

Argus - Controlo, Monitorização e Comunicação de Processos

Grupo 112

José Reis^[A87980], José Rodrigues^[A87961] e Rui Reis^[A84930]

Universidade do Minho, Departamento de Informática, 4710-057 Braga, Portugal

e-mail: {a87980,a87961,a84930}@alunos.uminho.pt

Resumo Desenvolvimento de um sistema com capacidades de controlo, monitorização e comunicação de processos, construído sobre uma arquitetura cliente - servidor.

1 Introdução & Contextualização

Num sistema ideal, devemos ser capazes de processar vários pedidos em simultâneo. Porém, esse tipo de mecanismos implicam uma maior atenção ao detalhe com que cada operação é efectuada. A paralelização obriga um cuidado com a gestão dos diversos recursos em processamento.

Com esse intuito, e no âmbito da unidade curricular de Sistemas Operativos, foi desenvolvido o sistema *Argus*, um sistema capaz de eficiente processamento de distintas tarefas, com uma atenção especial pelo controlo, monitorização e comunicação de processos.

2 Arquitetura da Solução

O sistema requisitado, apesar de não aparentar, constitui uma peça de software com mecanismos específicos e complexos, que devem ser bem ponderados, em sincronia, de forma a garantir um produto final viável.

2.1 Cliente e Servidor

Na sua componente mais atómica, este sistema pode ser representado por um cliente e servidor. O cliente, que tanto pode ser na forma de *bash* ou com a passagem de argumentos, comunica ao servidor mensagens na forma de pedidos, **Requests**. O servidor, por sua vez, capta estes pedidos e destaca internamente as entidades responsáveis pelo processamento de cada tipo de pedido, individualmente.

De forma simétrica, o servidor responde ao cliente com respostas, *Response*, encapsulando a resposta necessária. Tendo sempre em atenção que o servidor é completamente independente do contexto do cliente. Podemos então resumir este fluxo na seguinte imagem, na qual verificamos que tanto o cliente, como o servidor, tiram partido de pipes com nome, de forma a comunicar mensagens entre si. Resultando em:

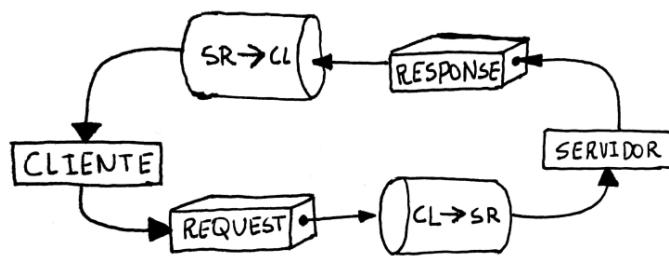


Figura 1. Comunicação entre Cliente e Servidor.

Assume-se que o cliente está a utilizar a linguagem portuguesa. Porém, se o cliente pretender interagir com o servidor utilizando a linguagem alemã, iremos obter algo do género:

```

→ src git:(master) ✘ ./argus language DE
argus$ output 2
Aufgabe #2 existiert nicht
argus$ █

```

Figura 2. Resposta independente do contexto.

No qual podemos verificar, que a resposta do servidor é completamente independente do contexto. Neste caso o servidor comunica ao cliente que não existe nenhuma tarefa armazenada com o *id* 2, sem qualquer tipo de noção de linguagem.

2.2 Monitorização de Processos

De forma a manter um constante controlo de quais processos estão em funcionamento, é armazenada uma tabela de *hash* no qual a todos os pedidos em abertos é associada informação relativa ao estado atual da

tarefa, bem como o identificador de processo, *pid*, do processo principal responsável pelo processamento desse comando.

Deste modo, garantimos um total controlo sobre todos os elementos a serem processados. Assim, caso o objetivo seja, por exemplo, assassinar uma tarefa específica, podemos simplesmente aceder à respectiva posição da tabela e exigir a terminação do grupo do processo principal deste, anexando o tipo de terminação induzida.

2.3 Controlo de Processos

No intuito de controlo de processos, este sistema encapsula a capacidade de limitar o tempo de vida de um determinado comando, bem como, limitar o tempo de inactividade dos pipes anónimos utilizados ao longo do comando indicado, desta forma atingimos o controlo processual desejado.

Programas cujo funcionamento demore mais que `EXEC_TIMEOUT` são automaticamente terminados, recorrendo a sinais. De igual forma, pipes anónimos que fiquem `PIPE_TIMEOUT` segundos inativos são também terminados recorrendo a sinais, ambas as metodologias anexam o tipo de terminação daquele comando.

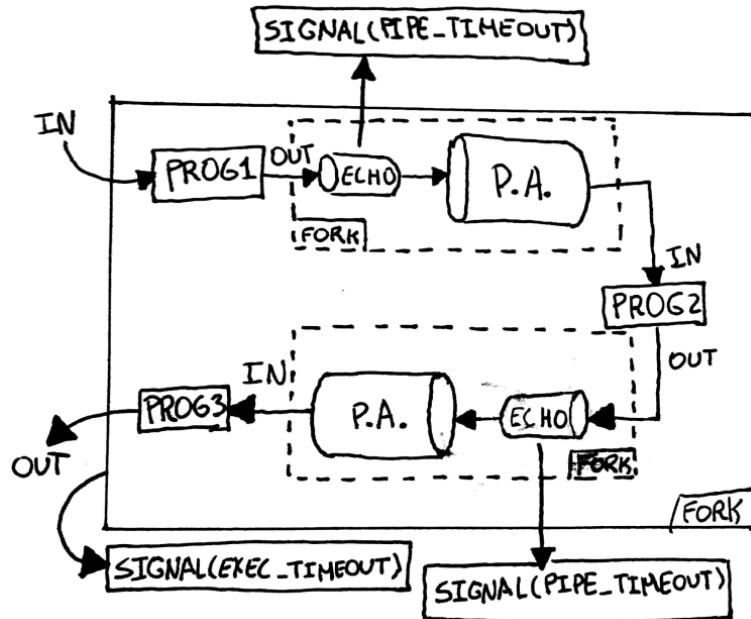


Figura 3. Funcionamento de um comando com 2 *pipes* intermédios.

Acima vemos exemplificado este mecanismo, onde, para cada *pipe* anónimo, existe um outro *pipe* anónimo, na figura denotado por ECHO, que serve de intermédio entre o output do 1º programa, e a ponta de escrita do *pipe* anónimo. Sempre que o *pipe* ECHO capta uma mensagem, emite um `alarm(PIPE_TIMEOUT)`, que tanto cria um alarme, como faz reset de qualquer outro que exista. Assume-se que o `SIGALARM` irá, posteriormente, terminar adequadamente o comando em execução. Após a colocação do alarme, este *pipe* reencaminha a mensagem para o *pipe* P.A.

Numa vista mais abrangente, o processo de onde originam todos os programas e *pipes* anónimos, aquando da sua criação, emite um alarme, da seguinte forma, `alarm(EXEC_TIMEOUT)`, que irá, posteriormente, obrigar a terminação do processamento caso este ultrapasse o tempo máximo de execução.

2.4 Histórico de Comandos

Assim que o programa termina, deverá ser mantido um ficheiro contendo o *id*, código de terminação e comando executado de cada tarefa, garantido assim um *logging* de todo o conteúdo.

Posteriormente, a impressão deste ficheiro, intitulado de `command_log`, poderá ser requisitada pelo comando `historico`, ou com a flag `-h` na versão por argumentos.

2.5 Armazenamento do Output

Como funcionalidade adicional, era desejado que o output fosse armazenado, e, posteriormente, acedido pelo cliente. Desta forma, é necessário que todo o output dos diferentes comandos seja armazenado no ficheiro `log`. Como método auxiliar, deverá ser, também, gerado um ficheiro `log.idx` com o objetivo de permitir a indexação rápida de cada uma das tarefas.

O ficheiro `log.idx`, na nossa implementação, corresponde a um ficheiro, binário, de entradas com tamanho fixo, em que cada entrada corresponde a um tuplo com os seguintes elementos.

```
[ID TAREFA] [OFFSET OUTPUT] [LENGTH OUTPUT]
```

O primeiro argumento é imediato, porém os restantes, em sincronia, permitem aceder diretamente, no ficheiro `log`, à posição onde se encontra o output daquela tarefa, com o argumento `length` conseguimos obter o tamanho do output armazenado.

Porém, é necessário considerar uma questão mais profunda. Sem a existência de *locks*, ou outros mecanismos externos, não temos controlo sobre o acesso ao ficheiro de *logs*. Isto, no pior cenário, pode implicar um entrelaçamento de outputs, tornando, assim, o output de cada tarefa ilegível. O que não é de todo o desejado, por essa razão, há a necessidade de encontrar um mecanismo que permita o acesso atómico ao ficheiros de *logs*.

Derivado desse problema, em vez de cada comando escrever o seu output imediatamente no ficheiro de logs, é utilizado, para cada comando, um buffer intermédio que armazena a totalidade do output daquele comando. De seguida, e de forma individual, estes buffers são transmitidos ao longo de um *bottleneck*, um pipe anónimo cuja ponta de leitura ficará encarregue de ler cada entrada do *bottleneck*. Após a sua leitura, cada entrada é imediatamente escrita no ficheiro desejado. Segundo uma lógica análoga à da figura seguinte.

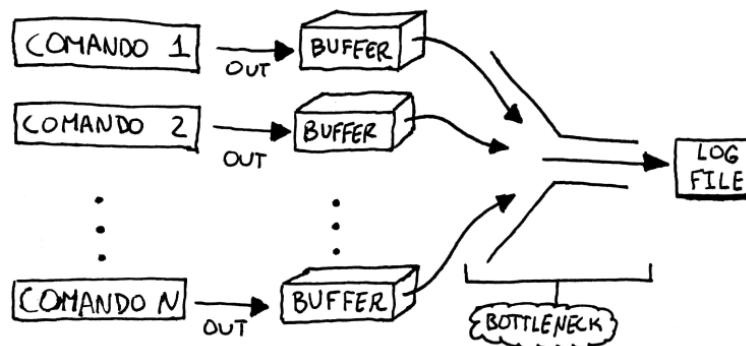


Figura 4. Arquitetura da Metodologia *bottleneck*.

Desta forma, e assumindo que os buffers são transmitidos de uma só vez pelo *bottleneck*, temos a garantia de que cada operação é atómica, não existindo qualquer tipo de entrelaçamento de diferentes outputs.

3 Testes Realizados

Inicialmente, foram propostas as seguintes capacidades para este sistema.

1. Executar uma tarefa.
2. Listar tarefas em execução.

3. Listar histórico de tarefas terminadas.
4. Consultar o *standard output* de uma tarefa.
5. Limitar o tempo de inactividade de um pipe anónimo.
6. Limitar o tempo de vida de um comando.

De seguida, de forma a testar todas estas capacidades, foram desen-
volvidos os seguintes testes. Omitem-se certos requisitos, que do ponto de
vista técnico são apenas extensões triviais de outros requisitos.

3.1 Requisito 1

É possível executar vários pedidos seguidos no sistema, sendo que estes
ficam todos catalogados no servidor para posterior utilização.

```
argus$ executar `ls -l`  
Nova tarefa #0  
argus$ executar `cat /etc/passwd`  
Nova tarefa #1  
argus$ executar `ls -l | head -5 | sort`  
Nova tarefa #2  
argus$ █
```

Figura 5. Cumprimento do 1º requisito.

3.2 Requisito 2

Ademais, é também possível aceder a todas as tarefas em execução, bas-
tando para tal utilizar o seguinte comando:

```
argus$ executar `sleep 5`  
Nova tarefa #6  
argus$ listar  
#6: sleep 5  
argus$ █
```

Figura 6. Cumprimento do 2º requisito.

3.3 Requisito 3

O histórico de tarefas, bem como o seu tipo de terminação, podem ser a
todo o momento acedidas.

```
argus$ historico
#0, concluida: ls -l
#1, concluida: cat /etc/passwd
#2, concluida: ls -l | head -5 | sort
#3, concluida: cat /dev/null
#4, concluida: cat /dev/nul
#5, concluida: cat /etc/passwd
#6, concluida: sleep 5
argus$ █
```

Figura 7. Cumprimento do 3º requisito.

3.4 Requisito 4

O output pode também ser verificado da seguinte forma, garantido o resultado exato àquele produzido pela própria bash.

```
argus$ output 2
prw-r--r-- 1 syrayse syrayse      0 jun 11 14:37 cl2srpipe
-rw-r--r-- 1 syrayse syrayse  45168 jun 11 14:36 Cliente.o
-rw-r--r-- 1 syrayse syrayse    7325 jun 10 12:31 Cliente.c
-rwxr-xr-x 1 syrayse syrayse 104640 jun 11 14:36 argus
total 652
argus$ █
```

Figura 8. Cumprimento do 4º requisito.

3.5 Requisito 5

O mecanismo desejado pode ser obtido inserindo um pipe intermédio que dorme durante um tempo que excede o limite máximo, neste caso 10, tal como se pode verificar:

```
argus$ executar `ls | sleep 7 | sort`
Nova tarefa #0
argus$ executar `ls | sleep 15 | sort`
Nova tarefa #1
argus$ historico
#0, concluida: ls | sleep 7 | sort
#1, max tempo inactividade do pipe: ls | sleep 15 | sort
argus$ █
```

Figura 9. Cumprimento do 5º requisito.

3.6 Requisito 6

Este mecanismo pode ser simplesmente testado colocando em execução um `sleep` que ultrapasse o tempo limite de execução, neste caso 20.

```
argus$ executar `sleep 15`  
Nova tarefa #0  
argus$ executar `sleep 25`  
Nova tarefa #1  
argus$ historico  
#0, concluida: sleep 15  
#1, max tempo execucao: sleep 25  
argus$ █
```

Figura 10. Cumprimento do 6º requisito.

4 Conclusão

Em suma e ponderando todas as distintas componentes envolvidas, achamos que conseguimos atingir um mecanismo com o nível de complexidade desejado. Isto é explicado tendo em conta a capacidade que o sistema tem em lidar com um vasto leque de objetivos, garantindo assim o cumprimento de toda a funcionalidade mínima, bem como funcionalidade adicional requisitada pelos docentes.