Laboratório 5: Comunicação entre Processos (IPC)

1. Objetivos

- Entender os principais mecanismos de comunicação entre processos UNIX/LINUX.
- Desenvolver aplicações cooperativas ou concorrentes usando mecanismos de IPC.

2. Materiais

- Distribuição Linux/Unix
- Ambiente de desenvolvimento para C/C++.
- Comandos do sistema e bibliotecas de programação.

3. Atividades

1. Para cada exercício a seguir, faça uma versão usando **pipe** e outra usando **fifo**.

   a) Faça um programa que lê strings digitadas pelo usuário e envia para outro programa que recebe essas strings e exibe na tela: a string, o tamanho, o número de consoantes, o número de vogais e o número de espaços.

   b) Faça um programa que lê uma expressão matemática simples (+, -, *, /) e passe para outro programa que realiza o cálculo e devolve a resposta.

2. Resolva os exercícios a seguir usando **sinais**:

   a) Faça um programa que lê atributos de configuração de inicialização (p. ex.: diretório padrão, dono, ...) de um arquivo e, ao receber o **signal 1 (SIGHUP)**, refaz a leitura desse arquivo e modifica as variáveis internas. Para provar que funciona, faça um menu com a opção para imprimir os atributos carregados na leitura.

   b) Faça um programa que manipule arquivos (ler e escrever) e que, ao receber o **signal 2 (SIGINT)** ou **signal 15 (SIGTERM),** faça uma finalização limpa (*graceful stop*) – armazenar as informações pendentes e fechar o arquivo.

   c) Faça um programa que recebe o **signal 2 (SIGINT)** e dispara um alarme para finalizar sua execução dentro de 10 segundos. Nesse intervalo, o programa deve ficar exibindo na tela uma contagem regressiva de 1 em 1 segundo.

3. Implemente um exemplo com memória compartilhada que possibilite e a troca de informação usando uma `struct` com ao menos dois tipos diferentes de dados (ex: `struct Livro {char titulo[40]; char autor[30]; int num_paginas;}`). Pode ser no modelo de produtor-consumidor (um processo produz e outro consome).

4. Implemente um programa que realize a soma de vetores utilizando processos para fazer o cálculo, mas com os vetores sendo compartilhados pelos processos. Como os espaços de memória entre os processos são isolados, um mecanismo fornecido pelo SO deve ser usado. No caso, use **memória compartilhada** para que todos os filhos operem sobre os dados, e **pipes** para a realização do despacho de trabalho (intervalo de índices no vetor). O número de elementos do vetor e o número de processos filhos deve ser fornecido pelo usuário. Por exemplo, *numElementos = 1000* e *numProcessos = 5*, cada filho processará 200 índices; para *numElementos = 1000* e *numProcessos = 4*, cada filho processará 250 índices.

   Exemplo de soma:

   V1 = [1,2,3,4], V2 = [2,5,3,1], V3 = V1 + V2 = [3,7,6,5]

a) Para o desenvolvimento do trabalho, uma sugestão para o fluxo dos eventos é a seguinte:

- Pai cria os pipes para comunicação com filhos;
- Pai cria os filhos;
- Pai cria (aloca) memória compartilhada
  - memória para o vetor com os dados (dados);
  - e memória para que os filhos informem o término do trabalho (sinalização);
- Filhos bloqueiam aguardando dados no pipe;
- Pai escreve no pipe de cada filho o intervalo em que eles devem trabalhar;
- Filho acessa a memória compartilhada e faz o trabalho;
- Filhos avisam pai que acabaram e encerram;
- Pai aguarda todos acabarem, imprime o resultado e finaliza.

b) Para o filho, o fluxo é o seguinte:

- Lê o pipe (bloqueia);
- Acessa a memória compartilhada (dados) e faz o trabalho;
- Acessa a memória compartilhada (sinalização) e escreve que acabou;
- Filho faz limpeza e finaliza.

5. Faça dois programas para atuarem respectivamente como cliente e servidor para um serviço de tradução simples (só traduz palavras). O serviço de tradução deve receber um código especificando a língua de origem e destino (use o padrão ISO 639-1) e a palavra para traduzir (ex: **pt-en:cachorro**). Se não conseguir fazer a tradução devolve **ERROR:UNKNOWN**, caso contrário, a tradução. Por exemplo, traduzir de **pt-en** (português para o inglês) a palavra **cachorro**, devolve **dog**.

   a) Faça uma versão usando **sockets UNIX** e que delega o processamento para um processo filho, isto é, o cliente pode continuar enviando mensagens para traduzir até que envie a mensagem NO-NO como código.

   b) Faça uma versão usando filas de mensagens **mqueue**. O servidor continuamente processa a fila e devolve para o respectivo cliente a resposta da tradução.

6. Faça um Jogo da Velha que possibilite dois clientes se conectarem ao servidor usando **sockets**. No servidor, para cada cliente será criado um processo filho. Os processos filhos devem se comunicar via **pipe** para trocarem os lances dos jogadores. Ao final do jogo, os processos filhos devem ser finalizados. *Sugestão:* utilize a implementação do jogo da velha *velha.c* e *velha.h* para facilitar o desenvolvimento do projeto.

7. Faça um jogo de turnos (p. ex. jogo de tabuleiro) para que dois ou mais jogadores possam jogar entre si. Use os mecanismos de IPC para implementar o jogo.

8. Faça uma aplicação com sockets para que dois processos em máquinas distintas possam trocar um arquivo. Um processo envia e outro recebe o arquivo.

Instruções para entrega via Moodle:

Entregar as questões:

- Questão **1. (a ou b)** usando fifo. (ex1)
- Questão **2. (a ou b)** com sinais. (ex2)
- Questão **4.** (ex3)
- Questão **5. (a),** Questão **6**. ou Questão **7.** (ex4)

Colocar a solução de cada questão em uma pasta separada nomeada por ex1, ex2, ex3 e ex4, respectivamente.
Cada pasta deve conter o *Makefile* para o exercício e *README* (se necessário).
Inclua em todos os arquivos de código fonte um cabeçalho com a funcionalidade, autor(es) e data.
Adicione comentários antes dos nomes das funções descrevendo a finalidade e os parâmetros de entrada e saída.
Adicione comentários nos principais trechos de códigos do programa.
Compactar em um único arquivo (**tar.gz**) e enviar via Moodle.
Os exercícios podem ser feitos em até **4** pessoas.
