Faculdade de Engenharia da Universidade do Porto

Mestrado Integrado em Engenharia Informática e Computação

Application Server
Server Side

Sistemas Operativos

2º ano, 2º semestre

Turma 6, Grupo 1

Sérgio da Gama up201906690

Rui Moreira up201906355

José Silva up201904775

Índice

1.Tratamento de argumentos

No servidor este tratamento teve de ser mais exaustivo do que no cliente, dado que existe um maior número de argumentos e, por consequência, um maior número de opções de *input* do utilizador. Deste modo, para verificar se este último é corretamente introduzido, é passado por um conjunto de condições que não deve validar, sendo que se o fizer para pelo menos uma delas, algo está errado e é de imediato demonstrada a forma correta de utilização do programa ao utilizador, seguido do término do mesmo. As condições são baseadas no número de argumentos que pode ter, ou 4 ou 6, incluindo o nome do programa. Para além disso, é sempre obrigatório que o argumento 1 seja o "-t" e que o argumento seguinte seja um número válido. No caso de serem 6 argumentos, é obrigatório o argumento 3 ser "-l", também seguido de um número válido. O diretório do fifo público vem sempre em último e pode ser relativo ou completo.

2.*Multithreading*

No decorrer do programa existem três tipos principais de threads, os produtores, o consumidor e a thread principal (*main thread*), sendo que apenas existe uma de cada um dos últimos dois tipos. A inicialização do programa e tratamento dos argumentos são efetuados na thread principal. Contudo, sempre que existe um pedido por parte do cliente, é criada uma thread produtora, que, tal como o nome indica, produz o resultado para uma dada tarefa pretendida pelo cliente, através da chamada à função da biblioteca B. De seguida, disponibiliza o resultado e a restante informação necessária (utilizando um buffer específico abordado no tópico de 5.Gestão de Memória) para a única thread consumidora. Esta última encontra-se responsável por enviar os resultados para o cliente através do *fifo* privado criado pelo mesmo, procedimento que também será melhor abordado no tópico seguinte.

3.Comunicação servidor - cliente

Os pedidos do cliente são lidos através do *fifo* público, criado pelo servidor no início do programa. De seguida, para este comunicar com o cliente, utiliza a mesma *struct* que o cliente (*typedef struct message { int rid; pid_t pid; pthread_t tid; int tskload; int tskres; } message_t;*). Deste modo, cada mensagem enviada entre os dois, tem o mesmo formato. Para, efetivamente, enviar a mensagem, este tem de reconstruir o diretório do *fifo* privado e abri-lo em modo de escrita, seguida então do envio.

4.Evitamento de *deadlocks/busy-waiting*

De modo a serem evitados *deadlocks*, foram implementados os *mutexes* sempre que, quer a *producer thread*, quer a *consumer thread*, acediam ao *buffer* partilhado. Para além disso, ainda foram utilizados semáforos, com o intuito de controlar o acesso ao *buffer*. Assim, sempre que o *producer* produzia um dado número de respostas, definido pelo tamanho do *buffer* (dado com argumento), o *consumer* é notificado, podendo agir em conformidade.

5.Gestão de Memória

Nesta secção encontra-se descrita a parte da gestão de memória, tal como a estrutura de dados utilizada para funcionar como armazém (*buffer*). No âmbito de uma melhor organização da memória partilhada foi então desenvolvida um conjunto de funções associados à manipulação de um *queue* de *messages*. Esta trata-se de uma estrutura de dados *fifo*, (*first in first out*), dando prioridade às mensagens que entram primeiro. O acesso a esta *queue* é controlado através dos semáforos, tal como já foi referido anteriormente.

6.Sinais

Ao longo do projeto, apenas necessitamos de utilizar um tipo de sinal, o *SIGALARM*. Desta forma, assim que o tempo de execução do servidor terminar são efetuadas todas as ações necessárias para finalizar o programa, tal como apagar o *fifo* público, que não vai ser mais necessário, e remover a estrutura dos *mutexes*, abordados no tópico anterior. Adicionalmente, é também exposta uma mensagem que notifica o utilizador da chegada do *timeout* do servidor.

7.Saída do Programa

Tal como já foi referido anteriormente, a saída do programa pode ser induzida pelo *SIGALARM*, assim que o tempo de execução introduzido como *input*, tenha acabado. Assim que o programa termina, este liberta todos os recursos e termina todas as threads que ainda estão em execução (*producers*).

8.Registos

Os registos são efetuados tal e qual se encontra requisitado no enunciado, saindo todos as mensagens importantes através da saída padrão (stdout). Pelo contrário, tudo o que se relaciona com debug sai pela saída padrão do erro (stderr). O formato das mensagens é idêntico ao formato já utilizado nos clientes.

9.Auto-avaliação

Aluno	Nota (0-20)	Esforço (%)
Sérgio da Gama	18	33.3
Rui Moreira	18	33.3
José Silva	18	33.3