Exemplo

Operações de E/S simultâneas em várias fontes de dados com *threads*

Este exemplo:

- **Trata da leitura de várias fontes em simultâneo com *threads***
  - **Dois *named pipes* e o teclado**
- **Deve ser analisado juntamente com o exemplo anterior do mecanismo *select*.**
  - **Pretende-se mostrar uma abordagem *versus* a outra**
- **Cenário**
  Existe um processo que deve receber *input* vindo de dois *named pipes* e do teclado. Não se sabe qual destas fontes de dados terá dados em primeiro lugar e as operações de leitura são bloqueantes.

Estratégia usada:

- O processo começa com uma *thread* (como todos os processos)
- São lançadas duas *threads* para tratar da leitura dos *named pipes*
  - Cada *thread* trata de um *named pipe*
  - A função executada em ambas as novas *threads* é a mesma
    - O código é o mesmo porque as threasd fazem o mesmo – apenas variam os dados com que trabalham
    - Os dados nessa função (variáveis locais) são diferentes em cada uma dessas *threads* (são variáveis locais, cada thread tem a sua cópia)
    - Cada uma dessas novas *threads* é informada acerca de qual o *named pipe* que vai tratar através do parâmetro da função
      - Trata-se de um ponteiro para uma estrutura que tem a informação necessária ao algoritmo dessa função/*thread*
- Após o lançamento das duas *threads* o processo fica com três *threads* ao todo (a *thread* inicial e as duas que foram criadas)
  - Não existe o conceito de thread "*principal*"
- A *thread* inicial prossegue na função *main* e vai tratar do *input* do teclado (porque o programador assim o decidiu). As *threads* tratam dos *named pipes* (um *pipe* cada uma)

Ficheiros *header* envolvidos

```
#include <sys/types.h>      // mkfifo, open flags
#include <sys/stat.h>       // mkfifo, open flags
#include <fcntl.h>          // read
#include <unistd.h>         // unlink

#include <stdlib.h>         // exit, EXIT_SUCCESS, EXIT_FAILURE
#include <stdio.h>          // scanf, printf

#include <signal.h>
#include <string.h>         // strcpy, strlen, strcmp

#include <pthread.h>        // pthread_....
```

Funções auxiliares

- Encerra: encerra o programa libertando recursos (neste caso, os *named pipes*)
- trataCC: Atende o sinal SIGINT (no exemplo é acessório), encerrando o programa
- avisaErroESai: Imprime uma mensagem relacionada com um erro e encerra o programa

```
void encerra() {
   unlink("pipe_a");
   unlink("pipe_b");
   printf("\n");
   exit(EXIT_SUCCESS);
}

void trataCC(int s) {
   printf("\n ->CC<- \n\n");
   encerra();
   exit(EXIT_SUCCESS);  // Em princípio não atinge esta linha
}

void avisaErroESai(char * p) {
   perror(p);
   exit(EXIT_FAILURE);
}
```

O funcionamento das *threads* (neste exemplo) é definido por uma estrutura que tem informação acerca de:

- Descritor do *named pipe* que vai ser lido
- Informação (nome) do *named pipe* para efeitos informativos para o utilizador

```
typedef struct dados_pipes {
   char qual[10];
   int fd;
} ThrDados;        // podia-se acrescentar o ID da thread
```

Função da *thread*

- Esta função suporta as duas *threads* lançadas.
- A função (e a *thread*) mantêm-se em ciclo a ler dados de um *named pipe*.
- O ciclo não tem fim. A *thread* termina quando o processo termina.

```
void * trataPipes(void * p) {
   char * qual = ( (ThrDados *) p)->qual;
   int fd = ( (ThrDados *) p)->fd;
   char buffer[200];
   int bytes;
   while (1) {
      bytes = read(fd, buffer, sizeof(buffer));
      buffer[bytes] = '\0';
      if ( (bytes > 0) && (buffer[strlen(buffer)-1] == '\n') )
         buffer[strlen(buffer)-1] = '\0';
      printf("%s: (%d bytes) [%s]\n", qual, bytes, buffer);
      if (strcmp(buffer,"sair")==0)
         encerra();
   }
   return NULL;
}
```

Nota: A leitura de dados do *pipes* tem este aspecto "pesado" porque está a lidar com a gestão de '\n' e '\0' mas pode ser simplificada sem perda de funcionalidade. Deixa-se como desfio/TPC a sua simplificação

Função main

- Lança as threads para lidar com os named pipes
- Mantêm-se a ler o teclado

```
int main(int argc, char* argv[]) {
   char buffer[200];
   int bytes;
   int fd_a, fd_b;   // handles para os file descriptor dos pipes
   pthread_t tpipea, tpipeb;
   ThrDados tdados[2];     // estruturas com os dados das threads

   signal(SIGINT, trataCC); // p/ interromper via ^C (não era necec.)

   // cria os pipes
   mkfifo("pipe_a", 00777);
   mkfifo("pipe_b", 00777);

   // abre os pipes. RDRW vs RD – Recordar razões dadas na aula teor.
   fd_a = open("pipe_a", O_RDWR);
   if (fd_a == -1)
      avisaErroESai("Erro no open pipe_a");
   fd_b = open("pipe_b", O_RDWR);
   if (fd_b == -1)
      avisaErroESai("Erro no open pipe_b");

   // Thread "A": 1º prepara dados da thread, 2º lança a thread
   strcpy(tdados[0].qual, "Pipe A");   tdados[0].fd = fd_a;
   if (pthread_create(& tpipea, NULL, trataPipes, tdados  ) != 0)
      printf("Houve um problema a criar a thread 1 / Pipe A\n");

   // Thread "B": 1º prepara dados da thread, 2º lança a thread
   strcpy(tdados[1].qual, "Pipe B");   tdados[1].fd = fd_b;
   if (pthread_create(& tpipeb, NULL, trataPipes, tdados+1) != 0)
      printf("Houve um problema a criar a thread 2 / Pipe B\n");

   printf("ready\n");
   while (1) {
      scanf(" %199[^\n]", buffer);
      printf("Teclado: [%s]\n", buffer);
      if (strcmp(buffer,"sair")==0)
         encerra();
   }
   return 1;

  // em princípio não deve chegar aqui
  return EXIT_SUCCESS;
}
```