# Exemplo

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

Existem dois cenários típicos onde o uso de threads é vantajoso: i) a necessidade de um programa efetuar várias tarefas em simultâneo, e ii) a execução de um algoritmo que é paralelizável e se pretende obter mais desempenho aproveitando a existência de diversos núcleos/processadores. Este exemplo encaixa na primeira situação: sendo as operações de E/S bloqueantes, enquanto o programa dá atenção a uma fonte de dados, não consegue atender às outras fontes de dados. A solução mais adequada a este tipo de problemas é o uso de threads.

**Este exemplo:**

- **Trata da leitura de várias fontes em simultâneo com *threads***
  - **Dois *named pipes* e o teclado (stdin)**
- **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)

**IMPORTANTE: A primeira implementação tem falhas. Mas primeiro vê-se essa solução para perceber que falhas são essas, e depois é apresentada uma segunda solução melhorada.**

**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*)
- **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);  // devia mandar encerrar as threads e depois
}                       // fazer join às threads ( -> resolver isto)

void avisaErroESai(char * p) {
   perror(p);           // está a sair sem fechar/libertar recursos
   exit(EXIT_FAILURE);  // e threads ( -> resolver isto)
}
```

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);
   }
   return NULL;
}
```

Nota: A leitura de dados do *pipes* tem este aspecto "pesado" porque está a lidar com a gestão de '\n' e '\0'. Poderia ser simplificada num cenário em que existe uma maior garantia ou conhecimento prévio acerca dos caracteres que vão efetivamente ser recebidos.

**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
```

```
   // cria os pipes
   mkfifo("pipe_a", 0777);
   mkfifo("pipe_b", 0777);



   // abre os pipes. RDRW vs RD – Recordar razões dadas na aula teórica
   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();
   }

  // em princípio não deve chegar aqui. No entanto fica a faltar
  // fazer join às threads, o que deve sempre ser feito
  //   ( -> resolver isto )
  return EXIT_SUCCESS;
}
```

**Deficiências na implementação apresentada atrás**

- O Código apresentado atrás representa uma possível ***primeira versão*** para resolver o cenário apresentado (um programa que consegue ler de dois pipes e do utilizador, podendo este terminar a qualquer momento com a palavra "sair", e sem usar select)

- No entanto, o programa tem algumas deficiências importantes que é necessário resolver

- É importante analisar esta versão e perceber quais são essas deficiências como exemplo de problemas que realmente podem ser cometidos numa primeira versão de um exercício ou trabalho e aprender a **identificar** e **resolver** tais deficiências

**Os problemas são**

1. O programa encerra sem esperar que as *threads* terminem. Acerca dss threads, simplesmente é dito "*O ciclo não tem fim. A thread termina quando o processo termina.*". Isto não é uma forma aceitável de terminar *threads*: as *threads* devem ser "mandadas terminar" e o programa deve aguardar com *pthread_join* antes de ele próprio terminar.

2. A função encerra elimina os *pipes*, podendo estes estar a ser ainda usados pelas *threads* (relacionado com o problema anterior).

3. A função AvisaErroESai() não liberta recursos. Por exemplo, os *pipes* não são apagados.

4. Não é verificado para todos os recursos se efetivamente foram criados (exemplo: mkfifo)

**Para resolver estes problemas vai ser seguida a seguinte estratégia**

- Todas as operações de criação e abertura de recursos vão ser verificadas. Se a criação ou abertura de um recurso falhar e esse recurso for essencial, o programa termina, desfazendo as operações de criação e abertura, entretanto já realizadas.

- Vai ser crida e mantida uma estrutura que mantém o registo de todos os recursos que vão sendo criados (uma espécie de "livro de registo" – "log"). Cada recurso que é criado fica assinalado nesta estrutura. Se se mandar terminar o programa, vê-se nesta estrutura quais os recursos criados e apagam-se, pela ordem inversa com que foram criados.

- De igual forma, regista-se na estrutura referida atrás todos os recursos que estão em uso, de forma a fechar os recursos (por exemplo, os *pipes*) antes de os apagar. Mais uma vez, segue-se a lógica óbvia: a última coisa a ser criada é a primeira a ser apagada, e se o recurso X foi criado e de seguida aberto, então em primeiro lugar é fechado e só depois apagado.

- O ciclo de vida das *threads* passa a ser controlado pela estratégia habitual: uma variável "continua" cujo valor é inspecionada no ciclo principal da *thread*. Se outra parte do programa desejar indicar à *thread* que deve terminar, bastará colocar a variável "continua" dessa *thread* com o valor reconhecido como "não deve continuar". Esta variável fará parte dos dados de controlo da *thread*. Em caso algum deve ser forçado o término com *pthread_cancel*.

- Dado que as *threads* podem estar bloqueadas numa operação *read*, a função *main* irá recorrer à estratégia de escrever no *pipe* onde a *thread* está a ler, de forma desbloquear a operação *read* e com isso permitir à *thread* que se aperceba que a variável "continua" indica agora que não deve continuar (outra estratégia possível: enviar um sinal à *thread* em questão com *pthread_kill*).

**O código apresentado em seguida corresponde à solução com as deficiências identificadas atrás já corrigidas.**

**Os ficheiros header files são os mesmos**

```
#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_....
```

Definem-se duas constantes simbólicas para o nome dos names pipes para não ter literais espalhados pelo código. Esta melhoria tem a ver com qualidede de código-fonte e não com as deficiências apontadas.

```
#define PIPE_A "pipe_a"
#define PIPE_B "pipe_b"
```

A estrutura dos dados para controlo das threads passa agora a ter a flag “continua” para que as threads possam perceber que devem terminar assim que possível

- Note-se que, neste caso, essa flag não é suficiente dado que as threads poderão estar (e certamente estarão mesmo) bloqueadas num *read*, que será resolvido da forma explicada atrás.

```
typedef struct dados_pipes {
   char qual[10];
   int fd;
   int continua;
} ThrDados;
```

Estrutura para registar todos os recursos criados e abertos entretanto, de forma a que, em caso de necessidade de encerrar o programa, este saiba exatamente o que é que tem que fechar e apagar. Esta estrutura concretiza a ideia de “livro de registo” descrita a trás.

- O valor -1 vai ser usar para identificar um recurso/descritor/handle ainda não aberto ou criado

- À medida que o programa vai criando e abrindo recursos (ficheiros, pipes, threads, etc.), vai registando aqui os handles e identificadores de forma a que, mais tarde, ao encerrar o programa, se saiba exatamente que recursos devem ser fechados e apagados.

- Notar que a estrutura inclui um ponteiro para os dados de controlo de cada thread pois vai ser necessário aceder ao descritos de ficheiro do pipe de cada thread, assim como à variável continua dessa *thread*, dados esses que estão na estrutura de controlo das threads.

```
typedef struct {
   int pipe_a_criado;
   int pipe_b_criado;
   int pipe_a_fd;
   int pipe_b_fd;
   pthread_t thread_a_id;
   pthread_t thread_b_id;
   ThrDados * ptd_a;
   ThrDados * ptd_b;
} DadosContrl;
```

Função auxiliar para inicializar a estrutura de registo de recursos criados e abertos. Esta função ajuda a manter o código na função da main mais simples e legível

```
void zeraDadosControlo(DadosContrl * p) {
   p->pipe_a_criado = -1;
   p->pipe_b_criado = -1;
   p->pipe_a_fd = -1;
   p->pipe_b_fd = -1;
   p->thread_a_id = -1;
   p->thread_b_id = -1;
};
```

Função para encerrar o programa.

- Pode ser chamada a qualquer momento (por exemplo, quando é detetado um erro que não pode ser recuperado).
- Analisa a estrutura com o registo dos recursos criados ou abertos e fecha e apaga apenas aqueles que foram realmente abertos ou criados.
- Notar que a ordem de fecho/eliminação é do fim para o princípio, encerrando os recursos pela ordem inversa com que foram criados ou abertos.
- Nest caso a primeira coisa a fazer é indicar às threads (se estiverem de facto a correr)m para terminar, depois aguardar que realmente terminem (pthread_join), e só então se deve proceder ao fecho e eliminação dos pipies (que estariam em uso pelas threads se estas ainda estivessem a correr)
- A indicação para as threads terminarem é feita através de duas ações: colocar o valor “falso” na flag “continua”, e escrevendo qualquer cosia no named pipe da thread de forma a que o seu read, onde provavelmente estará bloqueada, desbloqueie, permitindo À thread aperceber-se do novo valor na flag “continua”

```
void fechaEncerra(DadosContrl * pdc, const char * msg) {
   perror(msg);
   // -- encerra numa lógica última coisa aberta/criada é a primeira a fechar/apagar
   if (pdc->thread_a_id != -1) {
      printf("\n A encerrar thread A");
      pdc->ptd_a->continua = 0;          // thread, assim que puderes, sai
      write(pdc->pipe_a_fd, "etc", 4);      // 4 = strlen "etc" + \0 -> desbl read
      pthread_join(pdc->thread_a_id, NULL); // NULL -> nao quer o return val. da thread
   }

   if (pdc->thread_b_id != -1) {
      printf("\n A encerrar thread B");
      pdc->ptd_b->continua = 0;          // thread, assim que puderes, sai
      write(pdc->pipe_b_fd, "etc", 4);      // 4 = strlen "etc" + \0 -> desbl read
      pthread_join(pdc->thread_b_id, NULL); // NULL -> nao quer o return val. da thread
   }

   if (pdc->pipe_a_fd != -1)
      close (pdc->pipe_a_fd);
   if (pdc->pipe_b_fd != -1)
      close (pdc->pipe_b_fd);
   if (pdc->pipe_a_criado != -1)
      unlink(PIPE_A);
   if (pdc->pipe_b_criado != -1)
      unlink(PIPE_B);

   // Nota: havendo vários recursos do mesmo tipo, esta lógica devia estar num ciclo
   exit(0);
}
```

A função da thread *trataPipes* é idêntica à da versão anterior

```
void * trataPipes(void * p) {
   char * qual = ( (ThrDados *) p)->qual;
   ThrDados * pdados = (ThrDados *) p;
   char buffer[200];
   int bytes;
   while (pdados->continua) {
      bytes = read(pdados->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);
   }
   return NULL;
}
```

A função main é semelhante à da versão anterior do programa, sendo a principal diferença o facto de ir registando na estrutura de controlo de recursos abertos/criados cada coisa (*pipe*, *thread*, etc.) que vai abrindo.

```
int main(int argc, char* argv[]) {
   DadosContrl controlo;
   char buffer[200];
   int bytes;
   ThrDados tdados[2];     // estruturas com os dados das threads

   zeraDadosControlo(& controlo);

   // cria os pipes, registando o que vai criando e abrindo
      if (mkfifo(PIPE_A, 00777) != 0)
      fechaEncerra(& controlo, "mkfifo pipe A ");
   else
      controlo.pipe_a_criado = 1;
   if (mkfifo(PIPE_B, 00777) != 0)
      fechaEncerra(& controlo, "mkfifo pipe B ");
   else
      controlo.pipe_b_criado = 1;

   // abre os pipes. RDRW vs RD – Recordar razões dadas na aula teórica
   // vai registando tudo o que vai abrindo ou criando

   controlo.pipe_a_fd = open("pipe_a", O_RDWR);
   if (controlo.pipe_a_fd == -1)
      fechaEncerra(& controlo, "open pipe A ");
   controlo.pipe_b_fd = open("pipe_b", O_RDWR);
   if (controlo.pipe_b_fd == -1)
      fechaEncerra(& controlo, "open pipe B ");

   controlo.ptd_a = tdados;
   controlo.ptd_b = tdados+1;

   // threads
   // Thread "A": 1º prepara dados da thread, 2º lança a thread
   strcpy(tdados[0].qual, "Pipe A");
   tdados[0].fd = controlo.pipe_a_fd;
   tdados[0].continua = 1;
   if (pthread_create(& controlo.thread_a_id, NULL, trataPipes, tdados) != 0) {
      controlo.thread_a_id = -1;
      fechaEncerra(& controlo, "create thread A ");
   }

   // Thread "B": 1º prepara dados da thread, 2º lança a thread
   strcpy(tdados[1].qual, "Pipe B");
   tdados[1].fd = controlo.pipe_b_fd;
   tdados[1].continua = 1;
   if (pthread_create(& controlo.thread_b_id, NULL, trataPipes, tdados+1) != 0) {
      controlo.thread_b_id = -1;
      fechaEncerra(& controlo, "create thread B ");
   }

   printf("\ntudo pronto\n");
   while (1) {
      scanf(" %199[^\n]", buffer);
      printf("Teclado: [%s]\n", buffer);
      if (strcmp(buffer,"sair")==0)
         fechaEncerra(& controlo, "ordem para sair ");
   }
```

```
  // em princípio não deve chegar aqui ( sai pela fechaEncerra() )
  return EXIT_SUCCESS;
}
```

**Melhoria adicional** (de qualidade de código) que poderia ser feita: em vez de duplicar o código de criação e lançamento das *threads*, poder-se-ia fazer essas duas operações num ciclo *for*.