Universidade do Minho

Departamento de Informática

Trabalho Prático

Sistemas Operativos
Grupo 92

João Freitas A83782

Mike Pinto A89292

Rafael Gomes A96208

13 de Maio de 2023

Conteúdo

Capítulo 1

Introdução

O presente relatório tem como objetivo apresentar uma explicação e justificação face à solução encontrada e desenvolvida pelo grupo de trabalho, no âmbito da Unidade Curricular de Sistemas Operativos.

Foi então proposto o desenvolvimento de um serviço de monitorização dos programas que são executados numa determinada máquina. Os utilizadores devem conseguir, através de um cliente e um monitor, executar programas obtendo o seu tempo de execução, realizar pedidos e consultas sobre os programas executados e obter ainda estatísticas sobre os mesmos.

Ao longo deste relatório serão analisados os dois programas, *monitor* e *tracer*, nomeadamente, as estruturas de dados utilizadas para armazenamento dos dados, os mecanismos de comunicação para a troca de mensagens entre os programas e a arquitetura de processos implementada.

Capítulo 2

Mecanismos de comunicação

A comunicação entre os dois programas, via *pipes com nome*, requer a troca de vários tipos de dados, de forma a manter as operações de execução e monitorização requeridas. É necessária uma estrutura flexível, capaz de acomodar os diversos tipos de pedidos feitos pelo *tracer* e as diferentes respostas dadas pelo *monitor*. Pretendemos ainda, obter um sistema de comunicação que garanta uma troca de mensagens entre ambos os lados clara, estruturada e de fácil interpretação.

Tendo estes fatores em vista, foi criada a *struct message*, que representa a interface de comunicação entre ambos os programas. Esta estrutura tem a flexibilidade de representar vários tipos de mensagens tendo sempre um tamanho fixo independentemente do tipo, garantindo assim, que qualquer que seja o conteúdo, este chegue ao destinatário sem ocorrer falhas ou faltas de informação, na medida em que a leitura e a escrita tem um tamanho fixado.

A estrutura está desenhada para poder incluir vários tipo de mensagens mantendo um tamanho fixo, isto é conseguido com:

- uma variável do tipo *int* que guarda o número inteiro correspondente ao tipo de mensagem.
- uma variável do tipo *union*, que guarda todas as estruturas correspondentes às diferentes mensagens. Este *data type* permite que a estrutura da mensagem possa tomar as várias formas necessárias, tendo em conta o contexto da tarefa a ser executada, mantendo a característica mais importante, o tamanho fixo, pois independentemente do tipo de informação, o tamanho de memória alocado é sempre o mesmo (o *sizeof(int)* mais o tamanho da maior estrutura dentro do *union*).

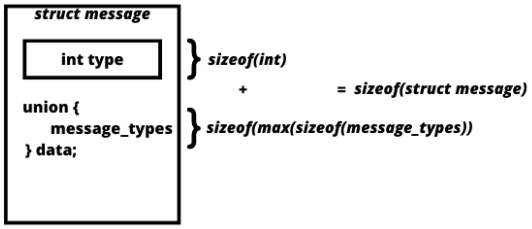

Figura 2.1: *struct message*

Tendo em conta os pedidos que são executados, foram criadas as seguintes estruturas de mensagens:

1. Tracer → Monitor (Starting Execute Single)
2. Tracer → Monitor (Finish Execute Single)
3. Tracer → Monitor (Starting Execute Pipeline)
4. Tracer → Monitor (Finish Execute Pipeline)
5. Tracer → Monitor (status request)
6. Monitor → Tracer (Status response single)
7. Monitor → Tracer (Status response pipeline)
8. Tracer → Monitor (stats time)
9. Tracer → Monitor (stats command)
10. Tracer → Monitor (stats uniq)
11. Tracer → Monitor (status-all request)
12. /13/14/15 - Tracer → Monitor (status-all response)

Capítulo 3

Arquitetura de Processos

A arquitetura de processos constitui um peça fundamental de qualquer sistema. Um sistema bem arquitetado, que tem em conta a gestão do fluxo de processos e canais de comunicação, permite garantir a performance, escalabilidade e confiabilidade do programa.

Neste capítulo iremos apresentar uma explicação relativa a toda a arquitetura do sistema (figura 5.1 dos Anexos), mais concretamente:a arquitetura principal utilizada para a comunicação *tracer→monitor*; a arquitetura do *tracer* no que concerne ao envio de mensagens, execução de comandos e receção de respostas; e a arquitetura do *monitor* referente ao processamento das mensagens, ao envio de respostas e à escrita em ficheiro.

3.1 *Main Channel*

O canal principal de comunicação (*main channel*), é criado pelo *monitor*, no início da execução do programa, e este fica depois a aguardar mensagens na extremidade de leitura do *pipe*. Os clientes (*tracer*) podem enviar todos os pedidos que são suportados, desde a execução de programas às interrogações ao servidor acerca do seu estado.

Dependendo do tipo de pedido efetuado, é criada uma mensagem com o tipo e informação correspondente. Posteriormente é enviada, através da extremidade de escrita do *pipe*, a mensagem para o *monitor* que está à espera de a receber, para que seguidamente possa fazer o processamento apropriado da mensagem.

Em qualquer tarefa ou interrogação ao servidor, é esperada uma resposta por parte do mesmo ao cliente. Esta parte será discutida nas próximas duas secções, onde iremos explicar como é que o cliente espera pelo *output* e como é que o servidor envia.

Por fim, foi acrescentado um comando para que o administrador do sistema consiga terminar normalmente o *monitor*, fechando o canal de comunicação e "apanhar" "processos-zombie" que existam.

3.2 *Tracer*

O programa *tracer* tem 3 tipos de mensagem a ser enviada ao utilizador: pedidos de execução, pedidos de *status* e pedidos de *stats*. Seja qual for o pedido, tal como foi salientado anteriormente, é esperada uma resposta por parte do servidor.

Para que essa resposta seja entregue ao cliente, não pode ser usado o canal principal onde se enviam os pedidos, é necessário criar um canal de comunicação independente e único por cliente, para que o servidor possa responder ao cliente correto com o *output* esperado.

Dito isto, é criado pelo *tracer* um *named pipe*, com o nome do pid do processo para que seja um canal único por cliente. O caminho para onde o servidor irá ter de escrever a resposta é enviado juntamente com a informação do pedido, logo após a criação do *pipe* (para evitar erros com a escrita da resposta). Após o envio da mensagem, o *tracer*, em qualquer uma das execuções irá ficar à espera do *output* na extremidade de leitura do *pipe* criado por si e de seguida apresenta a informação ao cliente no STDOUT.

3.2.1 *Execute*

No pedido de *execute*, o *tracer* é responsável por fazer a execução de um ou mais programas do utilizador e comunicar ao servidor o estado da execução. É o único comando que implica mais do que uma comunicação com o servidor durante a sua execução e também o único que requer algum tipo de processamento por parte do *tracer*.

Existem duas execuções disponíveis, a execução de um programa e a execução de uma *pipeline* de comandos.

Single execution "-u"

Na execução de um comando, o *tracer* recolhe a informação do pedido, nomeadamente o pid, o comando a executar e o tempo de início de execução, e cria uma mensagem com estes dados que sinaliza o servidor sobre o início da execução do programa.

Após ter sido enviada a mensagem ao *monitor*, procede a execução do comando (Figura 5.2 dos Anexos). Para tal, é lançado um filho que executa o comando, com recurso à primitiva *execvp()* e apresenta o *output* do programa ao cliente, sendo que o pai apenas espera pela terminação da execução do filho.

Depois da execução do comando, o programa cria uma nova mensagem que informa ao servidor sobre o fim de execução do comando. Nesta mensagem está o tempo final de execução registado e o caminho por onde o cliente vai esperar pelo tempo total de execução do programa. Aquando a receção da resposta, o programa apresenta a informação no STDOUT ao cliente, fecha os descritores de ficheiros necessários, fecha o *pipe* criado para receber a resposta e termina a execução do programa.

Pipeline execution "-u"

Na execução de uma *pipeline* de comandos (Figura 5.3 dos Anexos), é recolhida a informação de cada comando a ser executado e criada uma mensagem de início de execução da *pipeline* que é enviada ao servidor. Tal como na execução de um comando singular, após o envio da mensagem é iniciada a execução encadeada de comandos.

Neste caso em particular, para o programa conseguir realizar esta tarefa, tem de recorrer aos *pipes* anónimos. Esta mecanismo permite a troca de informação entre processos dentro do mesmo programa.

Após ter sido enviada a primeira mensagem ao *monitor*, é lançado um filho, responsável por executar e apresentar o *output* ao cliente. Este filho, vai criar *(n-1) pipes* anónimos e lançar *n* filhos (sendo n o número de comandos) onde cada um deles vai executar um comando. Dependendo do número do filho, diferentes tarefas são necessárias para executar o comando e encadear a informação corretamente. Existem 3 casos possíveis de execução:

- Primeiro comando (Filho i=0)
  - Redireciona escrita para *pipe* 0
  - Fecha descritores do *pipe* 0
  - Fecha descritores dos restantes *pipes*
  - Executa comando
- Comandos intermédios (Filho i=1 .. Filho (i=n - 2))
  - Redireciona leitura para *pipe* (i-1)
  - Redireciona escrita para *pipe* i
  - Fecha descritores do *pipe* (i-1) e do *pipe* (i)
  - Fecha descritores dos restantes *pipes*
  - Executa comando
- Último comando (Filho i=n-1)
  - Redireciona leitura para *pipe* (i-1)
  - Fecha descritores do *pipe* (i-1)
  - Fecha descritores dos restantes *pipes*
  - Executa comando

Enquanto este trabalho é feito pelos processos filho lançados, o pai fecha os descritores de todos os *pipes* de forma a assegurar que a pipeline não fica bloqueada e depois espera pela terminação dos n filhos lançados.

A restante execução do programa ocorre nos mesmos moldes que a execução de um comando.

3.2.2 *status*

No pedido de *status*, o *tracer* questiona o servidor pelo estado atual dos programas que estão a ser execução no momento. Para realizar esta tarefa, é criada uma mensagem inicial (especifica para este comando), com o pid do processo, o caminho pelo qual o cliente espera a resposta e um *clock* que representa o instante atual para obter o tempo dos programas em execução. Esta mensagem é enviada ao servidor e tal como descrito anteriormente, o *tracer* cria o *pipe* de resposta e fica à espera da mesma na extremidade de leitura. Aquando a chegada da resposta, é impressa no STDOUT para o cliente e realizam-se os passos de normal terminação do programa.

3.2.3 *stats*

Os pedidos *stats* são interrogações do cliente sobre os programas já terminados. O processamento destas interrogações é realizado do *monitor* sendo este detalhado mais à frente. No lado do *tracer*, o *modus operandis* é o mesmo que o comando *status*, é criada uma mensagem específica a cada tipo de pedido para interrogar o servidor e depois é esperada a resposta num *pipe* criado pelo próprio *tracer*, antes do envio da mensagem inicial, e finalmente apresentada a resposta ao cliente.

3.2.4 *status-all*

Este comando foi acrescentado pelo grupo de trabalho de forma a ser possível consultar, nos mesmos moldes do comando *status*, os processos a serem executados no momento e todos os processos já executados anteriormente. O modo de funcionamento deste comando é o mesmo que o *status*, é enviada uma mensagem de interrogação e depois é esperada a resposta no *pipe* criado para o propósito.

3.3 *Monitor*

O programa *monitor*, tem como tarefa principal a monitorização de processos através de mensagens que são lidas e processadas pelo mesmo. É também responsável por gerar e enviar as respostas às interrogações feitas pelos clientes.

Ao inicializar o programa, é passado como argumento o caminho para a pasta onde são persistidas em ficheiro, as informações relativas a cada pedido processado pelo servidor. O canal de comunicação principal é criado pelo monitor antes de este ficar à espera para ler as mensagens na extremidade de leitura do *pipe*. Consoante o tipo de pedido, é feito o processamento adequado para responder ao mesmo.

3.3.1 Estrutura de armazenamento de dados

Quanto ao armazenamento de informação dos pedidos, foi criada a *struct task* que representa um programa executado por um cliente. Tal como na estrutura usada na mensagem, esta também é constituída por uma variável que indica o tipo de tarefa, execução de um comando ou de uma pipeline, e por uma variável do tipo *union* que contem as estruturas respetivas a cada tipo de tarefa. A estrutura é representada na figura 5.4 dos Anexos.

De forma a guardar os pedidos a serem executados e os pedidos terminados, existem 2 variáveis globais que são *arrays* do tipo Task que vão sendo atualizados pelo *monitor*.

3.3.2 *Execute*

No pedido de execute, o *monitor* tem duas ações individuais a realizar: adicionar um novo pedido de execução aos pedidos e finalizar o pedido enviando a resposta ao cliente.

1. Adicionar novo pedido - O *monitor* cria com os dados da mensagem uma nova Task e adiciona diretamente o pedido ao *array* de pedidos a serem executados informando que este foi adicionado com sucesso, no STDOUT do *monitor*. Depois retorna à leitura do *pipe* e fica à espera da próxima mensagem de um qualquer cliente.
2. Finalizar pedido - O *monitor* procura o pedido nos pedidos a serem executados através do pid enviado na mensagem pelo cliente. Depois coloca o pedido no *array* dos pedidos finalizados, retirando do outro. Para não causar demoras desnecessárias no *monitor*, é lançado um filho que vai enviar a resposta ao cliente e depois persistir a informação em ficheiro.

Para enviar a resposta ao cliente, independentemente do tipo de tarefa, é criado o filho, que vai abrir a extremidade de escrita do *pipe*, utilizando o caminho que é enviado na mensagem de finalização de execução. Assim, o filho vai aceder à estrutura do pedido adequado, e enviar pelo *pipe* o tempo total de execução da tarefa, fechando o descritor aberto para a escrita no *pipe*.

Quanto a persistir a informação em ficheiro, o mesmo filho lançado anteriormente, antes de terminar a sua execução, vai guardar um ficheiro com o nome correspondente ao pid do pedido e com a informação respetiva, na pasta passada como argumento ao inicializar o *monitor*. É então aberto o descritor para escrita com as *flags* O_WRONLY | O_CREAT | O_TRUNC e é guardado o pedido. Antes de terminar, é fechado o descritor aberto.

3.3.3 *status*

No pedido de *status* (Figura 5.5 dos Anexos), o *monitor* recebe a mensagem, e lança um filho que vai proceder à resposta a esta interrogação. Assim, a primeira coisa que é feita, é a abertura da escrita para o *pipe* onde o cliente espera a resposta. O descritor é aberto para o caminho enviado na mensagem do pedido. Depois, é percorrido o *array* de pedidos em execução na totalidade e para cada um deles, é criada uma mensagem de resposta para o efeito com a informação sobre a tarefa a ser executada. Após terminar a escrita, fecha o *pipe* e termina a sua execução, informando no STDOUT que foi enviada a resposta com sucesso.

3.3.4 *stats*

Para a execução destas interrogações, um dos aspetos que foi tido em consideração foi a otimização dos tempos de pesquisa quando múltiplos PIDs são passados e é necessário realizar várias operações, pode ser feita de forma independente e distribuída por vários filhos. Também, para qualquer um dos comandos, do lado do *monitor*, é sempre lançado um filho para atender o pedido de forma a que o processamento concorrente de pedidos possa existir, evitando que clientes a realizar pedidos que obriguem a um maior tempo de processamento possam bloquear a interação de outros clientes com o servidor.

A parte de abertura do descritor para escrita de resposta não será mencionada por funcionar da mesma forma que o comando *status*.

stats-time

Para o comando *stats-time* (Figura 5.6 dos Anexos), o objetivo é obter o tempo total de execução de um certo número de processos, passados como argumento do programa. De forma a otimizar o tempo necessário à execução desta *query*, é criado um *pipe* anónimo e lançados tantos filhos quanto o número de processos passados. Cada um destes vai procurar o pedido através do pid no *array* de pedidos finalizados e escrever para o *pipe* criado pelo pai, o tempo total de execução da tarefa. O filho termina a execução com sucesso se encontrar o filho, caso contrário, é enviado o código -1 ao pai para evitar uma leitura desnecessária.

O pai vai esperar pela terminação dos filhos, verificar o código de saída, e ler o tempo de execução, adicionando-o a uma variável que guarda o valor da soma de todos os tempos. Após terminada a leitura do *pipe*, é escrita a resposta para o cliente e fechado todos os descritores necessários.

stats-command

Este comando (Figura 5.7 dos Anexos) serve para interrogar ao servidor sobre o número de vezes é que foi executado um certo comando em x processos, sendo esta informação passada como argumento no *tracer*. Da mesma forma que no comando anterior, é também usado um *pipe* anónimo e lançados tantos filhos quanto pedidos a verificar. Em cada um deles é procurado o processo pelo pid e, para o caso do comando individual corresponder ao comando procurado, é enviado para o *pipe* um "1" que representa uma execução do comando; no caso da pipeline de comandos, é verificado para cada comando executado se corresponde ao comando procurado e é enviada a contagem do número de vezes que o comando é executado na pipeline. O pai da mesma forma que o comando anterior, espera pela terminação dos filhos, soma os valores lidos do *pipe* anónimo e envia a resposta para o cliente, fechando os descritores utilizados.

stats-uniq

Exatamente como os comandos do mesmo género apresentados anteriormente, o início e o mecanismo de optimização usado é o mesmo, um *pipe* anónimo os filhos para executar a tarefa mais eficazmente. Por outro lado, o processamento é um pouco diferente, neste caso, queremos obter a lista de programas que foram executados (sem repetições) para um dado número de processos. Para isso, cada filho lançado vai procurar o pedido terminado e enviar para o *pipe* anónimo o nome do comando executado. De forma a esta escrita e leitura ocorrerem sem erros, é utilizado um tamanho fixo que é o tamanho máximo suportada pela estrutura Task para o nome do comando ou pipeline de comandos.

O pai vai ler do *pipe* anónimo cada um dos nomes dos comandos e adicionar a um *array*. Este processo de adicionar, verifica se o programa lido do *pipe* já consta na lista, no caso de não estar, é adicionado no fim do *array*. No final, é enviado, um por um, o nome de cada comando que consta no *array* e o pai recolhe todos os filhos que criou e fecha os descritores necessários.

3.3.5 *status-all*

Para este comando adicional, o comportamento descrito do *status* é muito semelhante ao deste comando. Difere apenas na tarefa a executar, pois ao invés de procurar apenas pelas tarefas que estão a ser executadas, envia também as tarefas já concluídas, num ciclo adicional que apenas muda o *array* a que acede, sendo o restante processamento igual.

Capítulo 4

Conclusão

Após a realização deste trabalho, podemos compreender melhor os conceitos abordados nas aulas práticas desta UC. Trabalhámos com variadas primitivas que nos trouxeram conhecimento mais aprofundado daquilo que são as bases de Sistema Operativos.

Foi também uma oportunidade de utilizar novas estruturas de dados como o tipo *union*, normalmente esquecido entre os *data types* disponibilizados pela linguagem C.

O grupo conclui que o trabalho desenvolvido atingiu as metas estabelecidas pelo corpo docente. Para além disso, as expetativas e objetivos da equipa foram atingidos com sucesso, fruto do contínuo esforço ao longo do semestre.

Finalmente, quanto à aplicação desenvolvida, constitui uma boa base para facilmente escalar e adicionar novas funcionalidades para os clientes. Poderia também ter sido melhorado o comando extra que foi feito para ser mais eficiente, utilizando processos filho para percorrer os dois arrays simultâneo.

Capítulo 5

Anexos

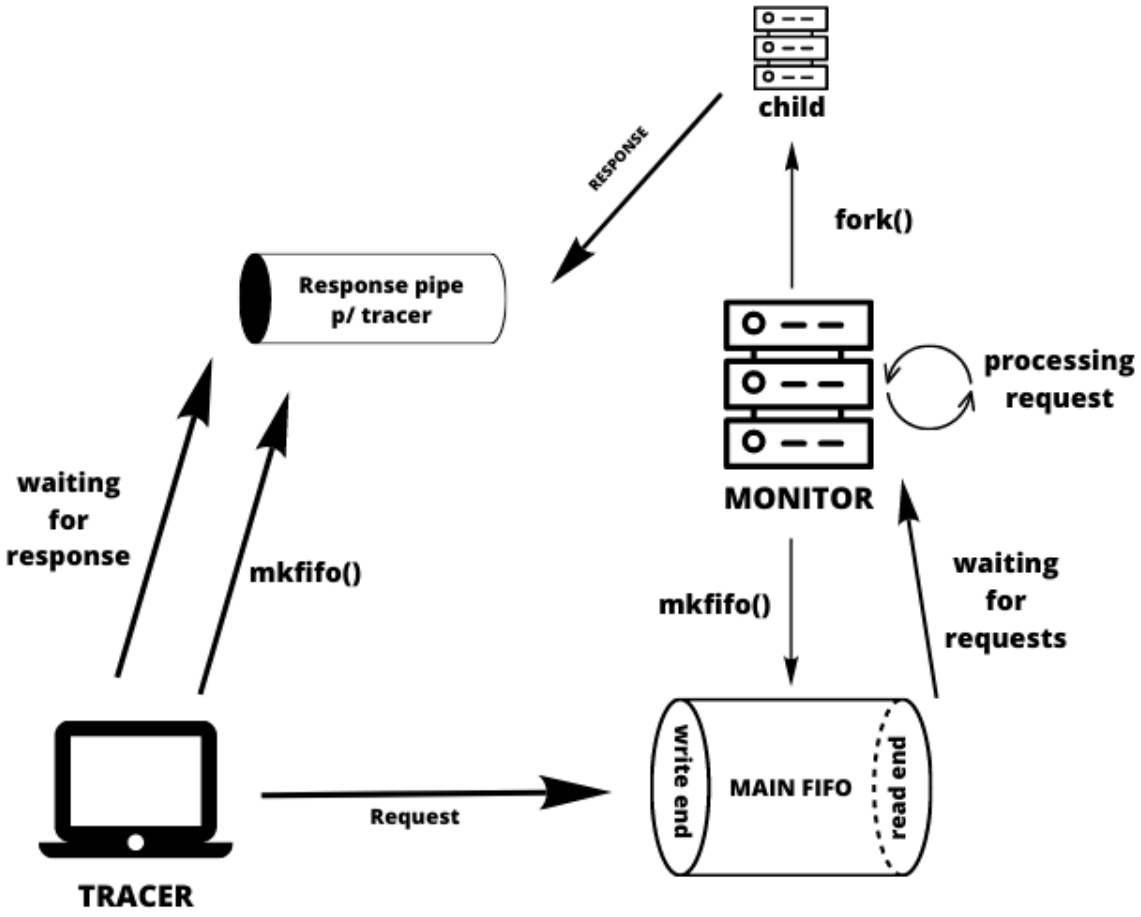

Figura 5.1: Arquitetura geral de comunicação *full-duplex tracer→monitor*

Figura 5.2: Execução de um comando

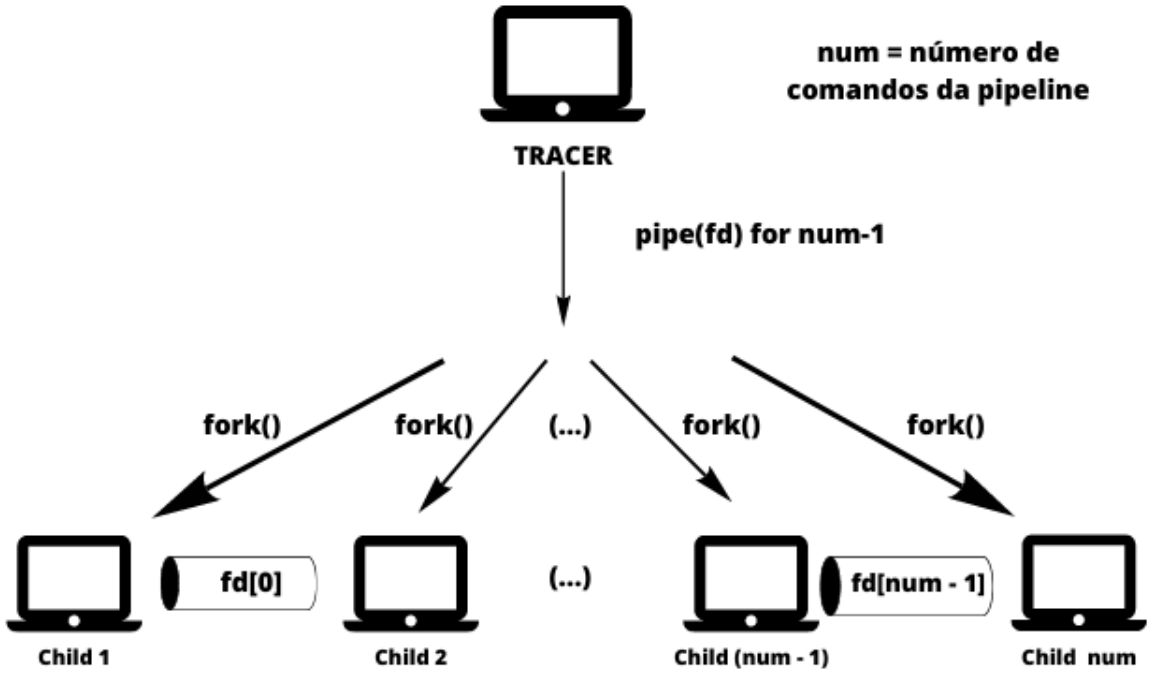

Figura 5.3: Execução de uma pipeline de comandos

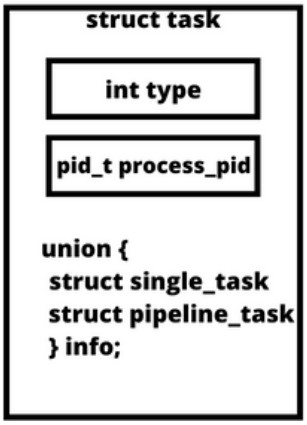

Figura 5.4: *struct task*

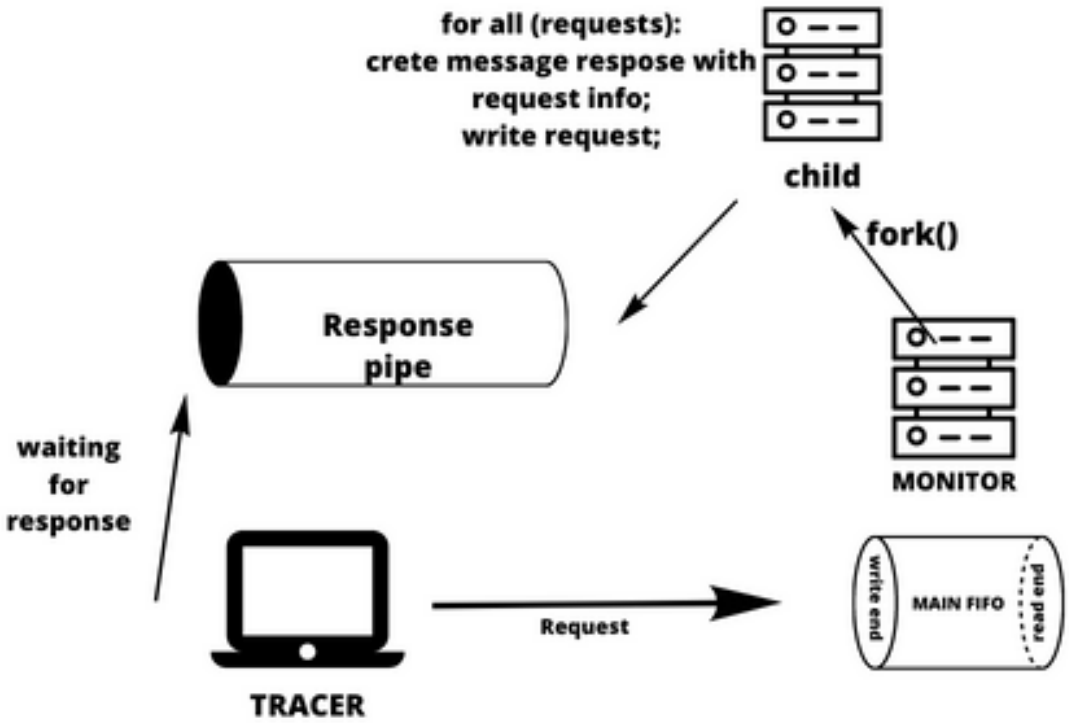

Figura 5.5: Arquitetura do comando *status*

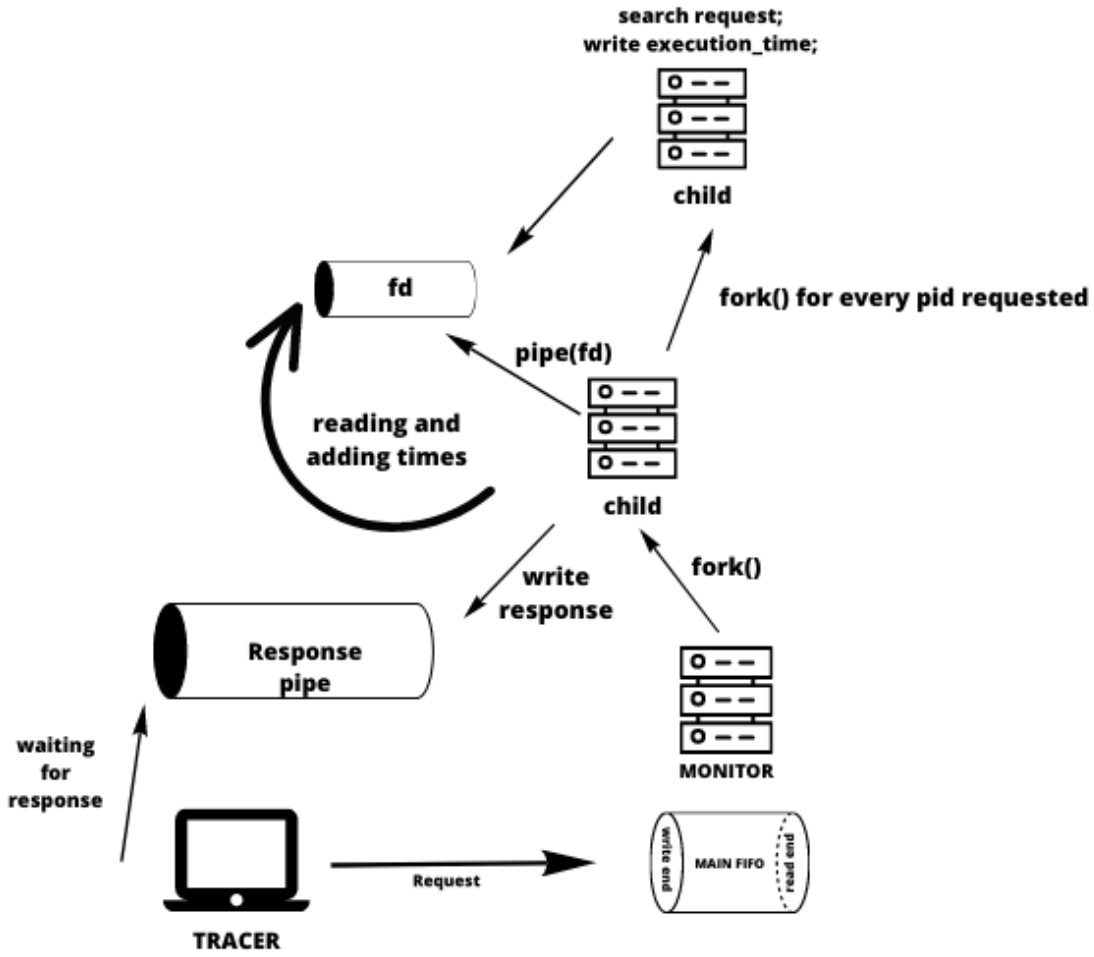

Figura 5.6: Arquitetura do comando *stats-time stats-command*

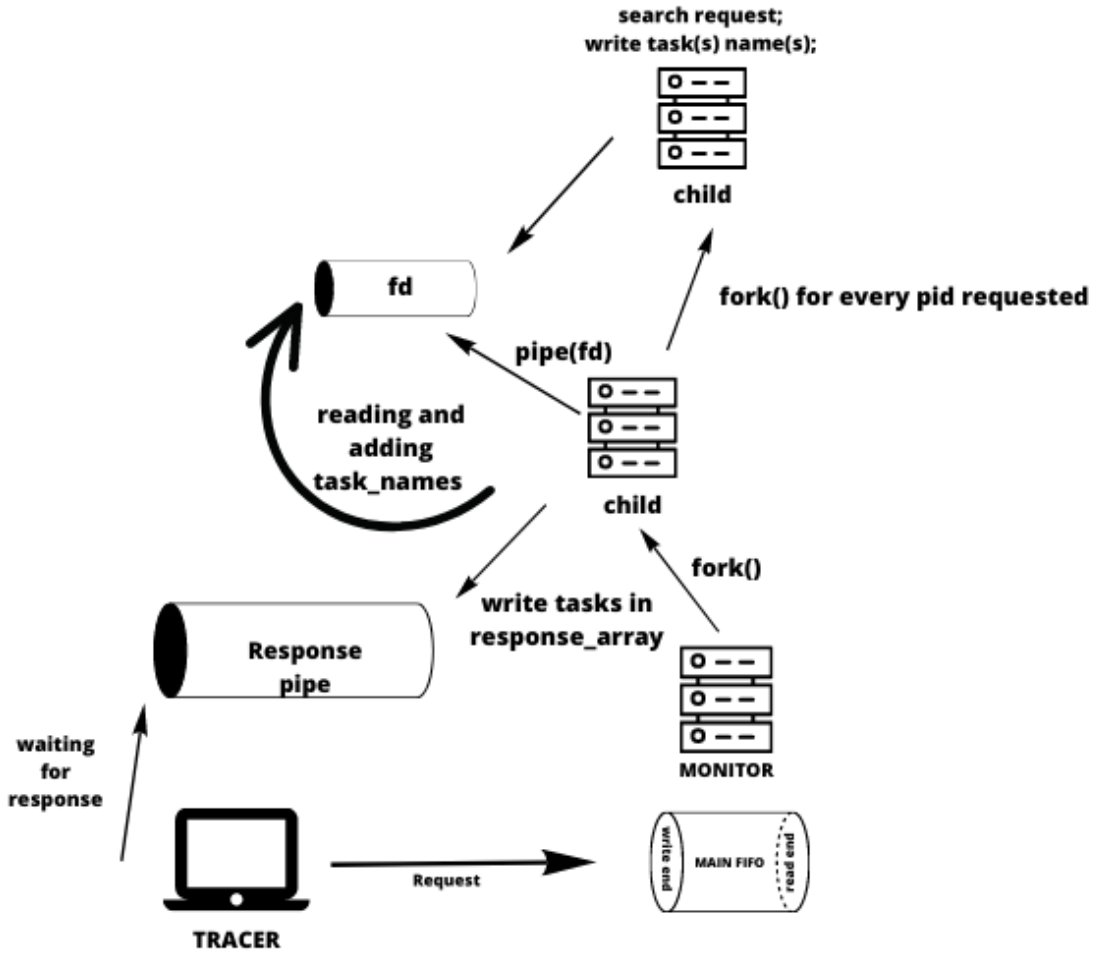

Figura 5.7: Arquitetura do comando *stats-uniq*