Sistemas Operativos

Orquestrador de Tarefas

Grupo 31

Universidade do Minho
2024/05/07

[A104000] Luis Enrique Diaz De Freitas
[A104001] Frederico Cunha Afonso
[A104002] André Barbosa Teixeira

Índice

Introdução

Este projeto visa a criação de um serviço de orquestração (execução e escalonamento) de tarefas num computador, mais concretamente. O serviço envolve o uso de um programa cliente (por utilizador) e um programa servidor (orquestrador).
Cada cliente deverá ser capaz de se comunicar com o servidor através do uso de pipes com nome. Este deverá escalonar e executar Tarefas que lhe foram propostas, sendo que o standard output/error de cada uma destas serão redirecionadas para um ficheiro cujo nome corresponde ao identificador da tarefa (este dado pelo servidor). Para além disto também terá de ser possível verificar o estado de cada Tarefa ("Agendada", "Executando" e "Concluída").
Neste relatório iremos expor as ideias, funcionalidades e testes unitários concebidos e executados, e finalmente, a arquitetura e implementação criada.

Funcionalidades

Após uma breve análise do enunciado, chegámos à conclusão que a melhor maneira de abordar o projeto seria ao cumprir com as funcionalidades propostas, sendo estas:

- O uso de um programa cliente como interface para cada utilizador e o uso de um servidor (orchestrator) que comunique com o cliente (e vice-versa) via pipes com nome;
- Execução de Tarefas propostas por um utilizador;
- Execução encadeada de Programas (por pipes anônimos);
- Registo e consulta de Tarefas agendadas, em execução e terminadas;
- Processamento de várias Tarefas em paralelo.

Como resposta a duas das funcionalidades anteriormente propostas, fizemos com que o utilizador possa requisitar ao cliente que:

- Execute uma dada Tarefa através do comando execute;
- Verifique o estado de todas as Tarefas propostas até à altura através do comando status;
- Termine com o processo do servidor através do comando break.

Execute

De maneira ao utilizador usar este comando corretamente, terá de escrever no terminal o seguinte formato:

```
./client execute [time] [flag] [task]
```

Onde o utilizador teria de substituir o:

- **`[time]`** por um número, representante do tempo estimado que a tarefa demoraria a ser executada;
- **`[flag]`** pelo tipo de Tarefa que seria proposta.
    - Se esta envolvesse a execução de mais que um programa, teria de ser substituída por "-p", caso contrário teria de ser substituída por "-u";
- **`[task]`** pela Tarefa a ser executada, sendo que esta é simplesmente um conjunto de comandos que seriam capazes de ser executados pelo terminal em bash
  (p.e. "`cat fich1 | grep "palavra" | wc -l`");

Status

De forma a verificar o estado de todas as Tarefas que foram guardadas pelo servidor (numa dada pasta) até à altura, o utilizador terá de usar o seguinte comando:

```
./client status
```

Break

Para além do que foi pedido, acreditámos que fosse boa prática da nossa parte adicionar uma interação extra entre o cliente e servidor, uma que permitisse "fechar o servidor" de forma a libertar qualquer memória ocupada pelo ficheiro "orchestrator" (queue de Tarefas por executar, Tarefas a serem executadas, nome da pasta em que os Outputs serão guardados e identificador da política de escalonamento) e a terminar qualquer processo proveniente de um fork.

```
./client break
```

Arquitetura

Client

Após uma análise dos requisitos impostos anteriormente, chegámos à conclusão que a melhor maneira de abordar o projeto, seria, primeiramente, criar um ficheiro correspondente ao programa cliente, que irá comunicar com o programa orchestrator através do uso de um pipe com nome. Mais concretamente, irá criar um pipe com o nome "server" e irá apenas ser capaz de escrever (mandar) mensagens nela e um pipe com um nome equivalente ao identificador do processo em que o programa está a ser identificado, que apenas é capaz de ler (receber) mensagens (as mensagens que são esperadas que o cliente receba, e mais tarde expostas no standard output, podem tanto ser o estado atual de todas as Tarefas pedidas até à altura, como também a confirmação de que o servidor recebeu uma Tarefa).

O cliente só é capaz de lidar com as interações previstas do projeto (todas estas serão mencionadas e explicadas mais tarde), e servirá como interface para cada utilizador via linha de comandos.

Orchestrator

O servidor (programa orchestrator), terá sido a parte mais complicada de se conceber, mas, com o auxílio da struct Server, TTL, Tasks e Progs, fomos capazes de simplificar a situação em causa.
O servidor é capaz de:

- Executar Tarefas comunicadas por um cliente através do uso de forks e pipes anônimos (que serão usados para guardar o output de cada Programa, que não o último, de uma lista de Programas de uma dada Tarefa com a flag "-p"). No nosso caso, caso uma Tarefa tivesse N Programas, seriam usadas N-1 pipes de forma ao código ser-se mais fácil de perceber;
- Registar Tarefas agendadas, em execução e terminadas num ficheiro binário presente na pasta em que os outputs serão guardados;

- Guardar, de forma ordenada, Tarefas numa fila de espera caso o número total de Tarefas a serem executadas tenha alcançado o limite proposto (de forma a respeitar a política de escalonamento apresentada);
- Processamento de várias tarefas em paralelo;
- Criar e guardar o nome da pasta em que será guardada toda a informação que seria mandada ao standard output/error por cada Tarefa executada.

De tudo o mencionado anteriormente, é importante salientar que o Registo de Tarefas é feito 4 bytes a seguir ao início do ficheiro, sendo que são guardados nestes uma variável int que guardará a quantidade de Tarefas até à altura pedidas e que, no final do Processamento de uma Tarefa, o servidor, no filho criado para executar a Tarefa, abre a pipe do servidor em modo de escrita (irá mandar), e irá enviar a mensagem "done". Quando o servidor recebe esta mensagem, irá diminuir o número de Tarefas a serem executadas e irá esperar que o processo filho termine corretamente (evitando a criação de um processo "Zombie").

Implementação

Interações cliente/servidor

Execute

```
./client execute [time] [flag] [task]
```

Após o utilizador escrever esta linha de comando, o cliente (servindo como interface do utilizador) irá enviar ao servidor uma string que contém: o identificador do processo (em que o cliente está a ser executado), a flag usada, o tempo estimado e finalmente, a Tarefa solicitada.

O servidor, depois de ler o pedido do cliente (e realizar um parsing da string recebida), irá criar uma Tarefa baseada nessa string, responder ao cliente com o número que atribuiu ao identificador da Tarefa (mais tarde impresso no standard output em que o pedido foi inicialmente feito) e a seguir irá executar esta Tarefa (guardando qualquer output que fosse enviado ao standard output/error pela Tarefa num ficheiro .txt relacionado à Tarefa pedida). Mal o servidor receba a Tarefa, irá documentar o seu estado atual num ficheiro binário, que será mais tarde alterado (passando o estado de Agendado a Executando e finalmente num Concluído).

Apesar do que foi mencionado anteriormente, caso o número de Tarefas a serem executadas atualmente seja igual ao número de Tarefas que podem ser executadas em paralelo, a Tarefa construída não será executada de imediato, esta será guardada numa "lista de espera" (queue) e terá de esperar que todas as outras Tarefas que tenham sido lá guardadas sejam executadas.

Status

```
./client status
```

Após o utilizador escrever esta linha de comando, o cliente irá enviar ao servidor uma string que contém: o identificador do processo do cliente e o segundo argumento recebido. Assim que a string for recebida, o servidor irá ler todas as Tarefas escritas no ficheiro binário (ignorando os primeiros quatro bytes) e, finalmente, irá escrever uma string com toda a informação necessária de cada Tarefa registada (independentemente do estado em que se encontrem atualmente), esta será mandada de volta ao cliente, sendo mais tarde impressa no terminal do mesmo (a string terá as Tarefas ordenadas pelo seu estado atual, de "Agendado" a "Executando" a "Concluído").

Break

```
./client break
```

Após o utilizador escrever esta linha de comando, o cliente irá enviar ao servidor uma string com apenas o segundo argumento recebido. Visto que o servidor só se mantém ativo através do uso de uma variável (int) a que lhe foi atribuída o valor de 1, quando o servidor receber esta string, este irá simplesmente subtrair a variável por 1 e o servidor irá fazer um clean up de tudo o que foi guardado até à altura na memória.

Progs

O struct Prog irá guardar o caminho do programa que será executado ou o nome do comando a ser executado, a quantidade de argumentos relativos ao comando a ser executado (incluindo o nome do comando como primeiro argumento) e um array de strings correspondentes a todos os argumentos que foram recebidos para um dado programa.

Tasks

O struct Task tem associado um identificador único que lhe foi atribuído pelo servidor, um identificador de processo relativo ao cliente que comunicou a Tarefa, a string da flag recebida (quer seja "-u" ou "-p"), a quantidade de Progs guardados nela, seguido de um array de todos os Progs lá armazenados (tudo isto na memória), o tempo de duração estimado e real e o estado atual da Tarefa (variando de "Agendado", "Executando" e "Concluída").

TTL

Esta struct não passa de uma lista ligada de Tasks que tem-nas ordenadas pelo tempo de duração estimado ou pela altura em que foram recebidas (dependendo da política de escalonamento afirmada, sendo estas a **FCFS** - "First Come, First Served" e a **SJF** - "Shortest Job First").

Server

Finalmente, iremos tratar do struct Server, relativo ao servidor (programa orchestrator), que contém uma struct TTL, um contador de Tarefas a serem executadas atualmente, um contador de Tarefas já recebidas, um contador de forks já feitos só no ficheiro orchestrator.c para mais tarde fazer wait() N vezes antes do programa terminar, o nome da pasta onde serão guardados os outputs de cada Tarefa proposta e do ficheiro binário em que serão guardadas as Tasks alguma vez recebidas e o seu número, o número de tasks que podem ser feitas em paralelo e a política de escalonamento.

Testes Unitários

Após termos conseguido uma versão final do projeto, acabámos por criar alguns scripts que testassem a eficiência de cada política de escalonamento disponível, afirmando que para ambos casos, os servidores a serem criados, o número de Tarefas máximas a serem executadas em paralelo era de 2 (acreditámos ser mais conveniente para uma análise mais direta e simples dos resultados obtidos).

- **FCFS** - First Come, First Served
  Esta, apesar de ser a mais de implementar, não tem em noção o tempo de duração esperado e apenas percorre a queue inteira até chegar ao fim desta, guardando a Tarefa atual no fim (a queue podia ter sido melhor implementada, mas mesmo assim, esta não é a política de escalonamento mais eficiente).

- **SJF** - Shortest Job First
  Após alguns testes, fomos capazes de concluir que esta é a melhor política de escalonamento de ambas. Guarda a Tarefa atual de forma ordenada, dependendo do tempo de duração esperado, fazendo assim com que as Tarefas mais rápidas sejam executadas prontamente
  e deixando as que demorassem mais tempo para o final, evitando grandes obstruções.

Após executar as mesmas Tarefas a servidores semelhantes (só diferindo na política de escalonamento), percebemos que o servidor com a política de escalonamento **SJF** seria capaz de concluir as Tarefas que lhe foram propostas ligeiramente mais cedo que o que tinha a política **FCFS**.

Conclusão

Apesar de alguns aspetos que podiam ter sido melhorados com mais algum cuidado (respeito do encapsulamento, modularidade, etc.), após aplicarmos tudo o que nos foi lecionado da unidade curricular, fomos capazes de criar um projeto que cumprisse com tudo o que fora esperado pelos critérios expostos pelos professores.

Anexos

```
typedef enum{
      EXECUTING,
      SCHEDULED,
      COMPLETED
}Task_Status;

typedef struct{
      char *
path_to_program;

      short amount_args;
      char ** args;
}Prog;
```

```
typedef struct{
      int id;
      int pid;
      char * pipe_flag;

      short amount_programs;
      Prog ** programs;

      long estimated_duration;
      long real_duration;

      Task_Status status;
}Task;
```

```
typedef struct orchestrator{

      TTL* queue;
      int active_tasks;
      int log;

      long amount_forks;
      char * output_folder;
      int parallel_tasks;
      char * sched_policy;

}Server;
```

```
typedef struct TaskLinkedList{

      Task * task;
      struct TaskLinkedList * next;

}TTL;
```