pipe anónimo.
2. **Tarefa 2** Definir o tempo máximo de execução de um tarefa.
3. **Tarefa 3** Executar uma tarefa.
4. **Tarefa 4** Listar tarefas em execução.
5. **Tarefa 5** Listar registo histórico de tarefas terminadas.

6. **Tarefa 6** Apresentar ajuda à utilização. Esta opção é dependente da linha de comandos e *shell*.
7. **Tarefa 7** Consultar *standard output* produzido por uma tarefa já executada.

3 Servidor

Corresponde à peça fundamental do sistema, unidade centralizada de processamento. É aqui onde são efetuados todos os necessários processamentos, respectivo armazenamento de logs, *et cetera*. Desta forma, o servidor deve ser capaz de encapsular os seguintes mecanismos:

1. **Tarefa 8** Existir múltiplos *jobs* em execução simultaneamente.
2. **Tarefa 9** Armazenar ficheiro de *log* dos comandos executados.
3. **Tarefa 10** Armazenar ficheiro de *log* do *standard output*. Com o auxilio de um ficheiro de *offsets*, indicando, para cada *id* de pedido, o respectivo offset onde se encontra no ficheiro de *logs*, idealmente será também necessário indicar o tamanho em bytes desse resultado.

4 Resolução teórica de cada uma das tarefas.

4.1 Tarefa 1

Definir o tempo máximo de inatividade de cada um dos pipes é imediato aquando do despacho dos pipes. Quando o pipe é encadeado, basta definirmos um SIGALARM, que apos o TIME_TO_LIVE_PIPE do pipe emite um alarme para o processo indicado, executando um SIGKILL sobre este e indicar dentro do sistema que este processo foi morto devido a inatividade do pipe.

4.2 Tarefa 2

Definir o tempo máximo de execução de uma tarefa é exatamente igual, basta, antes do processamento, fixarmos um alarme, no processo responsável por processar o comando, que apos TIME_TO_LIVE_JOB emita um SIGALARM que deverá provocar um SIGKILL no processo atrasado.

Isto poderá ser posteriormente testado usando *sleeps*.

4.3 Tarefa 3

Executar uma tarefa é trivial, o código *source* já foi feito em guiões passados. Já possuo um mecanismos capaz de lidar com um número arbitrário de encadeamento de pipes. Por isso isto não será de todo um problema.

4.4 Tarefa 4

Para listar tarefas em execução, assume-se que o processo tem acesso a um lista de todos os processos que estão em ativos. Porém, dá jeito que esta "lista" seja uma estrutura eficiente para a questão, isto porque, eventualmente, vamos precisar de retirar estes valor assim que o comando seja terminado, e a tarefa deixe de estar em execução.

O que acontece se considerarmos uma array? a inserção é em tempo constante, basta fazer append na cauda, porém a sua remoção seria muito dispendiosa, em tempo linear. O que é pessimo.

Uma coisa que podemos utilizar é um mapeamento direto entre o numero máximo do PID, e, desta forma, associar a cada posição de um array, um mapeamento direto de valores segundo o seu PID. O número máximo do PID é dado pelo comando `cat /proc/sys/kernel/pid_max`. Porém, este processo pode atingir valores extremamente altos, numa arquitetura 32 bit corresponde a 32768 e numa arquitetura de 64 bit corresponde a 4194304. Apesar de parecem intimidadores, é preciso considerar, a todo o momento, que estamos a falar de um servidor, que teoricamente tem uma grande capacidade de processamento e armazenamento. No pior dos casos, o PID máximo corresponde a 4194304, logo vamos precisar de considerar esses tantos inteiros, cada um desses inteiros ocupa 4 bytes, então, no total, só com a tabela, iremos ter uma ocupação de 4194304 * 4 bytes = 16 Megabytes, o que pode parecer muito, porém, considerando o panorama global, é completamente insignificante.

Porém, isto induz num outro problema, de que forma vamos verificar quais dos processos é que estão ativos? Evidentemente, percorrer um array com 4194304 entrada é ridiculo, pois introduziria uma complexidade muito maior. Para resolver este problema podemos utilizar uma lista ligada, sendo que cada posição da tabela de hash vai ter acesso direto ao nó da lista ligada, por meio de endereços partilhados, permitindo assim uma inserção e remoção em tempo constante, apenas ao custo de um pouco de memória. Assim, sempre que pretendemos saber quais os processos em execução, apenas temos de percorrer a lista ligada em questão.

4.5 Tarefa 5

De forma a listar o registo histórico de tarefas terminadas, primeiro temos de ter em conta que podem existir, potencialmente, milhões de milhares de tarefas já terminadas, um número que podemos majorar como sendo demasiado grande para arrecadar em memória. Então, a única opção viavel é possuir um ficheiro unicamente para o efeito de especificar o historico de tarefas terminadas. Sempre que um processo termina, simplesmente damos append no ficheiro indicado para o efeito.

4.6 Tarefa 6

Apresentar ajuda à utilização é trivial, sendo, porém, necessário distinguir entre os dois metódos principais, *shell* e argumentos.

4.7 Tarefa 7

O standard output de cada tarefa tem de ser armazenado num ficheiro, por potencialmente, ser demasiado grande. Porém, isto induz em problemas graves, o que acontece se tivermos mais de 1 pessoas como cliente? temos de arranjar um mecanismo de controlo de acesso ao ficheiro. Porém, acho que isto não é âmbito da cadeira, tenho de perguntar ao prof. Para já assume-se que só existe um cliente, existindo só um cliente, então cada vez que escrevemos o output no ficheiro indicado para esse efeito, basta que, simultaneamente escrevamos, num ficheiro de indices, o offset dessa informação, bem como o tamnho em bytes que ocupa em memória, para permitir um rápido acesso, por meio de lseek. Fazer lseek será sempre melhor, no pior dos casos, do que reabrir o ficheiro, ir ao offset, e voltar à posição final.

A formatação do ficheiro será a seguinte:

```
[ID1] [OFFSET] [TAMANHO BYTES]
[ID2] [OFFSET] [TAMANHO BYTES]
...
```

4.8 Tarefa 8

Para permitir muitos jobs simultaneamente, basta que, a cada novo *job* seja criada uma threada para esse efeito, devidamente anexada na tabela para esse efeito.

4.9 Tarefa 9

Isto é identico ao ponto da tarefa 5.

4.10 Tarefa 10

Isto é identico ao ponto da tarefa 7.