Licenciatura em Engenharia Informática, redes e telecomunicações

Sistemas Operativos

2º TRABALHO PRÁTICO

Docente: Nuno Oliveira

Trabalho realizado por:

- João Fatelo nº 51792
- Afonso San Miguel nº 51792
- Gonçalo Antão nº 51785

Turma: LEIRT41D

Semestre de verão 24/25

12/05/2025

Índice

Introdução

O segundo trabalho prático da unidade curricular de Sistemas Operativos teve como principal objetivo consolidar conhecimentos sobre a programação ao nível de sistema em ambientes UNIX/LINUX. Através de exercícios práticos, explorou-se a utilização de chamadas de sistema, a medição de tempos de execução a implementação de comunicação entre processos via sockets. A parte final do trabalho consistiu no desenvolvimento de um servidor, capaz de executar remotamente programas à escolha do cliente, com suporte para comunicação por sockets TCP e UNIX. Este relatório descreve as abordagens adotadas, as decisões de implementação e os resultados obtidos.

Grupo I

1.

ex1

Programa	Buffer (bytes)	Tempo (real)	Observações
fcopy	1	0m0,305s	
fcopy	64	0m0,041s	
fcopy	128	0m0,078s	
fcopy	256	0m0,026s	
fcopy	512	0m0,063s	
fcopy	1024	0m0,044s	
copy	1	0m0,631s	10485760 read() calls
copy	64	0m0,438s	163841 read() calls
copy	128	0m0,261s	81921 read() calls
copy	256	0m0,158s	40961 read() calls
copy	512	0m0,085s	20481 read() calls
copy	1024	0m0,083s	10241 read() calls

Figura 1 - Comparação dos tempos obtidos e das diferenças de desempenho observadas

Durante os testes foi observado que o tempo de cópia do ficheiro tende a diminuir com o aumento do tamanho do buffer, tanto para o fcopy como para o copy. Isto deve-se, pois, com um buffer maior são realizadas menos leituras e escritas, reduzindo assim o numero de chamos e o tempo gasto.

O copy usa chamadas direta do read e do write o que é mais custoso em termos de tempo de CPU, principalmente em buffers pequenos, pois cada read e write implica uma transição entre modo utilizador e modo kernel.

O fcopy usa fread e fwrite o que realiza buffering interno, reduzindo assim o número de chamadas reais ao sistema operativo. Embora os tempos não apresentem diferenças drásticas com buffers maiores, nos buffers pequenos a versão com a biblioteca fcopy tem claramente um melhor desempenho.

2.

a)

./test_func_times

R.: (Total elapsed time = 0.176011 s (average 0.00 us))

b)

./test_syscall_time

R.: (Total elapsed time = 0.079656 s (average 0.80 us))

c)

./test_proc_time

R.: (Total elapsed time = 0.563615 s (average 563.62 us))

d)

./test_exec_time

R.: (Total elapsed time = 0.947824 s (average 1895.65 us))

e)

. /test_thread_time

R.: (Total elapsed time = 24.909369 s (average 249.09 us))

3.

ex3

Versão	Termos	Nº Threads/Processos	Tempo (us)
Threads	1000000	2	2660
Threads	1000000	4	2460
Threads	1000000	8	3047
Threads	10000000	2	16401
Threads	10000000	4	16902
Threads	10000000	8	21765
Processos	1000000	2	7000
Processos	1000000	4	8000
Processos	1000000	8	9000
Processos	10000000	2	18000
Processos	10000000	4	22000
Processos	10000000	8	24000
Sequencial	1000000	-	7000
Sequencial	10000000	-	33000

Figura 2 - Tempos de execução das versões sequencial, com processos e com threads, para diferentes configurações.

A nova implementação com threads é executada com o comando:

. /pi-threads <num_terms> <num_threads>

Por exemplo:

./pi-threads 100 1

A implementação com threads, em geral, mostrou o melhor desempenho, sendo mais eficiente do que a versão com processos, sobretudo quando o número de termos é reduzido. Para valores mais elevados, a diferença de desempenho entre threads e processos diminui ligeiramente, mas as threads mantêm-se como a opção mais vantajosa.

Grupo II

4.

Comandos para testar as diferentes funcionalidades:

1) Servidor

Para iniciar o servidor:

. /server

Este servidor aceita conexões simultâneas via socket TCP ou UNIX, com suporte para múltiplos clientes e logging detalhado.

2) test_protocol_client

Para testar o protocolo manualmente com um único ficheiro:

. /test_protocol_client <ficheiro_entrada.jpg> <ficheiro_saida.jpg>

Exemplo:

. /test_protocol_client entradaISEL.jpg saida.jpg

O resultado será guardado automaticamente na pasta 'ReceivedFiles/saída.jpg'.

3) stress_client

Para simular múltiplos envios para o servidor:

. /stress_client <ficheiro_entrada.jpg> <numero_envios>

Exemplo:

. /stress_client entradaISEL.jpg 5

O resultado de cada envio será guardado como `ReceivedFiles/output_<PID>_<i>.jpg`.

3) evaluationClient

Para testar:

. /evaluationClient -c convert -a "-rotate 45 - -" -r entradaISEL.jpg

Os resultados serão guardados na pasta `ReceivedFiles/`.

Objetivo de cada ficheiro:

1) server.c

Lida com as conexões de clientes, parsing de cabeçalhos, execução do comando e envio da resposta.

2) log.c / log.h

Implementa o sistema de logging para ficheiro, com timestamp e tipo de mensagem (INFO, ERROR, DEBUG).

3) socket_utils.c / socket_utils.h

Fornece funções reutilizáveis para criar e aceitar sockets TCP e UNIX.

4) test_protocol_client.c

Cliente simples de teste para validar o protocolo com um único ficheiro.

5) stress_client.c

Cliente que envia o mesmo ficheiro várias vezes para testar a robustez do servidor.

Estrutura do MakeFile:

1. Define variáveis: compilador (gcc), flags (-Wall -g) e objetos comuns.
2. Primeiro compila os módulos comuns (log.o e socket_utils.o).
3. Depois compila os binários: server, test_protocol_client, stress_client.
4. A regra `clean` remove os binários, objetos e a pasta de resultados.

Esta ordem garante que todas as dependências são compiladas primeiro, e evita recompilar ficheiros desnecessários.

Grupo II – Escolha múltipla

1.

Para dividir o processamento por ações concorrentes de forma a maximizar a utilização de toda a capacidade de processamento do *hardware*.	V
Para poder executar dois programas (ficheiros executáveis) diferentes em concorrência e de uma forma mais rápida.	F
Para poder realizar operações I/O em simultâneo com outras operações num mesmo processo.	V
Para ter a execução de dois troços de código concorrentes com espaços de endereçamento separados num mesmo processo.	F

Figura 3 - Escolha múltipla - ex1

2.

Os sockets do domínio UNIX e os fifos são identificados através de um ficheiro especial no sistema de ficheiros.	V
O mecanismo de comunicação fifo (named pipe) apenas funciona entre processos com grau de parentesco.	F
Nos sockets stream a função bind serve para associar o socket a uma tarefa	F
Os sockets são representados ao nível do núcleo do sistema operativo como um tipo de ficheiro e podem ser usados para o redireccionamento de I/O e a receção e envio de dados realizados através das funções de read() e write().	V

Figura 4 - Escolha múltipla - ex2

3.

<table>
<tr><td rowspan="2"><pre>int main () {
  int s = tcp_serversocket_init(HOST, PORT);

  while (1) {
    int ns = tcp_serversocket_accept(s);
    handle_client(ns);
  }
  return 0;
}</pre></td><td>Servidor disponível através de um socket no domínio internet atendendo múltiplos clientes em sequência.</td><td>V</td></tr>
<tr><td>Servidor disponível através de um socket no domínio internet atendendo múltiplos clientes em concorrência.</td><td>F</td></tr>
<tr><td rowspan="2"><pre>void `thHandleCient (void *arg) {
  int ns = *((int *)arg);
  handle_client(ns);
  return NULL;
}

int main () {
  int s = tcp_serversocket_init(HOST, PORT);
  pthread_t th;
  while (1) {
    int ns = tcp_serversocket_accept(s);
    int *ps = malloc(sizeof(int));
    *ps = ns;
    pthread_create(&th, NULL,
                   thHandleClient, ps);
    pthread_join(th);
  }
  return 0;
}</pre></td><td>Servidor disponível através de um socket no domínio internet atendendo múltiplos clientes em concorrência.</td><td>F</td></tr>
<tr><td>Servidor disponível através de um socket no domínio internet atendendo múltiplos clientes em sequência.</td><td>F</td></tr>
</table>

Figura 5 - Escolha múltipla - ex3

Conclusão

A realização deste trabalho prático permitiu aplicar vários conceitos fundamentais da programação de sistemas operativos, como a criação de threads, o uso de sockets para comunicação em rede, e o desenvolvimento de servidores. Verificou-se, através de testes, o impacto do tamanho de buffer na performance, a diferença de custo entre chamadas de função e chamadas de sistema, e o comportamento de soluções concorrentes com múltiplas tarefas. A construção do servidor remoto com suporte para múltiplos clientes reforçou a compreensão da comunicação interprocessos e da sincronização de tarefas, consolidando assim competências essenciais para o desenvolvimento de software de baixo nível em ambientes UNIX/Linux.