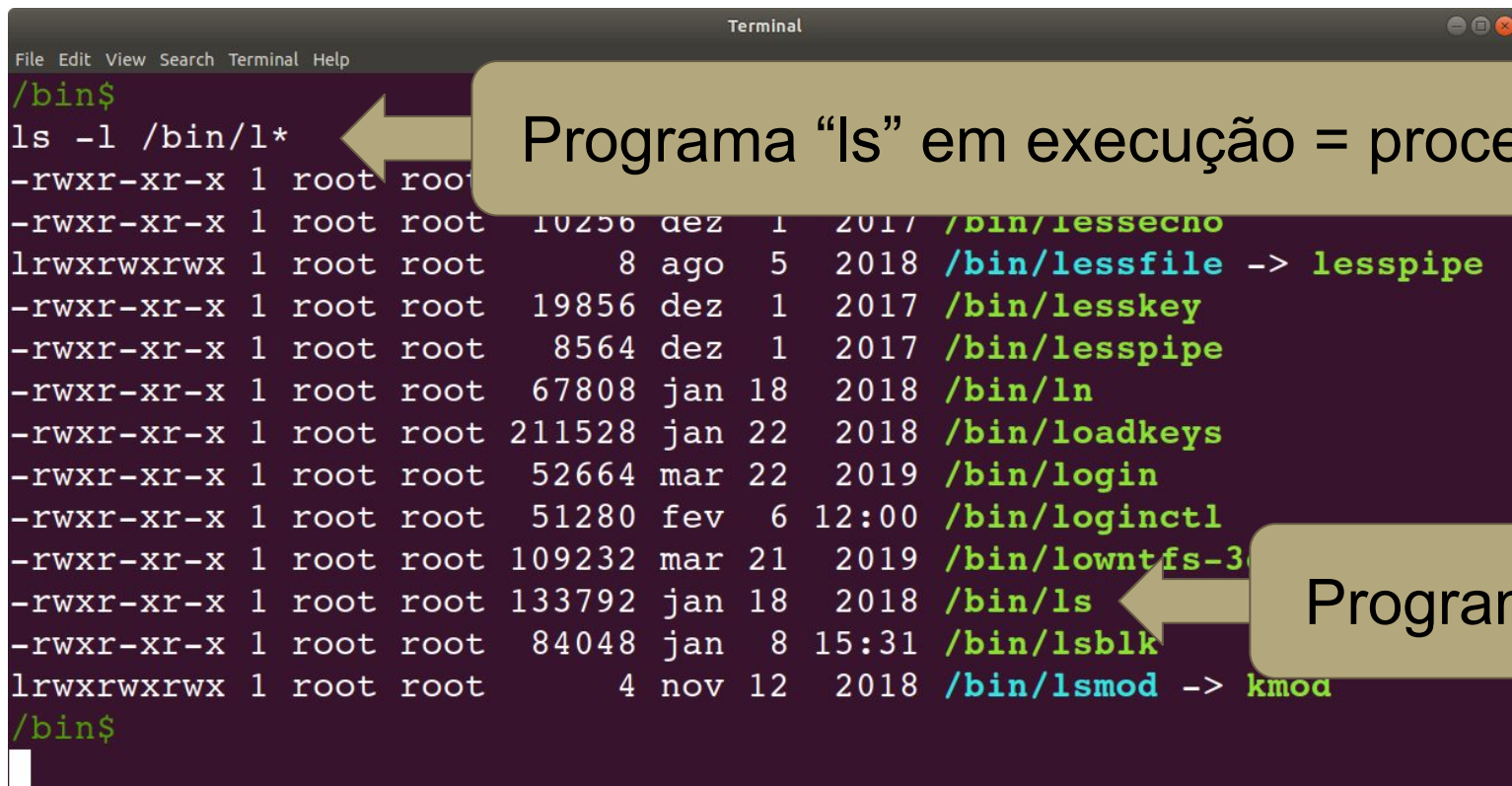


# Sistemas Operacionais Embarcados

Pipes, sinais e alarmes

# Processos

- **Definição:** um programa em execução
  - **Programa:** código em disco (passivo)
  - **Processo:** código sendo executado (ativo)



```
File Edit View Search Terminal Help
/bin$
ls -l /bin/l*
-rwxr-xr-x 1 root root 10256 dez 1 2017 /bin/less
lrwxrwxrwx 1 root root 8 ago 5 2018 /bin/lessfile -> lesspipe
-rwxr-xr-x 1 root root 19856 dez 1 2017 /bin/lesskey
-rwxr-xr-x 1 root root 8564 dez 1 2017 /bin/lesspipe
-rwxr-xr-x 1 root root 67808 jan 18 2018 /bin/ln
-rwxr-xr-x 1 root root 211528 jan 22 2018 /bin/loadkeys
-rwxr-xr-x 1 root root 52664 mar 22 2019 /bin/login
-rwxr-xr-x 1 root root 51280 fev 6 12:00 /bin/loginctl
-rwxr-xr-x 1 root root 109232 mar 21 2019 /bin/lowntfs-3
-rwxr-xr-x 1 root root 133792 jan 18 2018 /bin/ls
-rwxr-xr-x 1 root root 84048 jan 8 15:31 /bin/lsblk
lrwxrwxrwx 1 root root 4 nov 12 2018 /bin/lsmode -> kmod
/bin$
```

Programa “ls” em execução = processo

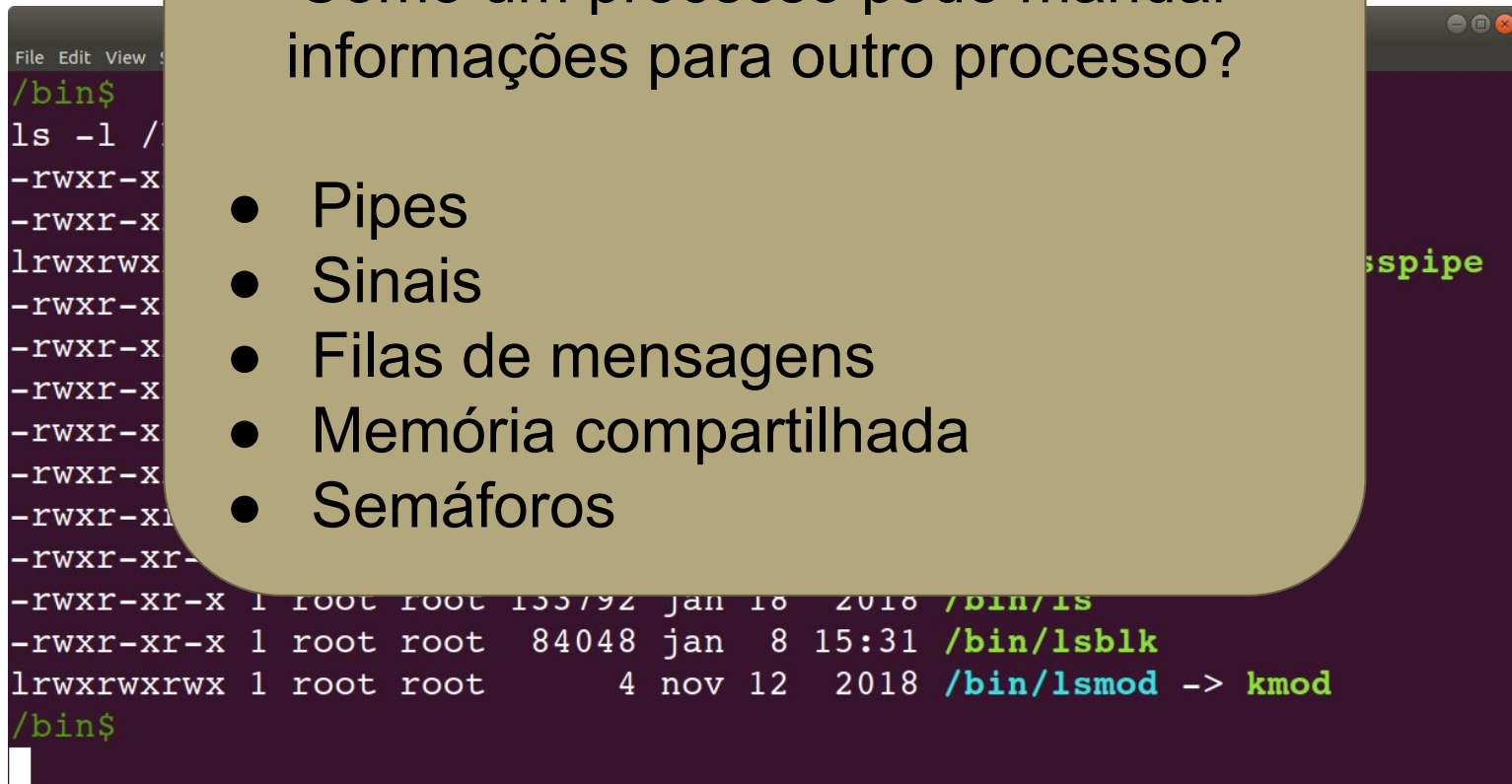
Programa “ls”

# Processos

- **Definição:** um programa em execução
  - **Programa:** código em disco (passivo)
  - **Processo:** código sendo executado (ativo)

Como um processo pode mandar informações para outro processo?

- Pipes
- Sinais
- Filas de mensagens
- Memória compartilhada
- Semáforos



A terminal window with a dark purple background and green text. The window title is 'File Edit View'. The prompt is '/bin\$'. The command 'ls -l /' is executed, showing a list of system files and directories with their permissions, owners, and sizes. The output includes entries for 'ls', 'lsblk', and 'lsmod'. The 'lsmod' command is also shown being executed, resulting in 'kmod'.

```
File Edit View :
/bin$
ls -l /
-rwxr-xr-x 1 root root 133792 Jan 18 2018 /bin/ls
-rwxr-xr-x 1 root root 84048 Jan 8 15:31 /bin/lsblk
lrwxrwxrwx 1 root root 4 Nov 12 2018 /bin/lsmod -> kmod
/bin$
```

# Pipes

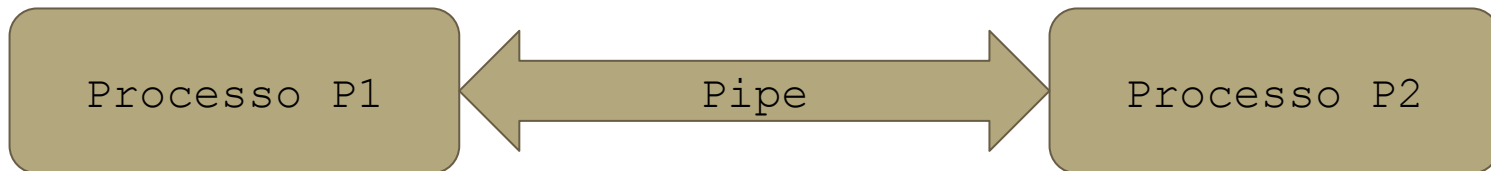
- Na linguagem de comando, os pipes são bastante utilizados:

```
~/Code/08_Pipes_Sinais_Alarmes $ ls | grep ".c"  
Ex1.c  
Ex2.c  
Ex3.c  
Ex4.c  
Ex5.c  
Ex6.c  
Ex7.c
```

- Podemos fazer o mesmo em programação UNIX para a comunicação entre processos.

# Pipes

- Os pipes são *buffers* protegidos em memória, acessados segundo a política FIFO (*first-in-first-out*).



```

#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <string.h>
int main()
{
    int pid;
    int fd[2];
    char mensagem[30];
    pipe(fd);
    pid = fork();
    if(pid == 0)
    {
        printf("Filho vai ler o
pipe\n");
        if(read(fd[0],mensagem,30)<0)
            printf("Erro na leitura do
pipe\n");
        else
            printf("Filho leu: %s\n",
                mensagem);
    }
}

```

```

else
{
    strcpy(mensagem, "HELLO
PIPE");
    printf("Pai vai escrever no
pipe\n");
    if(write(fd[1],mensagem,30)<
0)
        printf("Erro na escrita
do pipe\n");
    printf("Pai terminou de
escrever no pipe\n");
}
return 0;
}

```

(Continuação)

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <string.h>
```

```
int main()
```

```
{
```

```
    int pid;
```

```
    int fd[2];
```

```
    char mensagem[100];
```

```
    pipe(fd);
```

```
    pid = fork();
```

```
    if(pid == 0)
```

```
    {
```

```
        printf("Filho vai ler no
```

```
pipe\n");
```

```
        if(read(fd[0], mensagem, 100) < 0)
```

```
            printf("Erro na leitura do
pipe\n");
```

```
        else
```

```
            printf("Filho leu: %s\n",
                mensagem);
```

```
    }
```

```
else
```

```
{
```

```
    strcpy(mensagem, "HELLO
PIPE");
```

```
    printf("Pai vai escrever no
pipe\n");
```

```
    if(write(fd[1], mensagem, 30) <
0)
```

```
        printf("Erro na escrita\n");
```

```
    if(write(fd[1], mensagem, 30) < 0)
```

```
        printf("Erro na escrita\n");
```

```
    return 0;
}
```

**A função `pipe()` cria o *pipe* de comunicação, indicando dois descritores de arquivo, `fd[0]` e `fd[1]`, para leitura e escrita, respectivamente**

```

#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <string.h>
int main()
{
    int pid;
    int fd[2];
    char mensagem[30];
    pipe(fd);
    pid = fork();
    if(pid == 0)
    {
        printf("Filho vai ler o
pipe\n");
        if(read(fd[0],mensagem,30)<0)
            printf("Erro na leitura do
pipe\n");
        else
            printf("Filho leu: %s\n",
                mensagem);
    }
}

```

**Vamos criar um processo-filho para trocarmos informações com ele**

```

else
{
    strcpy(mensagem, "HELLO
PIPE");
    printf("Pai vai escrever no
pipe\n");
    if(write(fd[1],mensagem,30)<
0)
        printf("Erro na escrita
da mensagem\n");
}
}

```

hou de

(Continuação)



**O processo-pai coloca a string "HELLO PIPE" no vetor mensagem[ ]**



```
#inc
#inc
#inc
#inc
#include <string.h>
int main()
{
    int pid;
    int fd[2];
    char mensagem[30];
    pipe(fd);
    pid = fork();
    if(pid == 0)
    {
        printf("Filho vai ler o
pipe\n");
        if(read(fd[0],mensagem,30)<0)
            printf("Erro na leitura do
pipe\n");
        else
            printf("Filho leu: %s\n",
                mensagem);
    }
}
```

```
else
{
    strcpy(mensagem, "HELLO
PIPE");
    printf("Pai vai escrever no
pipe\n");
    if(write(fd[1],mensagem,30)<
0)
        printf("Erro na escrita
do pipe\n");
    printf("Pai terminou de
escrever no pipe\n");
}
return 0;
}
```

(Continuação)

O processo-pai escreve o conteúdo do vetor `mensagem[]` no descritor de arquivos `fd[1]`.

Ou seja, ele escreve o conteúdo do vetor no *pipe* aberto anteriormente

```
pipe(fd);
pid = fork();
if(pid == 0)
{
    printf("Filho vai ler o pipe\n");
    if(read(fd[0],mensagem,30)<0)
        printf("Erro na leitura do pipe\n");
    else
        printf("Filho leu: %s\n", mensagem);
}
```

```
else
{
    strcpy(mensagem, "HELLO PIPE");
    printf("Pai vai escrever no pipe\n");
    if(write(fd[1],mensagem,30)<0)
        printf("Erro na escrita do pipe\n");
    printf("Pai terminou de escrever no pipe\n");
}
return 0;
}
```

(Continuação)

```

#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <string.h>
int main()
{
    int pid;
    int fd[2];
    char mensagem[30];
    pipe(fd);
    pid = fork();
    if(pid == 0)
    {
        printf("Filho vai ler o
pipe\n");
        if(read(fd[0],mensagem,30)<0)
            printf("Erro na leitura do
pipe\n");
        else
            printf("Filho leu: %s\n",
                mensagem);
    }
}

```

```

else
{
    strcpy(mensagem, "HELLO
PIPE");
    printf("Pai vai escrever no
pipe\n");
    if(write(fd[1],mensagem,30)<
0)

```

Enquanto isso, o processo-filho lê 30 *bytes* do descritor de arquivos `fd[0]`, salvando o conteúdo no vetor `mensagem[ ]`

Ou seja, ele lê do *pipe* aberto anteriormente os *bytes* enviados pelo processo-pai

```
#include <...>
#include <...>
#include <...>
#include <...>
#include <...>
int main
{
```

```
    int p[2];
    int i;
    char mensagem[30];
    pipe(p);
    pid_t pid;
    if (pipefail < 0)
```

```
    {
        printf("Erro na criação do pipe\n");
        return 1;
    }
    if (read(fd[0], mensagem, 30) < 0)
        printf("Erro na leitura do pipe\n");
    else
        printf("Filho leu: %s\n", mensagem);
}
```

**Após abertos pela função `pipe()`, os descritores de arquivos `fd[0]` e `fd[1]` são usados para leitura e escrita do *pipe*, respectivamente. Essa ordem deve ser seguida pelos processos pai e filho, e por quaisquer outros que enxergam o *pipe***

```
else
{
    strcpy(mensagem, "HELLO PIPE");
    printf("Pai vai escrever no pipe\n");
    if (write(fd[1], mensagem, 30) < 0)
    {
        printf("Erro na escrita do pipe\n");
        printf("Pai terminou de escrever no pipe\n");
    }
    return 0;
}
```

(Continuação)

```

#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <string.h>
int main()
{
    int pid;
    int fd[2];
    char mensagem[30];
    pipe(fd);
    pid = fork();
    if(pid != 0)
    {
        printf("Pai vai ler o pipe\n");
        if(read(fd[0],mensagem,30)<0)
            printf("Erro na leitura do
pipe\n");
        else
            printf("Pai leu: %s\n",
                mensagem);
    }

```

```

else
{
    strcpy(mensagem, "HELLO
PIPE");
    printf("Filho vai escrever
no pipe\n");
    if(write(fd[1],mensagem,30)<
0)
        printf("Erro na escrita
do pipe\n");
    printf("Filho terminou de
escrever no pipe\n");
}
return 0;
}

```

(Continuação)

```

#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <string.h>
int main()
{
    int pid;
    int fd[2];
    char mensagem[30];
    pipe(fd);
    pid = fork();
    if(pid != 0)
    {
        printf("Pai vai escrever no pipe\n");
        if(write(fd[1], mensagem, 30) < 0)
        {
            printf("Erro na escrita no pipe\n");
        }
        else
        {
            printf("Pai leu: %s\n", mensagem);
        }
    }
}

```

```

else
{
    strcpy(mensagem, "HELLO PIPE");
    printf("Filho vai escrever no pipe\n");
    if(write(fd[1], mensagem, 30) < 0)
    {
        printf("Erro na escrita no pipe\n");
    }
    else
    {
        printf("Filho terminou de escrever no pipe\n");
    }
}

```

**Mesmo código que o anterior, exceto que é o processo-filho que manda uma mensagem para o processo-pai**

continuação)

```

#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <string.h>

int main()
{
    int pid;
    int fd[2];
    pipe(fd);
    pid = fork();
    if(pid == 0) //Codigo do filho
    {
        char buffer_filho[30];
        char msg_filho[30] = "FILHO DIZ:
HELLO PIPE";
        printf("Filho vai ler o pipe\n");
        if(read(fd[0],buffer_filho,30)<0)
            printf("Erro na leitura do
pipe\n");
        else
            printf("Valor lido pelo filho
= %s\n", buffer_filho);
        printf("Filho vai escrever no
pipe\n");
        if(write(fd[1],msg_filho,30)<0)
            printf("Erro na escrita do
pipe\n");
        printf("Filho terminou de escrever
no pipe\n");
    }

```

```

else //Codigo do pai
{
    char buffer_pai[30];
    char msg_pai[30] = "PAI DIZ:
HELLO PIPE";
    printf("Pai vai escrever no
pipe\n");
    if(write(fd[1],msg_pai,30)<0)
        printf("Erro na escrita do
pipe\n");
    printf("Pai terminou de
escrever no pipe\n");
    printf("Pai vai hibernar por 1
segundo, para dar tempo do filho ler
o pipe\n");
    sleep(1);
    printf("Pai vai ler o pipe\n");
    if(read(fd[0],buffer_pai,30)<0)
        printf("Erro na leitura do
pipe\n");
    else
        printf("Valor lido pelo pai
= %s\n", buffer_pai);
    }
    return 0;
}

```

(Continuação)

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <string.h>
```

```
int
{
```

**Neste código, o processo-pai manda uma informação para o processo-filho, e depois lê uma informação enviada pelo processo-filho**

```
else
    printf("Valor lido pelo filho
= %s\n", buffer_filho);
    printf("Filho vai escrever no
pipe\n");
    if(write(fd[1],msg_filho,30)<0)
        printf("Erro na escrita do
pipe\n");
    printf("Filho terminou de escrever
no pipe\n");
}
```

```
else // Código do pai
{
    char buffer_pai[30];
    char msg_pai[30] = "PAI DIZ:
HELLO PIPE";
    printf("Pai vai escrever no
pipe\n");
    if(write(fd[1],msg_pai,30)<0)
        printf("Erro na escrita do
pipe\n");
    printf("Pai terminou de
escrever no pipe\n");
    printf("Pai vai hibernar por 1
segundo, para dar tempo do filho ler
o pipe\n");
    sleep(1);
    printf("Pai vai ler o pipe\n");
    if(read(fd[0],buffer_pai,30)<0)
        printf("Erro na leitura do
pipe\n");
    else
        printf("Valor lido pelo pai
= %s\n", buffer_pai);
    }
    return 0;
}
```

(Continuação)



```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <string.h>
```

```
int main()
{
    int pid;
    int fd[2];
    pipe(fd);
    pid = fork();
    if(pid == 0) //Codigo do filho
    {
```

HELL

```
pipe\n");
    else
        printf("Valor lido pelo filho
= %s\n", buffer_filho);
        printf("Filho vai escrever no
pipe\n");
        if(write(fd[1],msg_filho,30)<0)
            printf("Erro na escrita do
pipe\n");
        printf("Filho terminou de escrever
no pipe\n");
    }
```

**Este atraso de 1 segundo foi usado para dar tempo ao processo-filho**

```
else //Codigo do pai
{
    char buffer_pai[30];
    char msg_pai[30] = "PAI DIZ:
HELLO PIPE";
    printf("Pai vai escrever no
pipe\n");
    if(write(fd[1],msg_pai,30)<0)
        printf("Erro na escrita do
pipe\n");
    printf("Pai terminou de
escrever no pipe\n");
    printf("Pai vai hibernar por 1
segundo, para dar tempo do filho ler
o pipe\n");
    sleep(1);
    printf("Pai vai ler o pipe\n");
    if(read(fd[0],buffer_pai,30)<0)
        printf("Erro na leitura do
pipe\n");
    else
        printf("Valor lido pelo pai
= %s\n", buffer_pai);
    }
    return 0;
}
```

(Continuação)

```

#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <string.h>

int main()
{
    int pid;
    int fd[2];
    pipe(fd);
    pid = fork();
    if(pid == 0) //Codigo do filho
    {
        char buffer_filho[30];
        char msg_filho[30] = "FILHO DIZ:
        HELLO PIPE";
        printf("Filho vai ler o pipe\n");
        if(read(fd[0],buffer_filho,30)<0)
            printf("Erro na leitura do
            pipe\n");
        else
            printf("Valor lido pelo filho
            = %s\n", buffer_filho);
        printf("Filho vai escrever no
        pipe\n");
        if(write(fd[1],msg_filho,30)<0)
            printf("Erro na escrita do
            pipe\n");
        printf("Filho terminou de escrever
        no pipe\n");
    }

```

```

else //Codigo do pai
{
    char buffer_pai[30];
    char msg_pai[30] = "PAI DIZ:
    HELLO PIPE";
    printf("Pai vai escrever no
    pipe\n");
    if(write(fd[1],msg_pai,30)<0)
        printf("Erro na escrita do
        pipe\n");
    printf("Pai terminou de
    escrever no pipe\n");
    printf("Pai vai hibernar por 1
    segundo, para dar tempo do filho ler
    o pipe\n");

```

**O processo-filho faz o  
inverso (primeiro  
recebe, depois manda)**

(Continuação)

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <string.h>
```

```
int main()
```

```
{
```

```
    int pid;
```

```
    int fd[2];
```

```
    pipe(fd);
```

```
    pid = fork();
```

```
    if(pid == 0) // Código do filho
```

```
    {
```

```
        char buffer_filho[30];
```

```
        char msg_filho[30] = "FILHO DIZ:
```

```
        HELLO PIPE";
```

```
        printf("Filho vai escrever no
```

```
        pipe\n");
```

```
        if(write(fd[1],msg_filho,30)<0)
```

```
            printf("Erro na escrita do
```

```
        pipe\n");
```

```
        else
```

```
            printf("Valor lido pelo pai
```

```
            = %s\n", buffer_filho);
```

```
            printf("Filho vai escrever no
```

```
            pipe\n");
```

```
            if(write(fd[1],msg_filho,30)<0)
```

```
                printf("Erro na escrita do
```

```
            pipe\n");
```

```
            printf("Filho terminou de escrever
```

```
            no pipe\n");
```

```
    }
```

**Ou seja, não há preferência sobre qual processo pode escrever ou ler no *pipe*.**

**O importante é que todos sigam a mesma lógica**

```
else // Código do pai
```

```
{
```

```
    char buffer_pai[30];
```

```
    char msg_pai[30] = "PAI DIZ:
```

```
    HELLO PIPE";
```

```
    printf("Pai vai escrever no
```

```
    pipe\n");
```

```
    if(write(fd[1],msg_pai,30)<0)
```

```
        printf("Erro na escrita do
```

```
    pai terminou de
```

```
    pipe\n");
```

```
    pai vai hibernar por 1
```

```
    dar tempo do filho ler
```

```
    pai vai ler o pipe\n");
```

```
    if(read(fd[0],buffer_pai,30)<0)
```

```
        printf("Erro na leitura do
```

```
    printf("Valor lido pelo pai
```

```
    = %s\n", buffer_pai);
```

```
    }
```

```
    return 0;
```

```
}
```

(Continuação)

# Sinais

- Um sinal é uma **interrupção por software** enviada aos processos pelo sistema para informá-los da ocorrência de eventos "anormais" dentro do ambiente de execução
  - Ex: falha de segmentação, violação de memória, erros de entrada e saída, etc.
- É uma forma de comunicação assíncrona
- Este é outro mecanismo que possibilita a comunicação e manipulação de processos

# Sinais

Nome	Significado
<b>SIGHUP</b>	Sinal emitido aos processos associados a um terminal quando o usuário perde conexão com a máquina (p.ex., em uma conexão remota)
<b>SIGINT</b>	Emitido aos processos do terminal quando as teclas de interrupção (CTRL+C) do teclado são acionadas.
<b>SIGQUIT</b>	Emitido aos processos do terminal quando as teclas de abandonos (CTRL+D) do teclado são acionadas.
<b>SIGILL</b>	Emitido quando uma instrução ilegal é detectada.
<b>SIGIOT</b>	Emitido em caso de problemas de hardware (entrada e saída digital).

# Sinais

Nome	Significado
<b>SIGFPE</b>	Erro de cálculo em ponto flutuante, assim como no caso de um número em ponto flutuante em formato ilegal. Indica sempre um erro de programação.
<b>SIGKILL</b>	Destruição: “arma absoluta” para matar os processos. Não pode ser ignorada, tampouco interceptada. Existe ainda o SIGTERM para uma morte mais “suave” para processos
<b>SIGTERM</b>	Versão mais “suave” do SIGKILL
<b>SIGSEGV</b>	Falha de segmentação

# Sinais

Nome	Significado
<b>SIGALRM</b>	Relógio: emitido quando o relógio de um processo termina sua contagem. O relógio é iniciado pela função <code>alarm()</code>
<b>SIGTERM</b>	Emitido quando o processo termina de maneira normal
<b>SIGUSR1</b>	Primeiro sinal disponível ao usuário, utilizado para a comunicação entre processos
<b>SIGUSR2</b>	Segundo sinal disponível ao usuário, utilizado para a comunicação entre processos
<b>SIGPWR</b>	Reativação sobre pane elétrica

# Sinais

- Um sinal (com exceção do SIGKILL) pode ser tratado de três maneiras distintas em UNIX:
  1. **Ignorado** (por exemplo, um processo em *background* pode ignorar as interrupções de teclado)
  2. **Execução-padrão:** de acordo com o sinal recebido, o processo aplica o comportamento típico para este sinal (por exemplo, matar o processo por falha de segmentação). A maioria dos sinais das tabelas anteriores termina o processo que o recebe.
  3. **Interceptado:** na recepção do sinal, o processo pára o que está fazendo para executar uma função pré-definida, e depois retoma a execução no ponto de onde foi interrompido. Os sinais SIGUSR1 e SIGUSR2 não possuem ação pré-definida, sendo usados para enviar interrupções entre processos de usuário.



```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

void funcao_para_control_c()
{
    printf("\nQuem mandou voce pressionar
    CTRL-C?\n");
    printf("Vou ter de fechar o
    programa!\n");
    exit(1);
}

int main()
{
    signal(SIGINT, funcao_para_control_c);
    printf("Pressione CTRL-C para retirar
    o programa do loop infinito
    abaixo.\n");
    while(1);
    return 0;
}
```

Code/07\_Pipes\_Sinais\_Alarmes/Ex4.c

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

void funcao_para_control_c()
{
    printf("\nQuem mandou voce pressionar
    CTRL-C?\n");
    printf("Vou ter de fechar o
    programa!\n");
    exit(1);
}

int main()
{
    signal(SIGINT, funcao_para_control_c);
    printf("Pressione CTRL-C para retirar
    o programa do loop infinito
    abaixo.\n");
    while(1);
    return 0;
}
```

**Se o processo atual receber o  
sinal SIGINT (CTRL+C  
pressionados), então  
funcao\_para\_control\_c()  
deve ser executada ao invés de  
matar o processo**

Code/07\_Pipes\_Sinais\_Alarmes/Ex4.c

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

void funcao_para_control_c()
{
    printf("\nQuem mandou voce pressionar
    CTRL-C?\n");
    printf("Vou ter de fechar o
    programa!\n");
    exit(1);
}

int main()
{
    signal(SIGINT, funcao_para_control_c);
    printf("Pressione CTRL-C para retirar
    o programa do loop infinito
    abaixo.\n");
    while(1);
    return 0;
}
```

Loop infinito para aguardar o usuário pressionar CTRL+C

Code/07\_Pipes\_Sinais\_Alarmes/Ex4.c

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

void funcao_para_control_c()
{
    printf("\nQuem mandou voce pressionar
CTRL-C?\n");
    printf("Vou ter de fechar o
programa!\n");
    exit(1);
}

int main()
{
    signal(SIGINT, funcao_para_control_c);
    printf("Pressione CTRL-C para retirar
o programa do loop infinito
abaixo.\n");
    while(1);
    return 0;
}
```

**Se o usuário pressionar CTRL+C, a função escreve uma mensagem na tela e mata o processo**

Code/07\_Pipes\_Sinais\_Alarmes/Ex4.c

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

void funcao_sigsegu()
{
    printf("Recebi segment fault. Vou morrer!!!\n");
    exit(1);
}

int main()
{
    char *p;
    signal(SIGSEGV, funcao_sigsegu);
    printf("Vou forcar um segment fault.\n");
    printf("%s", *p);
}
```

Code/07\_Pipes\_Sinais\_Alarmes/Ex5.c

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

void funcao_sigsegv()
{
    printf("Recebi segment fault. Vou morrer!!!\n");
    exit(1);
}

int main()
{
    char *p;
    signal(SIGSEGV, funcao_sigsegv);
    printf("Vou forçar um segment fault.\n");
    printf("%s", *p);
}
```

Se o processo atual receber o sinal SIGSEGV (falha de segmentação), então `funcao_sigsegv()` deve ser executada ao invés de matar o processo

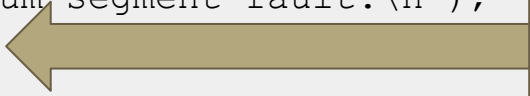
Code/07\_Pipes\_Sinais\_Alarmes/Ex5.c

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

void funcao_sigsegv()
{
    printf("Recebi segment fault. Vou morrer!!!\n");
    exit(1);
}

int main()
{
    char *p;
    signal(SIGSEGV, funcao_sigsegv);
    printf("Vou forçar um segment fault.\n");
    printf("%s", *p);
}
```

**Falha de segmentação (acesso a endereço não-inicializado)**



Code/07\_Pipes\_Sinais\_Alarmes/Ex5.c

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

void funcao_sigsegv()
{
    printf("Recebi segment fault. Vou morrer!!!\n");
    exit(1);
}

int main()
{
    char *p;
    signal(SIGSEGV, funcao_sigsegv);
    printf("Vou forçar um segment fault.\n");
    printf("%s", *p);
}
```

A função escreve  
uma mensagem na  
tela e mata o  
processo

Code/07\_Pipes\_Sinais\_Alarmes/Ex5.c



```

#include <errno.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

void tratamento_SIGUSR1()
{
    printf("Processo %d recebeu sinal
SIGUSR1... ele vai parar agora.\n",
getpid());
    exit(1);
}

int main()
{
    int pid_filho;
    signal(SIGUSR1, tratamento_SIGUSR1);
    printf("Processo pai [%d] vai
criar o filho e dormir por 1
segundo.\n", getpid());
    pid_filho = fork();
    if(pid_filho==0)
    {
        printf("Processo filho [%d] vai
entrar num loop infinito.\n", getpid());
        while(1);
    }
}

```

```

else
{
    sleep(1);
    printf("Processo %d vai enviar
o sinal SIGUSR1 para o processo %d\n",
getpid(), pid_filho);
    kill(pid_filho, SIGUSR1);
    printf("Processo %d vai dormir
por 1 segundo.\n", getpid());
    sleep(1);
}
printf("Processo %d
encerrando.\n", getpid());
exit(0);
}

```

(Continuação)

```

#include <errno.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

void tratamento_SIGUSR1()
{
    printf("Processo %d recebeu sinal
SIGUSR1... ele vai parar agora.\n",
getpid());
    exit(1);
}

int main()
{
    int pid_filho;
    signal(SIGUSR1, tratamento_SIGUSR1);
    printf("Processo pai [%d] vai
criar o filho e dormir por 1
segundo.\n", getpid());
    pid_filho = fork();
    if(pid_filho==0)
    {
        printf("Processo filho [%d] vai
entrar num loop infinito.\n", getpid());
        while(1);
    }
}

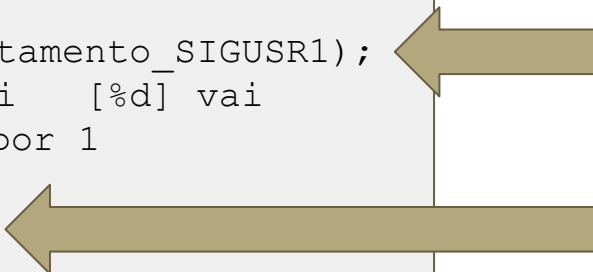
```

```

else
{
    sleep(1);
    printf("Processo %d vai enviar
o sinal SIGUSR1 para o processo %d\n",
getpid(), pid_filho);
    kill(pid_filho, SIGUSR1);
    printf("Processo %d vai dormir
por 1 segundo.\n", getpid());
    sleep(1);
}
printf("Processo %d
encerrando.\n", getpid());
exit(0);
}

```

**Vamos criar um  
processo-filho, e  
mandar o sinal  
SIGUSR1 para ele**



```

#include <errno.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

void tratamento_SIGUSR1()
{
    printf("Processo %d recebeu sinal
SIGUSR1... ele vai parar agora.\n",
getpid());
    exit(1);
}

int main()
{
    int pid_filho;
    signal(SIGUSR1, tratamento_SIGUSR1);
    printf("Processo pai [%d] vai
criar o filho e dormir por 1
segundo.\n", getpid());
    pid_filho = fork();
    if(pid_filho==0)
    {
        printf("Processo filho [%d] vai
entrar num loop infinito.\n", getpid());
        while(1);
    }
}

```

```

else
{
    sleep(1);
    printf("Processo %d vai enviar
o sinal SIGUSR1 para o processo %d\n",
getpid(), pid_filho);
    kill(pid_filho, SIGUSR1);
    printf("Processo %d vai dormir
por 1 segundo.\n", getpid());
    sleep(1);
}
printf("Processo %d
encerrando.\n", getpid());
exit(0);
}

```

(Continuação)

**O processo-filho  
escreve seu PID na  
tela e entra em um  
loop infinito**

## O processo-pai

- espera 1 segundo...

```
#include  
#include  
#include  
#include  
#include
```

```
void t  
{  
    pr  
    SIGUSR  
    getpid  
    ex  
}
```

```
int main()  
{  
    int pid_filho;  
    signal(SIGUSR1, tratamento_SIGUSR1);  
    printf("Processo pai [%d] vai  
criar o filho e dormir por 1  
segundo.\n", getpid());  
    pid_filho = fork();  
    if(pid_filho==0)  
    {  
        printf("Processo filho [%d] vai  
entrar num loop infinito.\n", getpid());  
        while(1);  
    }  
}
```

```
else  
{  
    sleep(1);  
    printf("Processo %d vai enviar  
o sinal SIGUSR1 para o processo %d\n",  
getpid(), pid_filho);  
    kill(pid_filho, SIGUSR1);  
    printf("Processo %d vai dormir  
por 1 segundo.\n", getpid());  
    sleep(1);  
}  
    printf("Processo %d  
encerrando.\n", getpid());  
    exit(0);  
}
```

(Continuação)

## O processo-pai

- espera 1 segundo...
- escreve seu PID na tela...

```
#include  
#include  
#include  
#include  
#include
```

```
void t  
{  
    pr  
    SIGUSR  
    getpid  
    ex  
}
```

```
int main()  
{  
    int pid_filho;  
    signal(SIGUSR1, tratamento_SIGUSR1);  
    printf("Processo pai [%d] vai  
criar o filho e dormir por 1  
segundo.\n", getpid());  
    pid_filho = fork();  
    if(pid_filho==0)  
    {  
        printf("Processo filho [%d] vai  
entrar num loop infinito.\n", getpid());  
        while(1);  
    }  
}
```

```
else  
{  
    sleep(1);  
    printf("Processo %d vai enviar  
o sinal SIGUSR1 para o processo %d\n",  
getpid(), pid_filho);  
    kill(pid_filho, SIGUSR1);  
    printf("Processo %d vai dormir  
por 1 segundo.\n", getpid());  
    sleep(1);  
}  
printf("Processo %d  
encerrando.\n", getpid());  
exit(0);  
}
```

(Continuação)

## O processo-pai

- espera 1 segundo...
- escreve seu PID na tela...
- envia o sinal SIGUSR1 para o processo-filho...

```
#inclu  
#inclu  
#inclu  
#inclu  
#inclu
```

```
void t  
{  
    pr  
    SIGUSR  
    getpid  
    ex  
}
```

```
int main()  
{  
    int pid_filho;  
    signal(SIGUSR1, tratamento_SIGUSR1);  
    printf("Processo pai [%d] vai  
criar o filho e dormir por 1  
segundo.\n", getpid());  
    pid_filho = fork();  
    if(pid_filho==0)  
    {  
        printf("Processo filho [%d] vai  
entrar num loop infinito.\n", getpid());  
        while(1);  
    }  
}
```

```
else  
{  
    sleep(1);  
    printf("Processo %d vai enviar  
o sinal SIGUSR1 para o processo %d\n",  
getpid(), pid_filho);  
    kill(pid_filho, SIGUSR1);  
    printf("Processo %d vai dormir  
por 1 segundo.\n", getpid());  
    sleep(1);  
}  
    printf("Processo %d  
encerrando.\n", getpid());  
    exit(0);  
}
```

(Continuação)

## O processo-pai

- espera 1 segundo...
- escreve seu PID na tela...
- envia o sinal SIGUSR1 para o processo-filho...
- espera 1 segundo...

```
#include  
#include  
#include  
#include  
#include
```

```
void t  
{  
    pr  
    SIGUSR  
    getpid  
    ex  
}
```

```
int main()  
{  
    int pid_filho;  
    signal(SIGUSR1, tratamento_SIGUSR1);  
    printf("Processo pai [%d] vai  
criar o filho e dormir por 1  
segundo.\n", getpid());  
    pid_filho = fork();  
    if(pid_filho==0)  
    {  
        printf("Processo filho [%d] vai  
entrar num loop infinito.\n", getpid());  
        while(1);  
    }  
}
```

```
else  
{  
    sleep(1);  
    printf("Processo %d vai enviar  
o sinal SIGUSR1 para o processo %d\n",  
getpid(), pid_filho);  
    kill(pid_filho, SIGUSR1);  
    printf("Processo %d vai dormir  
por 1 segundo.\n", getpid());  
    sleep(1);  
}  
    printf("Processo %d  
encerrando.\n", getpid());  
    exit(0);  
}
```

(Continuação)

## O processo-pai

- espera 1 segundo...
- escreve seu PID na tela...
- envia o sinal SIGUSR1 para o processo-filho...
- espera 1 segundo...
- e termina sua execução

```
#include  
#include  
#include  
#include  
#include
```

```
void t  
{  
    pr  
    SIGUSR  
    getpid  
    ex  
}
```

```
int main()  
{  
    int pid_filho;  
    signal(SIGUSR1, tratamento_SIGUSR1);  
    printf("Processo pai [%d] vai  
criar o filho e dormir por 1  
segundo.\n", getpid());  
    pid_filho = fork();  
    if(pid_filho==0)  
    {  
        printf("Processo filho [%d] vai  
entrar num loop infinito.\n", getpid());  
        while(1);  
    }  
}
```

```
else  
{  
    sleep(1);  
    printf("Processo %d vai enviar  
o sinal SIGUSR1 para o processo %d\n",  
getpid(), pid_filho);  
    kill(pid_filho, SIGUSR1);  
    printf("Processo %d vai dormir  
por 1 segundo.\n", getpid());  
    sleep(1);  
}  
    printf("Processo %d  
encerrando.\n", getpid());  
    exit(0);  
}
```

(Continuação)



```

#include <errno.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

void tratamento_SIGUSR1()
{
    printf("Processo %d recebeu sinal
SIGUSR1... ele vai parar agora.\n",
getpid());
    exit(1);
}

int main()
{
    int pid_filho;
    signal(SIGUSR1, tratamento_SIGUSR1);
    printf("Processo pai [%d] vai
criar o filho e dormir por 1
segundo.\n", getpid());
    pid_filho = fork();
    if(pid_filho==0)
    {
        printf("Processo filho [%d] vai
entrar num loop infinito.\n", getpid());
        while(1);
    }
}

```

```

else
{
    sleep(1);
    printf("Processo %d vai enviar
o sinal SIGUSR1 para o processo %d\n",
getpid(), pid_filho);
    // O processo pai vai dormir por 1 segundo
    sleep(1);
}
}

```

**O processo que receber o sinal SIGUSR1 vai escrever seu PID na tela, e depois terminar sua execução**

(continuação)

```
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

void tratamento_alarme(int sig)
{
    alarm(1);
    system("date +%H:%M:%S.%N");
}

int main()
{
    signal(SIGALRM, tratamento_alarme);
    alarm(1);
    printf("Aperte CTRL+C para acabar:\n");
    while(1);
    return 0;
}
```

Code/07\_Pipes\_Sinais\_Alarmes/Ex7.c

```
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

void tratamento_alarme(int sig)
{
    alarm(1);
    system("date +%H:%M:%S.%N");
}

int main()
{
    signal(SIGALRM, tratamento_alarme);
    alarm(1);
    printf("Aperte CTRL+C para acabar:\n");
    while(1);
    return 0;
}
```

O sinal SIGALRM é enviado  
ao processo quando um timer  
termina a contagem

Neste código, a função  
`tratamento_alarme()`  
será chamada ao final da  
contagem

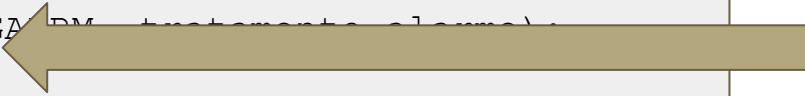
Code/07\_Pipes\_Sinais\_Alarmes/Ex7.c

```
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

void tratamento_alarme(int sig)
{
    alarm(1);
    system("date +%H:%M:%S.%N");
}

int main()
{
    signal(SIGALRM, tratamento_alarme);
    alarm(1);
    printf("Aperte CTRL+C para acabar:\n");
    while(1);
    return 0;
}
```

**A função  
alarm(1) inicia a  
contagem de 1  
segundo do timer**



Code/07\_Pipes\_Sinais\_Alarmes/Ex7.c

```
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

void tratamento_alarme(int sig)
{
    alarm(1);
    system("date +%H:%M:%S.%N");
}

int main()
{
    signal(SIGALRM, tratamento_alarme);
    alarm(1);
    printf("Aperte CTRL+C para acabar:\n");
    while(1);
    return 0;
}
```

**A função**  
`tratamento_alarme()`  
**reinicia o alarme e escreve**  
**a hora atual, os minutos,**  
**segundos e nanosegundos**

Code/07\_Pipes\_Sinais\_Alarmes/Ex7.c

```
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

void tratamento_alarme(int sig)
{
    alarm(1);
    system("date +%H:%M:%S.%N");
}

int main()
{
    signal(SIGALRM, tratamento_alarme);
    alarm(1);
    printf("Aperte CTRL+C para terminar a execucao\n");
    while(1);
    return 0;
}
```

Como o código principal fica em loop infinito, é preciso pressionar CTRL+C para terminar sua execução

Code/07\_Pipes\_Signals\_Alarmes/EX7.c

```

#include <signal.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

void tratamento_alarme(int sig)
{
    alarm(1);
    system("date +%F");
}

int main()
{
    signal(SIGALRM, tratamento_alarme);
    alarm(1);
    printf("Aperte qualquer tecla para disparar o alarme.\n");
    while(1);
    return 0;
}

```

**Observação:** A função `sleep()` chama a função `alarm()`. Deve-se então utilizá-la com maior prudência se o programa já manipula o sinal `SIGALRM`.

Code/07\_Pipes\_Sinais\_Alarmes/Ex7.c