

# **CURSO DE EXTENSÃO EM PROGRAMAÇÃO LINUX E DESENVOLVIMENTO DE SISTEMAS EMBARCADOS**

**Programação em C no Linux:  
Recursos do Sistema**

**SLACKWARE**

THOSE PENGUINS... THEY SURE 'AINT NORMAL...



**Edson Mintsu Hung**  
**[mintsu@image.unb.br](mailto:mintsu@image.unb.br)**

# Programação em C no Linux: recursos do sistema

- Coordenação:
  - Prof. Francisco Assis Oliveira Nascimento
  - Prof. Geovany Araújo Borges
- Instrutor:
  - Edson Mintsu Hung  
**`mintsu@image.unb.br`**

SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...



LINUX

# Programação em C no Linux: recursos do sistema

- Pré-requisitos:
  - Conhecimentos das linguagens C e C++;
  - Paciência e disposição para aprender.

# Programação em C no Linux: recursos do sistema

## MÓDULO 2

fedora

debian

SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...



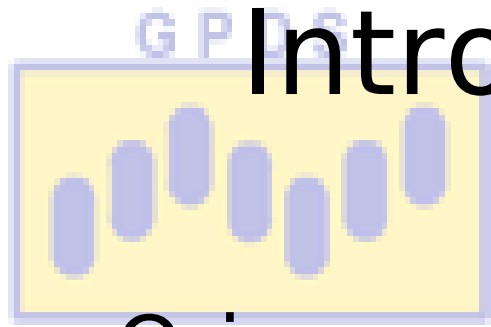
LINUX

# Introdução ao desenvolvimento com Linux

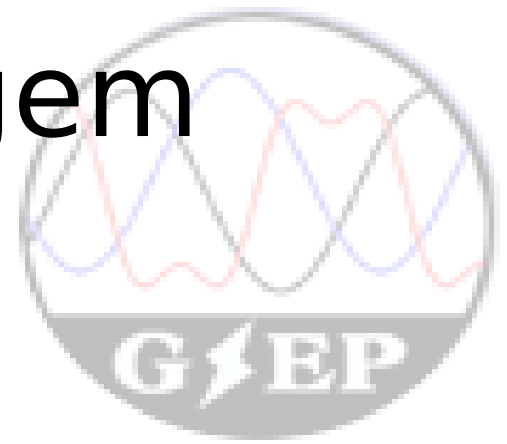
- Conteúdo:

- Breve introdução da Linguagem C ANSI;
- *Makefiles*;
- Acesso a arquivos;
- Processos e Sinais;
- Threads POSIX;
- Comunicação e sincronismo entre processos;
- Programação em soquetes;
- Device drivers.





# Introdução à Linguagem ANSI\*-C



- Origem:

- O C é uma linguagem de programação genérica inventada na década de 70 por Dennis Ritchie. O C é derivado de uma outra linguagem: o B, criado por Ken Thompson. O B, por sua vez, veio da linguagem BCPL, inventada por Martin Richards.

- Por que utilizar C?

- **Versatilidade:** ele possui tanto características de "alto nível" quanto de "baixo nível".
- **“Poder”:** o C possui ampla biblioteca de funções e é utilizado na desenvolvimento de softwares para os mais diversos projetos.

\* ANSI: Insituto Norte Americano de Padronização

THOSE PENGUINS... THEY SURE AINT NORMAL...

LINUX

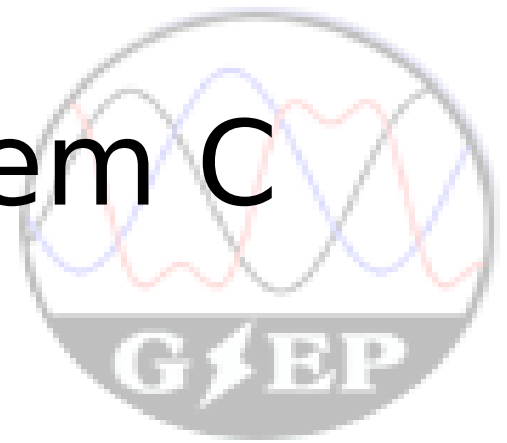
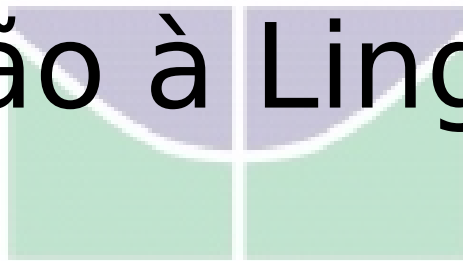


# Introdução à Linguagem C

- Um programa em C consiste, no fundo, de várias funções colocadas juntas num arquivo de extensão **.c**.
- Uma função é um bloco de código de programa que pode ser usado diversas vezes em sua execução. O uso de funções permite que o programa fique mais legível, mais estruturado, daí o nome de programação estruturada.
- A extensão **.h** vem de **header** não possuem os códigos completos das funções. Eles só contêm **protótipos** de funções.



# Introdução à Linguagem C



- **Forma geral**

```
tipo_de_retorno nome_da_função(lista_de_argumentos) {  
    código_da_função ;  
  
    return ( valor_de_retorno );  
}
```

- Chamada à função: toda função de ser chamada da seguinte forma:

```
nome_da_função(lista_de_argumentos);
```

- As funções do tipo **void** não necessitam da instrução **return** acima.

SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...



LINUX



# Introdução à Linguagem C

- **Em linguagem C:**

```
/* Isso é um comentário */  
  
void main()  
{  
    int x,y; /* corpo da função */  
    x= x+y;  
}
```

# Introdução à Linguagem C

- A função `main()`

- Todo programa deve ter uma função `main()` e ela deve ser única.
- A função `main()` é o ponto de partida quando o programa é executado.
- Arquivos auxiliares não devem conter a função `main()`.
- Sintaxe usual:

```
main(int argc, char argv[]){  
    // bloco de código  
}
```



# Introdução à Linguagem C

- As variáveis no C devem ser declaradas antes de serem usadas, no início de um bloco de código. A forma geral da declaração de variáveis é:

`tipo_da_variável lista_de_variáveis;`

- Exemplo:

```
char ch, letra;
```

```
int h = 0 ; /* inicialização */
```

SLACKWARE

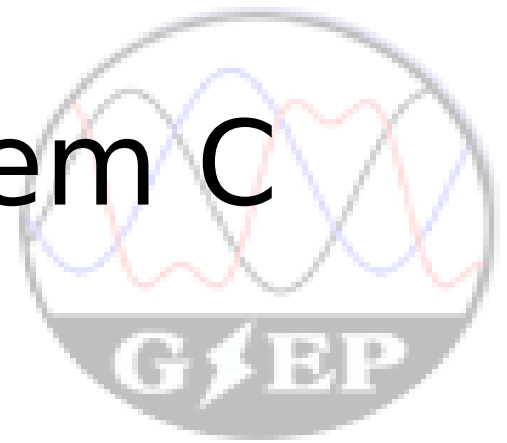
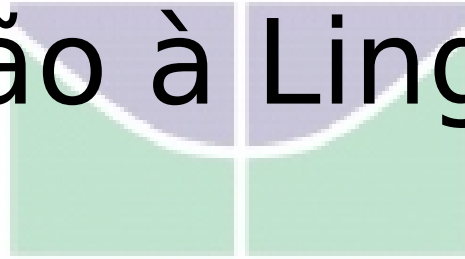
THOSE PENGUINS... THEY SURE 'AINT NORMAL...



LINUX



# Introdução à Linguagem C



- Modificadores:

- Para cada um dos tipos de variáveis existem os modificadores de tipo. Os modificadores de tipo do C são quatro: **signed**, **unsigned**, **long** e **short**.

`modificador tipo_da_variável lista_de_variáveis;`

SLACKWARE

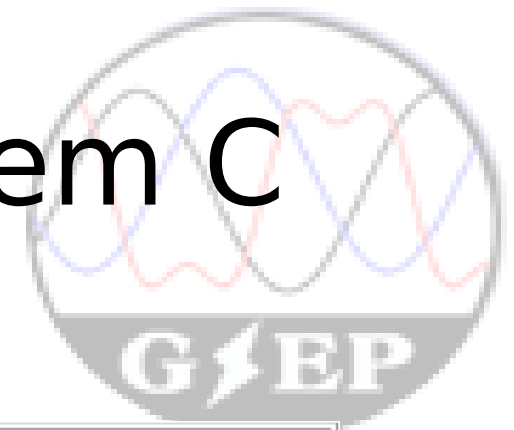
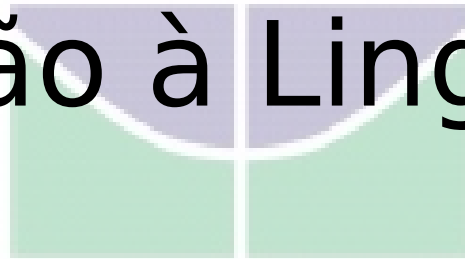
THOSE PENGUINS... THEY SURE 'AINT NORMAL...



LINUX



# Introdução à Linguagem C



- Tipos de variáveis:

Tipo	Num de bits	Formato para leitura com scanf	Intervalo	
			Início	Fim
char	8	%c	-128	127
unsigned char	8	%c	0	255
signed char	8	%c	-128	127
int	16	%i	-32.768	32.767
unsigned int	16	%u	0	65.535
signed int	16	%i	-32.768	32.767
short int	16	%hi	-32.768	32.767
unsigned short int	16	%hu	0	65.535
signed short int	16	%hi	-32.768	32.767
long int	32	%li	-2.147.483.648	2.147.483.647
signed long int	32	%li	-2.147.483.648	2.147.483.647
unsigned long int	32	%lu	0	4.294.967.295
float	32	%f	3,4E-38	3.4E+38
double	64	%lf	1,7E-308	1,7E+308
long double	80	%Lf	3,4E-4932	3,4E+4932

Exemplo:

```
char ch, pix;
```

```
int h = 0 ; /* inicialização */
```



# Introdução à Linguagem C

- Variáveis Globais e Locais:

Variáveis Globais	Variáveis Locais
Alocadas na memória, portanto existe fisicamente e ocupa espaço na área de dados	Alocadas na pilha, portanto temporária, o que reduz a área de dados
Declaradas fora das funções, pode ser usada em qualquer ponto do programa	Declaradas dentro de uma função, é de uso exclusivo dela

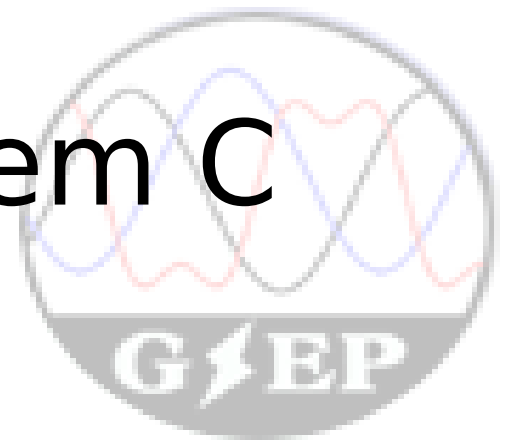
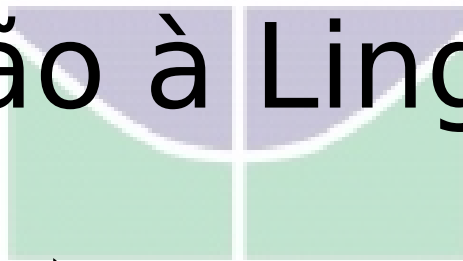
SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...





# Introdução à Linguagem C



- Modelador (*cast*):
  - Um modelador é aplicado a uma expressão. Ele *força* (*cast* = *coerção*) a mesma a ser de um tipo especificado. Sua forma geral é:

( *tipo* ) expressão

SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...



LINUX

# Introdução à Linguagem C

- Operadores Aritméticos e Atribuição:

Operador	Ação	Exemplo
=	Atribuição	<code>x = 0x1000 ;</code>
+	Adição	<code>x = 0x18 + y ;</code>
-	Subtração	<code>x = x - y ;</code>
*	Multiplicação (inteira)	<code>x = 8*y ;</code>
/	Divisão (inteira)	<code>x = y / 2 ;</code>
%	Resto de divisão	<code>x = y % 2;</code>
++	Incremento	<code>x++ ;</code>
--	Decremento	<code>y-- ;</code>





# Introdução à Linguagem C

- Operadores Relacionais:

Operador Relacional	Ação
>	Maior do que
>=	Maior ou igual a
<	Menor do que
<=	Menor ou igual a
==	Igual a
!=	Diferente de

SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...





# Introdução à Linguagem C

- Operadores Lógicos:

Operador Lógico	Ação
&&	AND (E)
	OR (OU)
!	NOT (NÃO)

SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...





# Introdução à Linguagem C

- Operadores Lógicos bit a bit:

Operador Lógico	Ação
&	AND
	OR
^	XOR (OR exclusivo)
~	NOT
>>	Deslocamento de bits a direita
<<	Deslocamento de bits a esquerda

SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...





# Introdução à Linguagem C

- Estruturas de controle de fluxo:

As estruturas de controle de fluxo são fundamentais para qualquer linguagem de programação. A partir delas é possível controlar a ordem com que as instruções são executadas, bastando especificar condições de execução. Vamos nos ater à sintaxe das estruturas mais utilizadas: **if**, **for**, **while** e **do**.

SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...



# Introdução à Linguagem C

- Comando **if**:

```
if (condição) {  
    comandos;  
}
```

- Comando **if-else**:

```
if (condição) {  
    comandos para condição verdadeira;  
}  
else{  
    comandos para condição falsa;  
}
```

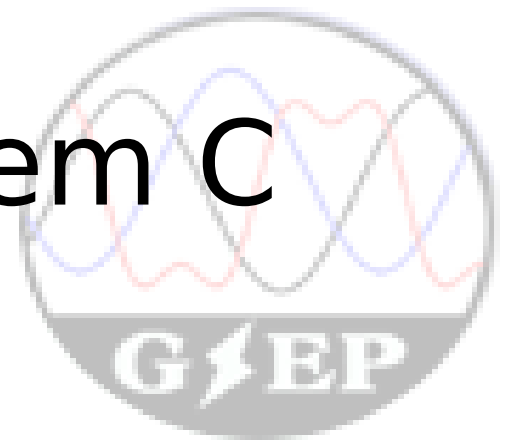
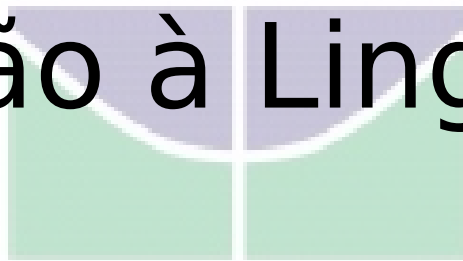
# Introdução à Linguagem C

- Comando **switch-case**:

```
switch (variável) {  
    case constante_A:  
        comandos para variável = constante_A;  
        break;  
    case constante_B:  
        comandos para variável = constante_B;  
        break;  
    default:  
        comandos para condição default;  
}
```



# Introdução à Linguagem C



- Comando **for**:

O laço **for** é usado para repetir um comando, ou bloco de comandos, diversas vezes, desde que uma condição seja satisfeita. Sua forma geral é:

```
for (inicialização; condição; operação) {  
    /* bloco de código */  
}
```

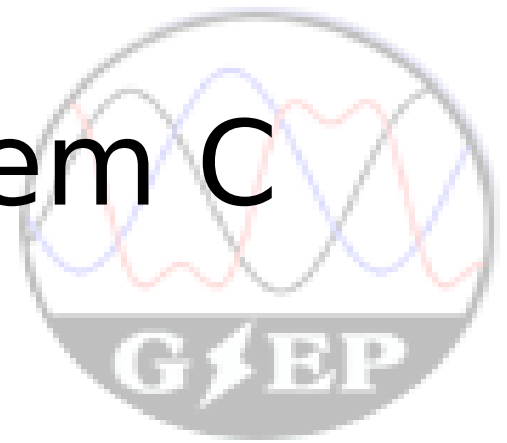
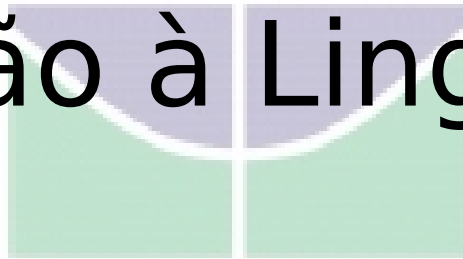
SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...





# Introdução à Linguagem C



- Comando **while**:

A estrutura de repetição **while**, assim como a **for**, faz a repetição de um bloco de código, a partir da avaliação de um condição, se esta for verdadeira o bloco é executado e faz-se o teste novamente, se esta nova avaliação for verdadeira, a execução ocorre novamente. Sua forma geral é:

```
while (condição) {  
    /* bloco de código */  
    expressões;  
}
```

SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...



LINUX



# Introdução à Linguagem C

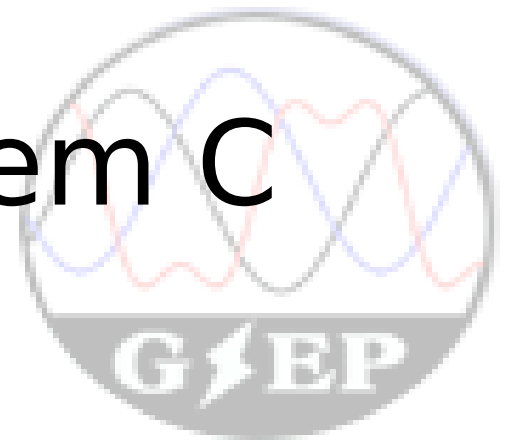
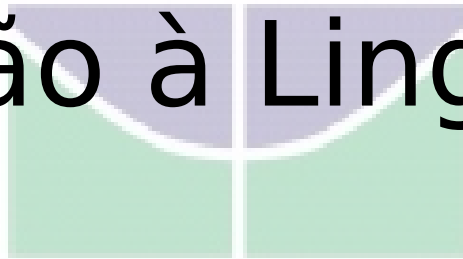
- Comando **do-while**:

A estrutura do-while também repete um conjunto de instruções a partir de uma condição verdadeira. A grande novidade no comando **do-while** é que ele, ao contrário do **for** e do **while**, garante que as instruções serão executadas pelo menos uma vez. Sua forma geral é:

```
do {  
    /* bloco de código */  
    expressões;  
} while (condição)
```



# Introdução à Linguagem C



- Vetores e matrizes:

Para se declarar um vetor podemos utilizar a seguinte forma geral:

```
tipo_da_variavel nome_da_variável [tamanho]
```

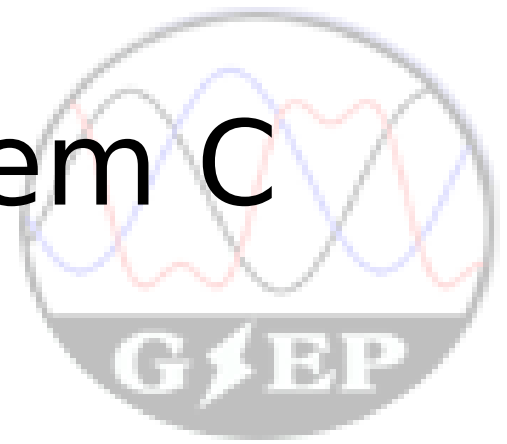
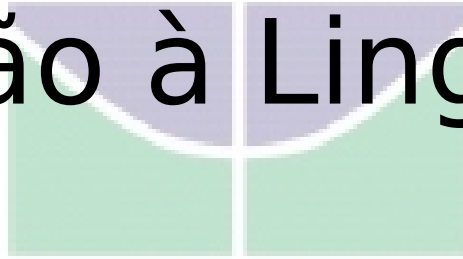
No caso da matriz:

```
tipo_var nome_var [tam_1][tam_2] ... [tam_N]
```

- \* Note que os elementos não permitem tipos distintos.
- \* Quando se declara no C variáveis como estas um espaço na memória, suficientemente grande para armazenar o número de células especificadas em tamanho, é reservado.



# Introdução à Linguagem C



- Strings:

Pode-se declarar string em vetores facilmente.

```
senha[ ] = "6pD5l1nux";
```

Caso se determine o número de posições do vetor deve-se considerar o terminado nulo.

```
sigla[5] = "GPDS"
```

Na memória estará gravada a sequência:

```
'G, 'P, 'D, 'S, 0x00;
```

SLACKWARE

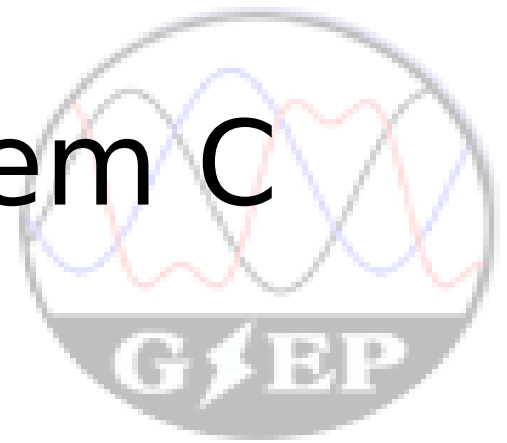
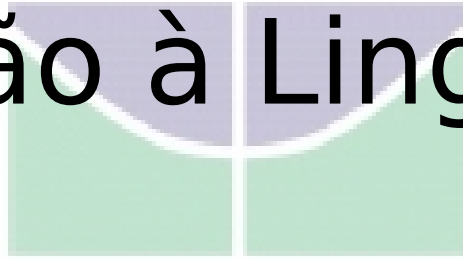
THOSE PENGUINS... THEY SURE 'AINT NORMAL...



LINUX



# Introdução à Linguagem C



- Ponteiros:

São variáveis que contém um endereço de memória.

- Pode ser utilizado para apontar variáveis.
- Pode também ser utilizado para apontar para funções.

Sintaxe:

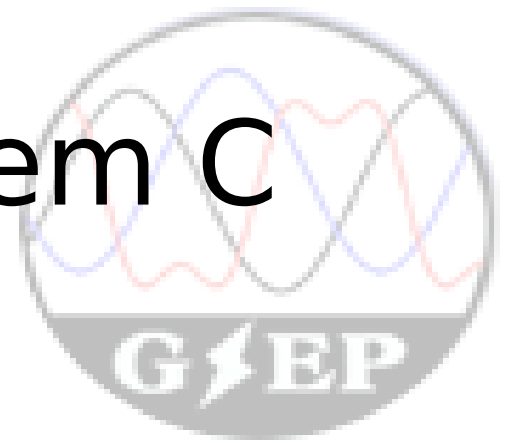
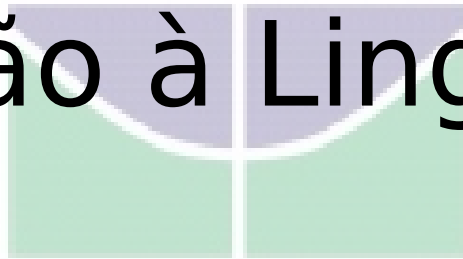
```
tipo_da_variável *nome_da_variável;
```

SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...



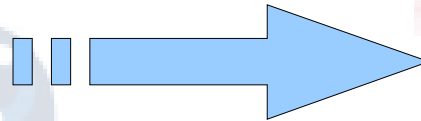
LINUX



# Introdução à Linguagem C

- Exemplo de ponteiro:

```
char c[ ] = "teste", *pchar = NULL;  
pchar = &c[0]; // ou pchar = c;
```



```
*(pchar+2)++; // c[2] = 't'  
++pchar; // pchar = 131h
```

Variável	Endereço	Conteúdo
c	130h	t'
	131h	e'
	132h	s'
	133h	t'
	134h	e'
	135h	\0'
pchar	20h	130h

SLACKWARE

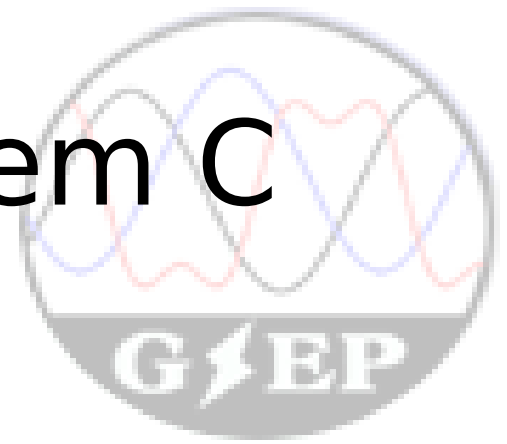
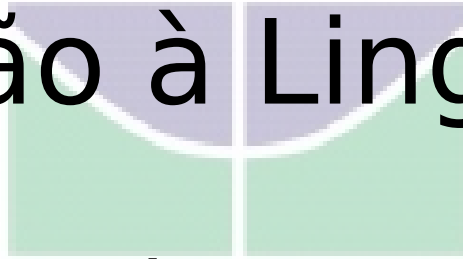
THOSE PENGUINS... THEY SURE 'AINT NORMAL...



LINUX

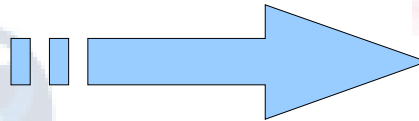


# Introdução à Linguagem C



- Exemplo de ponteiro:

```
int x[] = {10,34,40}, *pint = NULL;  
pint = &x[0]; // ou pint = x;
```



Variável	Endereço	Conteúdo
x	25Eh	10
	260h	24
	262h	40
pint	65h	25Eh

```
*(pint+=2) = 5; // x[2] = 5, pint = 262h  
++pint; // pint = 264h (cuidado aqui!)
```

SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...



LINUX

# Introdução à Linguagem C

```
void main (void)
{
```

```
    double a, b, c = 2.0, d = -1.0;
```

```
    if(!cartesian_to_polar(&a, &b, &c, &d)){
```

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

```
    }
```

```
}
```

```
unsigned char cartesian_to_polar(double *prho, double
    *palpha, double *px, double *py)
```

```
{
```

```
    if(*px != 0) *palpha = atan((*py)/(*px));
```

```
    else return 0;
```

```
    *prho = sqrt((*px)*(*px) + (*py)*(*py));
```

```
    return 1;
```

```
}
```

# Introdução à Linguagem C

- Exemplo de ponteiros para funções:

```
double normal(double a, double b){ return(fabs(a)+fabs(b)); }
```

```
double norma2(double a, double b){ return(sqrt(a*a+b*b)); }
```

```
void main (void)
```

```
{  
    double (*norma)(double a, double b);
```

```
    double x;
```

```
    norma = normal;
```

```
    x = norma(4.0,-3.0); // x = 7
```

```
    norma = norma2;
```

```
    x = norma(4.0,-3.0); // x = 5
```

```
}
```



# Introdução à Linguagem C

- Apontamentos para ponteiros:

```
int x = 10, y = 20;

void funcao(int **ppvar)
{
    *ppvar = &y; // coloca o ponteiro original apontando para y
    **ppvar = 25; // afeta y
}

void main (void)
{
    int *pint;
    pint = &x; // pint aponta para x, então *pint = 10.
    funcao(&pint); // pint aponta para y, então *pint = 25.
}
```

# Introdução à Linguagem C

- Estruturas:

- Agrupamento de variáveis/funções em uma única estrutura.

```
struct {  
    int dia;  
    int mes;  
    int ano;  
} data;  
void main (void)  
{  
    // entrando a data de 26/06/2007  
    data.dia = 26;  
    data.mes = 06;  
    data.ano = 2007;  
}
```

# Introdução à Linguagem C

- Alocação dinâmica:

- Em alguns casos não se conhece, com antecedência, o tamanho de uma variável. Sendo necessária a utilização da alocação dinâmica.

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

```
void main (void)
```

```
{
```

```
    int *pbuffer;
```

```
    // Aloca memória
```

```
    pbuffer = (int *) malloc(10 * sizeof(int));
```

```
    // Faz uso da memória alocada
```

```
    pbuffer[2] = 35;
```

```
    // Libera a memória alocada
```

```
    free(pbuffer);
```

```
}
```



# MakeFiles



- O que são *Makefiles*?
  - Makefiles são scripts que possuem as diretivas de compilação do programa, possuindo em seu bojo:
    - Estrutura do projeto (arquivos, dependências);
    - Instruções para criação de arquivos.

SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...

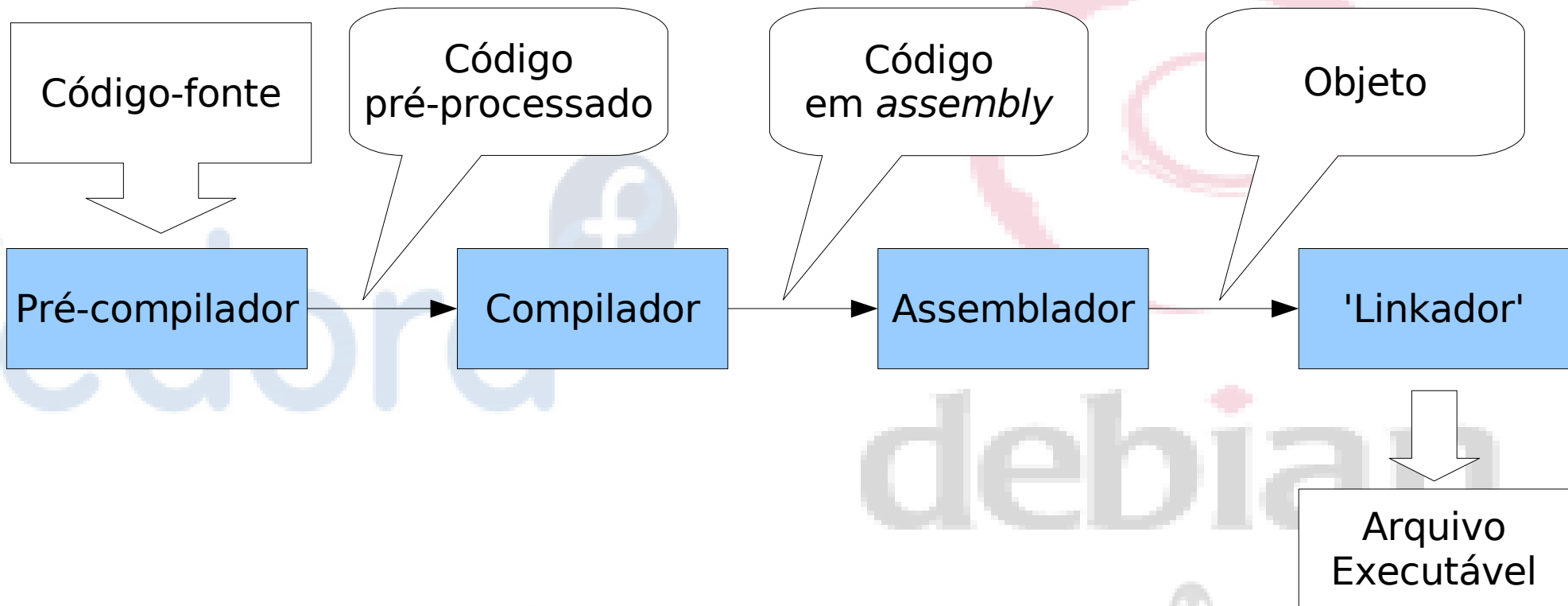


LINUX



# MakeFiles

- Fluxo de compilação:



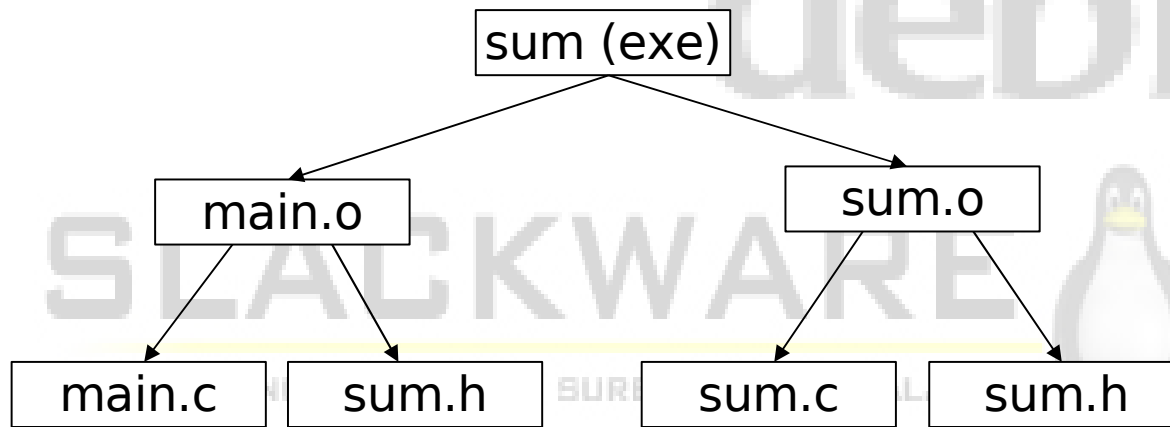
SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...



# MakeFiles

- As estruturas e dependências de um projeto podem ser representadas por um DAG (*Directed Acyclic Graph*).
- Exemplo :
  - Neste exemplo o programa possui três arquivos.
    - `main.c`, `sum.c`, `sum.h`
    - `sum.h` está incluído em ambos arquivos `.c`
    - O arquivo executável deve ser o `sum`





# MakeFiles



Alvo

```
sum.o: sum.c sum.h
```

Dependência

```
gcc -c sum.c
```

Ação

tab

fedora

debian

SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...



LINUX



# MakeFiles

- '*Clean*': Tem como objetivo excluir os objetos e/ou arquivos temporários ou desnecessários.

No Makefile:

```
clean:
```

```
rm -rf ./*.o
```

E para executá-lo basta digitar no shell:

```
$ make clean
```

SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...



LINUX





# MakeFiles

- O make file pode ser descrito como:

```
sum: main.o sum.o
```

```
gcc -o sum main.o sum.o
```

```
main.o: main.c sum.h
```

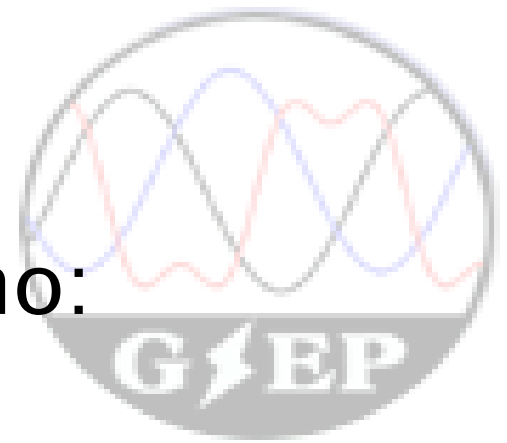
```
gcc -c main.c
```

```
sum.o: sum.c sum.h
```

```
gcc -c sum.c
```

```
clean:
```

```
rm -rf ./*.o
```



debian

SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...





# MakeFiles



- Macros:

- São palavras chaves ou mnemônicos que definem um conjunto de palavras:

`nome_da_macro=texto`

- E que depois de definidos são referenciados como:

`${nome_da_macro}`

`$(nome_da_macro)`





# MakeFiles

- Bom exemplo de macro, mas péssimo de makefile:

```
LIBS = -lX11
```

```
OBJS = draw.o plot_pts.o root_data.o
```

```
CC = gcc
```

```
BINDIR = /usr/local/bin
```

```
DEBUG_FLAG = # -g para habilitar o modo debug
```

```
CFLAGS = -Wall
```

```
plot: ${OBJS}
```

```
    ${CC} -o plot ${DEBUG_FLAG} ${CFLAGS} ${OBJS} \  
        ${LIBS}
```

```
mv plot ${BINDIR}
```

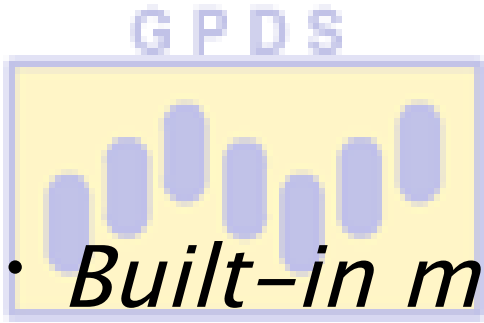


debian

SLACKWARE

mv plot \${BINDIR}





# MakeFiles



- *Built-in macros:* São as macros pré-definidas.

`$@` #nome do arquivo de saída.

`$?` #nome dos dependentes modificados.

exemplo: exemplo.c

```
$(CC) $(CFLAGS) $? $(LDFLAGS) -o $@
```

`$*` #o prefixo dos arquivos que estão como alvo e dependentes.

`$<` #"nome implícito".

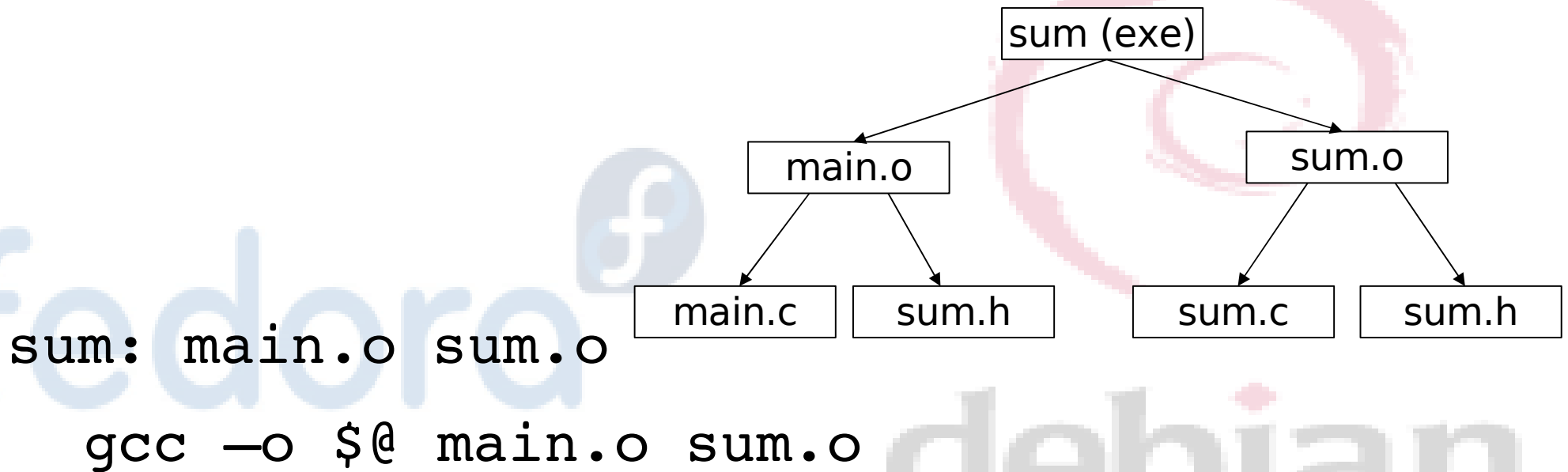
exemplo.o: exemplo.c

```
$(CC) $(CFLAGS) $*.c $(LDFLAGS) -c $<
```



# MakeFiles

- Podemos ainda reescrever o *makefile* do primeiro exemplo, como as mesmas dependências, utilizando uma notação reduzida na forma de *built-in macros*.



```
main.o sum.o: sum.h
```

```
gcc -c $*.c
```





# MakeFiles

- Por obséquio, não escreva seu makefile assim:

```
sum: main.c sum.c
```

```
gcc -o sum main.c sum.c
```

pois requer a (re)compilação total do projeto quando algum arquivo for mudado.

SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...





# MakeFiles

- Controle de fluxos simples também podem ser utilizados em makefiles.

– Sintaxe:

```
ifeq (valor_1, valor_2)  
    #script para valores iguais  
else  
    #script para valores diferentes  
endif
```



SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...



LINUX



# MakeFiles

- Exemplificando:

```
# Comment/uncomment the following line to\  
disable/enable debugging
```

```
#DEBUG = y
```

```
# Add your debugging flag (or not) to CFLAGS
```

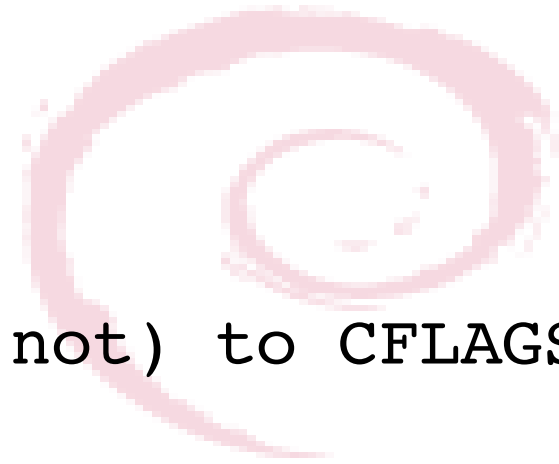
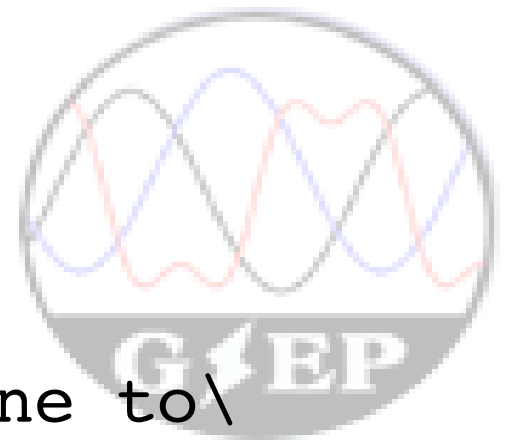
```
ifeq ($(DEBUG),y)
```

```
    DEBFLAGS = -O -g # "-O" is needed to expand\  
    inlines
```

```
else
```

```
    DEBFLAGS = -O2
```

```
endif
```



SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...





# MakeFiles

- **Exercício:** Escrever os arquivos `sum.c` (contendo a função soma) e `sum.h`, e escrever os makefiles expostos em aula, com a possibilidade de compilá-los com os makefiles utilizando ou não *built-in macros*.

```
# include "sum.h"
```

```
int main(int argc, char **argv)
```

```
{
```

```
    int a, b, c;
```

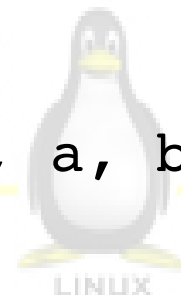
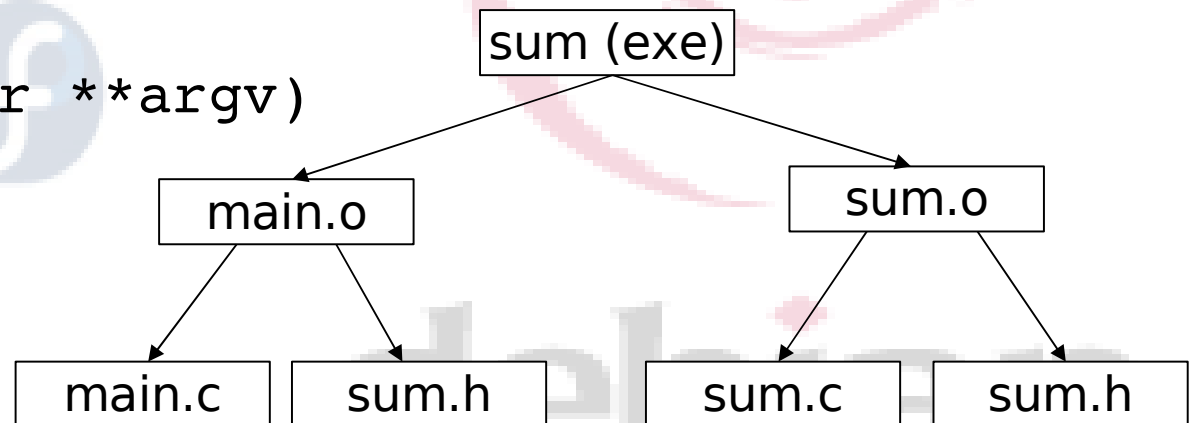
```
    a = atoi(argv[1]);
```

```
    b = atoi(argv[2]);
```

```
    c = soma(a,b);
```

```
    printf("\nA soma de %i com %i é %i \n", a, b, c);
```

```
}
```





# MakeFiles

- Referências Bibliográficas:



- GNU Make – O'Reilly

- <http://www.oreilly.com/catalog/make3/book/index.csp>

- GNU Make – [www.gnu.org](http://www.gnu.org)

- <http://www.gnu.org/software/make/manual/>

SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...



# Programação em C no Linux: recursos do sistema

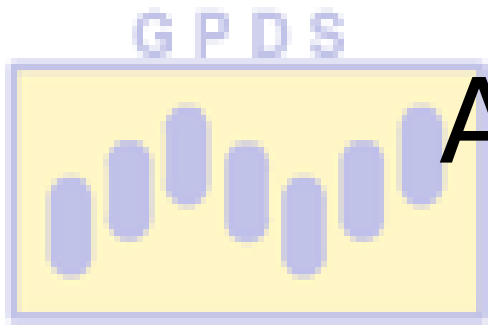
ACESSO A ARQUIVOS

SLACKWARE

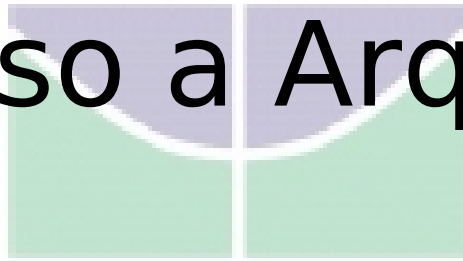
THOSE PENGUINS... THEY SURE 'AINT NORMAL...



LINUX



# Acesso a Arquivos



A importância de se estudar a estrutura de arquivos do UNIX se deve ao fato de praticamente todos os dispositivos e serviços do sistema têm interface de arquivos. Isso nos indica que necessitaremos, de maneira geral, de apenas cinco funções básicas: **abrir**, **fechar**, **ler**, **escrever** e **controle**.

SLACKWARE

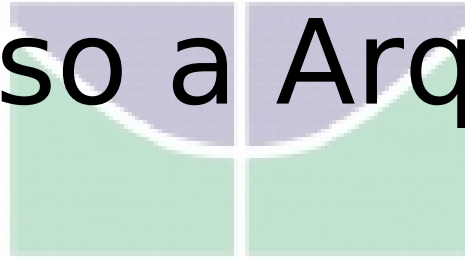
THOSE PENGUINS... THEY SURE 'AINT NORMAL...



LINUX



# Acesso a Arquivos



O sistema de entrada e saída do ANSI C é composto por uma série de funções, cujos protótipos estão reunidos em **stdio.h**. Todas estas funções trabalham com o conceito de "ponteiro de arquivo".

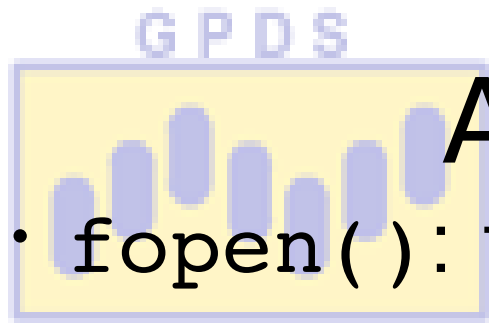
```
FILE *p;
```

SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...



LINUX

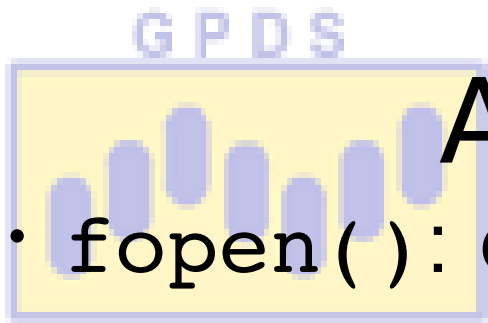


# Acesso a Arquivos

- `fopen()`: função de abertura de arquivos

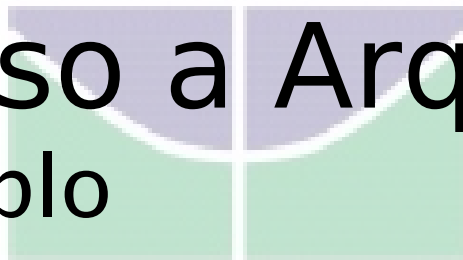
```
FILE *fopen (char *nome_do_arquivo, char *modo);
```

Modo	Significado
"r"	Abre um arquivo texto para leitura. O arquivo deve existir antes de ser aberto.
"w"	Abrir um arquivo texto para gravação. Se o arquivo não existir, ele será criado. Se já existir, o conteúdo anterior será destruído.
"a"	Abrir um arquivo texto para gravação. Os dados serão adicionados no fim do arquivo ("append"), se ele já existir, ou um novo arquivo será criado, no caso de arquivo não existente anteriormente.
"rb"	Abre um arquivo binário para leitura. Igual ao modo "r" anterior, só que o arquivo é binário.
"wb"	Cria um arquivo binário para escrita, como no modo "w" anterior, só que o arquivo é binário.
"ab"	Acrescenta dados binários no fim do arquivo, como no modo "a" anterior, só que o arquivo é binário.
"r+"	Abre um arquivo texto para leitura e gravação. O arquivo deve existir e pode ser modificado.
"w+"	Cria um arquivo texto para leitura e gravação. Se o arquivo existir, o conteúdo anterior será destruído. Se não existir, será criado.
"a+"	Abre um arquivo texto para gravação e leitura. Os dados serão adicionados no fim do arquivo se ele já existir, ou um novo arquivo será criado, no caso de arquivo não existente anteriormente.
"r+b"	Abre um arquivo binário para leitura e escrita. O mesmo que "r+" acima, só que o arquivo é binário.
"w+b"	Cria um arquivo binário para leitura e escrita. O mesmo que "w+" acima, só que o arquivo é binário.
"a+b"	Acrescenta dados ou cria um arquivo binário para leitura e escrita. O mesmo que "a+" acima, só que o arquivo é binário.



# Acesso a Arquivos

- `fopen()`: exemplo



```
FILE *fp;           // Declaração da estrutura

fp=fopen ("exemplo.bin","wb"); /* o arquivo
    se chama exemplo.bin e está localizado no
    diretório corrente */

if (!fp)
    printf ("Erro na abertura do arquivo.");
```

SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...



# Acesso a Arquivos

- Note que no exemplo anterior, o programa continua executando mesmo com erro na abertura de arquivo.

```
#include <stdio.h>

#include <stdlib.h> /* Para a função exit() */

main (void)

{

FILE *fp;

...

fp=fopen ("exemplo.bin","wb");

if (!fp)

{

    printf ("Erro na abertura do arquivo. Fim de programa.");

    exit (1);

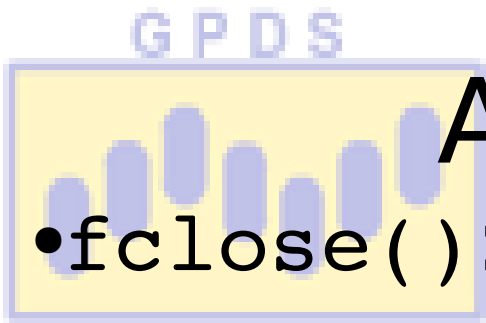
}

...

return 0;

}
```





# Acesso a Arquivos

- `fclose()`: Fechamento de arquivo

Quando acabamos de usar um arquivo que abrimos, devemos fechá-lo. Para tanto usa-se a função `fclose()`:

```
int fclose (FILE *fp);
```

O ponteiro `fp` passado à função `fclose()` determina o arquivo a ser fechado. A função retorna zero no caso de sucesso.

SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...



# Acesso a Arquivos

- `putc()`: Escreve um caracter em arquivo

```
int putc (int ch, FILE *fp);
```

- `getc()`: Retorna um caractere lido do arquivo.

```
int getc (FILE *fp);
```

- `feof()`: Ela retorna zero enquanto o arquivo não chegar ao fim (EOF).

```
int feof (FILE *fp);
```

- `fputs()`: Escreve uma string num arquivo.

```
char *fputs (char *str, FILE *fp);
```

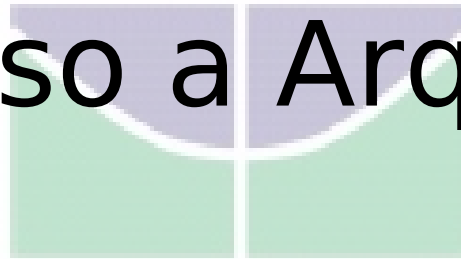
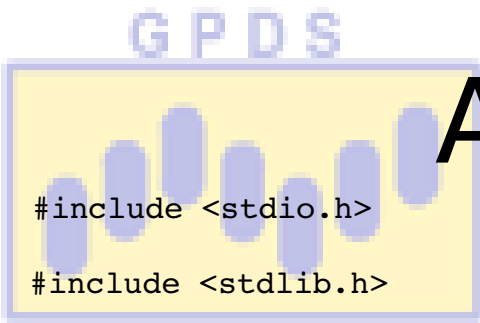
- `fgets()`: Lê uma string num arquivo.

```
char *fgets (char *str, int tamanho, FILE *fp);
```

- `ferror()`: Retorna zero se nenhum erro ocorreu durante durante um acesso ao arquivo.

```
int ferror (FILE *fp);
```

# Acesso a Arquivos



```
int main()

{

FILE *fp;

char string[100];

int i;

fp = fopen("arquivo.txt","w");    /* Arquivo ASCII, para escrita */

if(!fp) {

    printf( "Erro na abertura do arquivo");

    exit(0);

}

printf("Entre com a string a ser gravada no arquivo:");

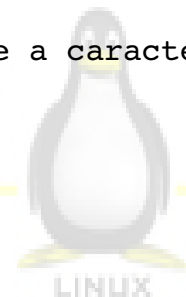
gets(string);

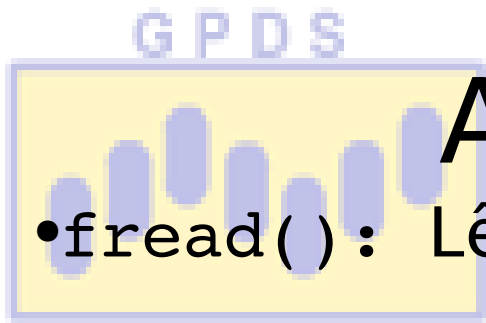
for(i=0; string[i]; i++) putc(string[i], fp); /* Grava a string, caractere a caractere */

fclose(fp);

return 0;

}
```





# Acesso a Arquivos

- `fread()`: Lê bloco de dados.

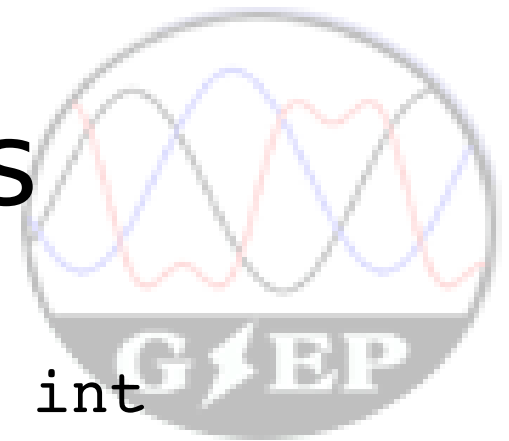
```
unsigned fread (void *buffer, int  
numero_de_bytes, int count, FILE *fp);
```

O buffer é a região de memória na qual serão armazenados os dados lidos. O número de bytes é o tamanho da unidade a ser lida. count indica quantas unidades devem ser lidas. Isto significa que o número total de bytes lidos é:

$\text{numero\_de\_bytes} * \text{count}$

A função retorna o número de unidades efetivamente lidas. Este número pode ser menor que count quando o fim do arquivo for encontrado ou ocorrer algum erro.

Quando o arquivo for aberto para dados binários, `fread()` pode ler qualquer tipo de dados.





# Acesso a Arquivos

- `fwrite()`: Escreve bloco de dados.

```
unsigned fwrite(void *buffer, int  
numero_de_bytes, int count, FILE *fp);
```

O buffer é a região de memória na qual serão armazenados os dados a serem escritos. O número de bytes é o tamanho da unidade a ser lida. `count` indica quantas unidades devem ser lidas. Isto significa que o número total de bytes lidos é:

$\text{numero\_de\_bytes} * \text{count}$

A função retorna o número de unidades efetivamente escritas. Este número pode ser menor que `count` quando o fim do arquivo for encontrado ou ocorrer algum erro.

SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...



# Acesso a Arquivos

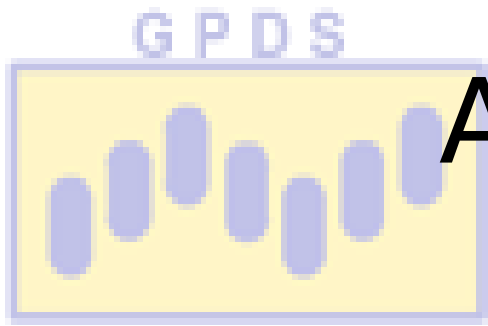
- `fseek()`: move a posição corrente de leitura ou escrita no arquivo de um valor especificado, a partir de um ponto especificado.

```
int fseek (FILE *fp, long numbytes, int origem);
```

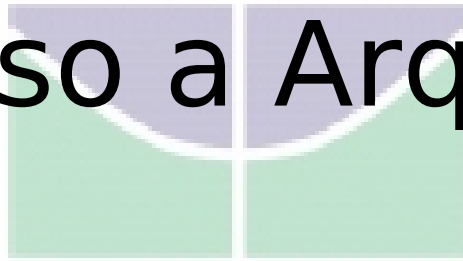
O parâmetro origem determina a partir de onde os numbytes de movimentação serão contados. Os valores possíveis são definidos por macros em `stdio.h` e são:

<i>Nome</i>	<i>Valor</i>	<i>Significado</i>
SEEK_SET	0	Início do arquivo
SEEK_CUR	1	Posição corrente do arquivo
SEEK_END	2	Fim do arquivo

Tendo-se definido a partir de onde irá se contar, numbytes determina quantos bytes de deslocamento serão dados na posição atual.



# Acesso a Arquivos



- `rewind()`: retorna a posição corrente do arquivo para o início.

```
void rewind (FILE *fp);
```

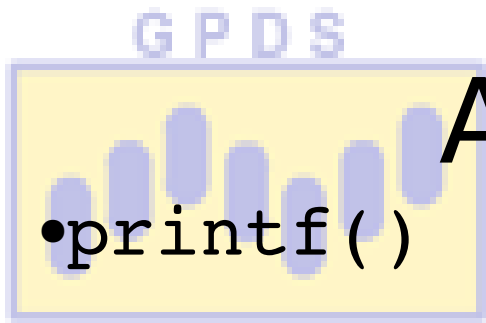
- `remove()`: Apaga um arquivo especificado.

```
int remove (char *nome_do_arquivo);
```

SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...





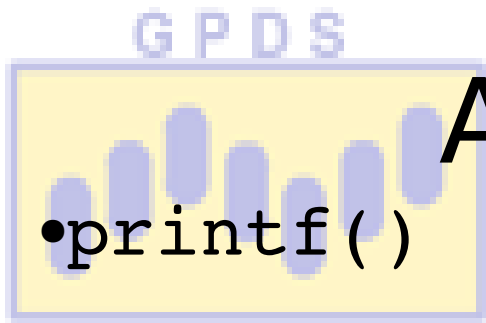
# Acesso a Arquivos

```
int printf (char *str,...);
```

Esta função tem um número de argumentos variável diretamente relacionado com a string de controle **str**, que deve ser fornecida como primeiro argumento. A string de controle tem dois componentes. O primeiro são caracteres a serem impressos na tela. O segundo são os comandos de formato. Os comandos de formato são precedidos de **%**. A cada comando de formato deve corresponder um argumento na função `printf()`. Se isto não ocorrer podem acontecer erros imprevisíveis no programa.





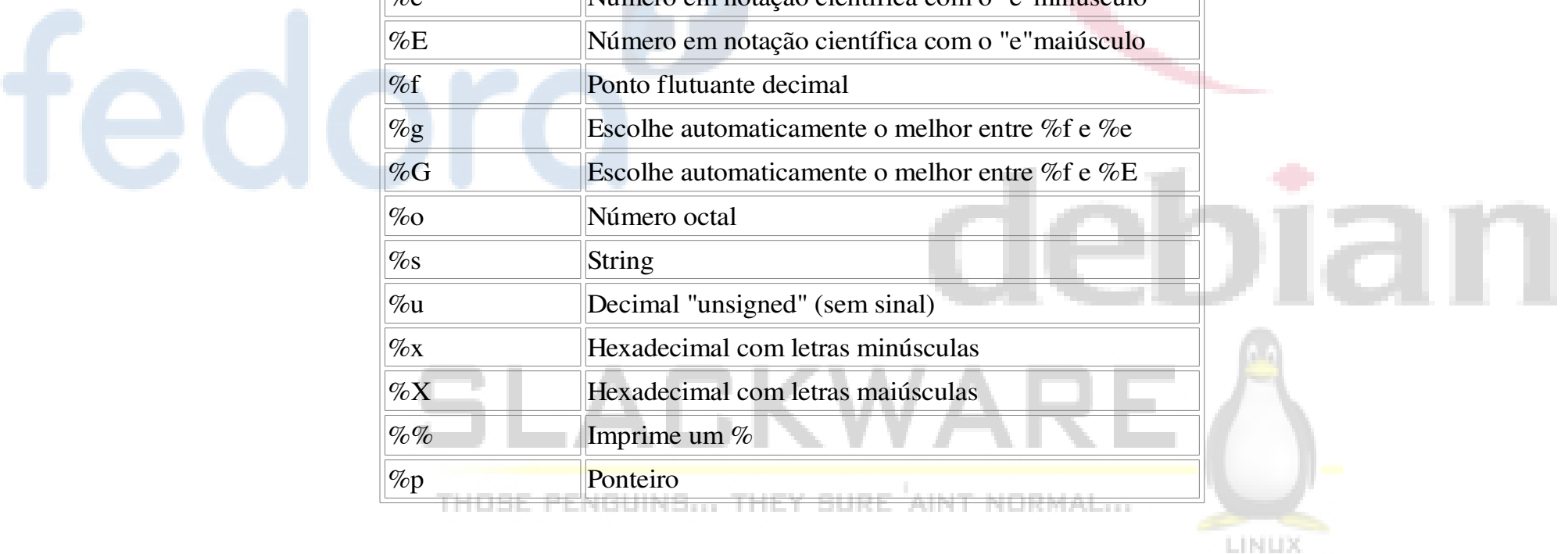


# Acesso a Arquivos

- `printf()`

- Tabela de códigos de formato:

Código	Formato
<code>%c</code>	Um caracter (char)
<code>%d</code>	Um número inteiro decimal (int)
<code>%i</code>	O mesmo que <code>%d</code>
<code>%e</code>	Número em notação científica com o "e"minúsculo
<code>%E</code>	Número em notação científica com o "e"maiúsculo
<code>%f</code>	Ponto flutuante decimal
<code>%g</code>	Escolhe automaticamente o melhor entre <code>%f</code> e <code>%e</code>
<code>%G</code>	Escolhe automaticamente o melhor entre <code>%f</code> e <code>%E</code>
<code>%o</code>	Número octal
<code>%s</code>	String
<code>%u</code>	Decimal "unsigned" (sem sinal)
<code>%x</code>	Hexadecimal com letras minúsculas
<code>%X</code>	Hexadecimal com letras maiúsculas
<code>%%</code>	Imprime um %
<code>%p</code>	Ponteiro



# Acesso a Arquivos

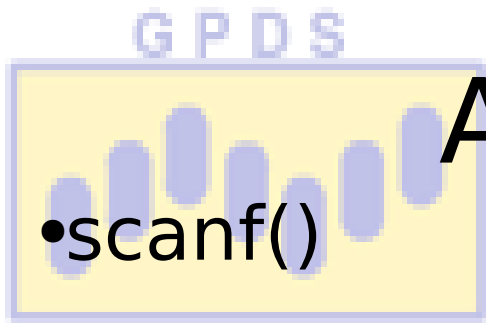
## •printf(): Exemplos

Código	Imprime
<code>printf ("Um %%c %s", 'c', "char");</code>	Um %c char
<code>printf ("%X %f %e",107,49.67,49.67);</code>	6B 49.67 4.967e1
<code>printf ("%d %o",10,10);</code>	10 12
<code>printf ("%5.2f",456.671);</code>	456.67
<code>printf ("%5.2f",2.671);</code>	2.67
<code>printf ("%10s","Ola");</code>	Ola

Temos então que `%5d` reservará cinco casas para o número.

O alinhamento padrão é à direita. Para se alinhar um número à esquerda usa-se um sinal negativo' antes do número de casas. Então `%-5d` será o nosso inteiro com o número mínimo de cinco casas, só que justificado a esquerda.

Pode-se indicar o número de casas decimais de um número de ponto flutuante. Por exemplo, a notação `%10.4f` indica um ponto flutuante de comprimento total dez casas decimais inteiras e com quatro casas decimais de precisão. Entretanto, esta mesma notação, quando aplicada a tipos como inteiros e strings indica o número mínimo e máximo de casas. Então `%5.8d` é um inteiro com comprimento mínimo de cinco e máximo de oito.



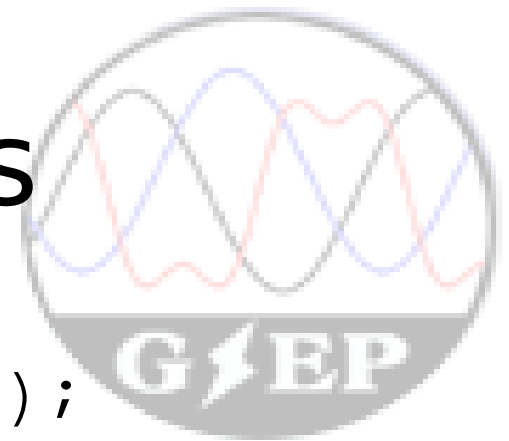
- scanf()

# Acesso a Arquivos

```
int scanf (char *str, ...);
```

A string de controle `str` determina, assim como com a função `printf()`, quantos parâmetros a função vai necessitar. Devemos sempre nos lembrar que a função `scanf()` deve receber ponteiros como parâmetros. Isto significa que as variáveis que não sejam por natureza ponteiros devem ser passadas precedidas do operador `&`.

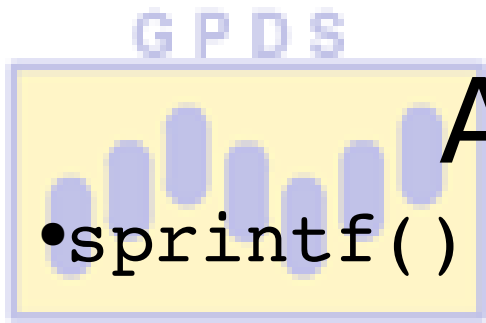
Código	Formato
%hi	Um short int
%li	Um long int
%lf	Um double



SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...





# Acesso a Arquivos

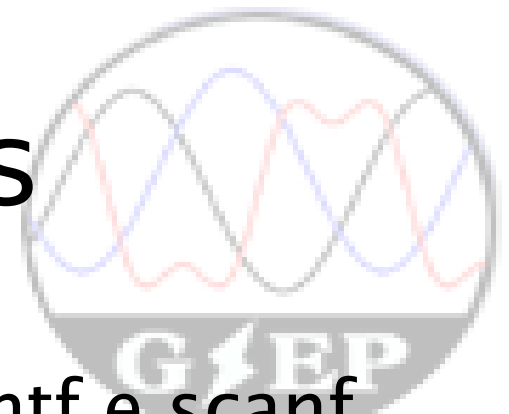
- `sprintf()` e `sscanf()`

`sprintf` e `sscanf` são semelhantes a `printf` e `scanf`.

Porém, ao invés de escreverem na saída padrão ou lerem da entrada padrão, escrevem ou leem em uma string. Os protótipos são:

```
int sprintf (char *destino, char *controle, ...);
```

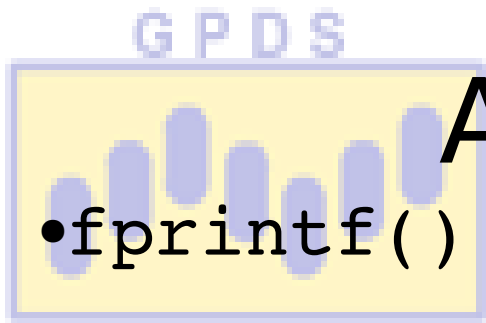
```
int sscanf (char *destino, char *controle, ...);
```



SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...





# Acesso a Arquivos

- `fprintf()`

A função `fprintf()` funciona como a função `printf()`. A diferença é que a saída de `fprintf()` é um arquivo e não a tela do computador. Protótipo:

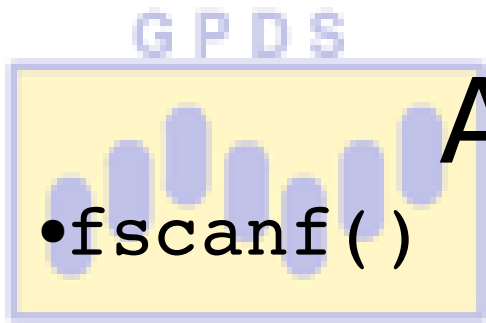
```
int fprintf (FILE *fp, char *str, ...);
```

Como já poderíamos esperar, a única diferença do protótipo de `fprintf()` para o de `printf()` é a especificação do arquivo destino por meio do ponteiro de arquivo.

SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...





# Acesso a Arquivos

- `fscanf()`

A função `fscanf()` funciona como a função `scanf()`. A diferença é que `fscanf()` lê de um arquivo e não do teclado do computador. Protótipo:

```
int fscanf (FILE *fp, char *str, ...);
```

Como já poderíamos esperar, a única diferença do protótipo de `fscanf()` para o de `scanf()` é a especificação do arquivo destino através do ponteiro de arquivo.

SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...



# Programação em C no Linux: recursos do sistema

PROCESSOS E SINAIS

SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...



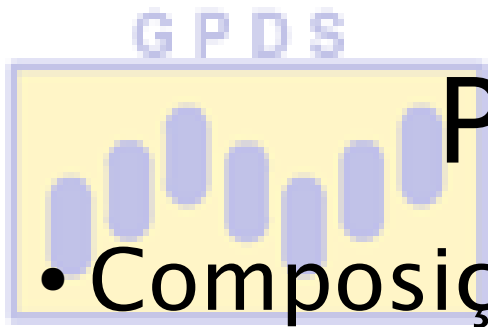
LINUX

# Processos e Sinais

- Processo é a forma de representar um programa em execução em um sistema operacional. É o processo que utiliza os recursos do computador – processador, memória, dispositivos, etc – para a realização das tarefas para as quais a máquina é destinada. Tipicamente o sistema UNIX compartilha códigos e bibliotecas entre processos, de modo que a memória seja ocupada sem redundância.
- Aplicativos desenvolvidos utilizando multi-processos permitem executar mais de uma operação simultaneamente, ou até mesmo utilizar programas prontos.

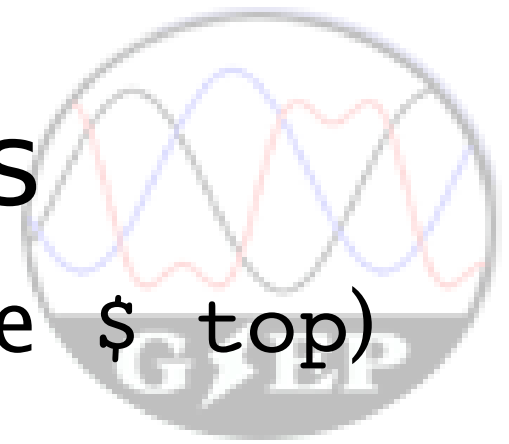






# Processos e Sinais

- Composição de um processo (vide `$ top`)
  - Proprietário do processo;
  - Estado do processo (em espera, em execução, etc);
  - Prioridade de execução; (vide `$ nice`)
  - Recursos de memória.



fedora

debian

SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...





# Processos e Sinais

- O trabalho de gerenciamento de processos precisa contar com as informações mencionadas anteriormente e com outras de igual importância para que as tarefas sejam executadas da maneira mais eficiente. Um dos meios usados para isso é atribuir a cada processo uma identificação denominada PID (Process Identifier).
- Os PID são valores inteiros de 16 bits que são alocados sequencialmente pelo sistema operacional à medida que novos processos são criados, evitando assim a colisão entre suas identificações.
- Para acessar esse valor dentro de um programa:

```
int getpid()
```

THOSE PENGUINS... THEY SURE 'RENT NORMAL...





# Processos e Sinais

- Os sistemas baseados em Unix precisam que um processo já existente se duplique para que a cópia possa ser atribuída a uma tarefa nova. Quando isso ocorre, o processo "copiado" recebe o nome de "processo pai", enquanto que o novo é denominado "processo filho". É nesse ponto que o PPID (Parent Process Identifier) passa a ser usado: o PPID de um processo nada mais é do que o PID de seu processo pai.

```
(int) getppid ()
```

SLACKWARE

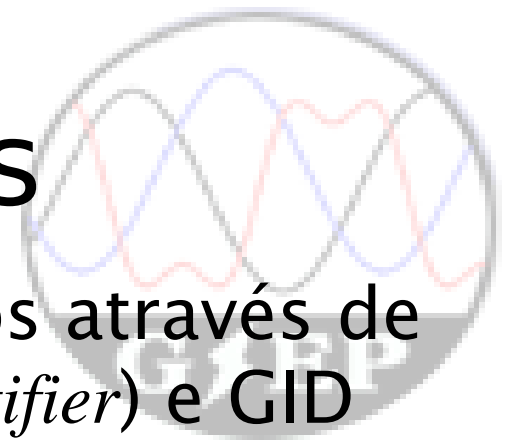
THOSE PENGUINS... THEY SURE 'AINT NORMAL...



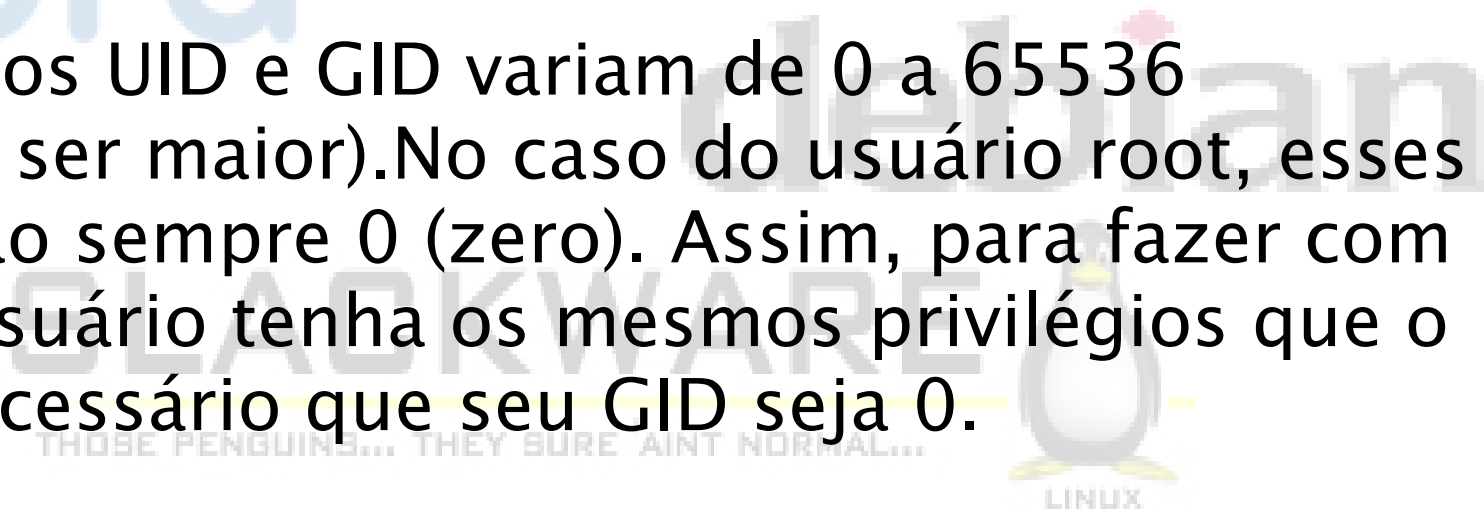


# Processos e Sinais

- O Linux gerencia os usuários e os grupos através de números conhecidos como UID (*User Identifier*) e GID (*Group Identifier*).
- Cada usuário precisa pertencer a um ou mais grupos. Como cada processo (e cada arquivo) pertence a um usuário, logo, esse processo pertence ao grupo de seu proprietário. Assim sendo, cada processo está associado a um UID e a um GID.
- Os números UID e GID variam de 0 a 65536 (podendo ser maior). No caso do usuário root, esses valores são sempre 0 (zero). Assim, para fazer com que um usuário tenha os mesmos privilégios que o root, é necessário que seu GID seja 0.



fedor





# Processos e Sinais

- Os sinais são meios usados para que os processos possam se comunicar e para que o sistema possa interferir em seu funcionamento.
- Quando um processo recebe um determinado sinal e conta com instruções sobre o que fazer com ele, tal ação é colocada em prática. Se não houver instruções pré-programadas, o próprio Linux pode executar a ação de acordo com suas rotinas.



SLACKWARE

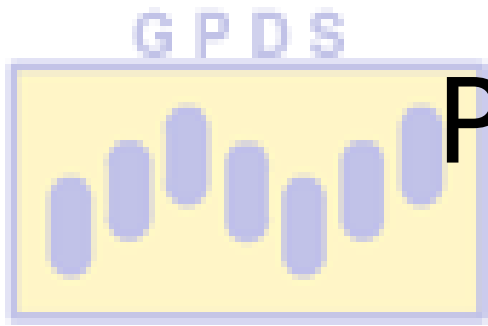
THOSE PENGUINS... THEY SURE 'AINT NORMAL...



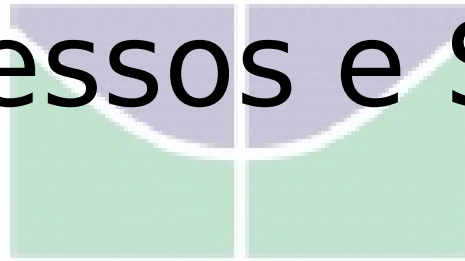
# Processos e Sinais

- Aplicativos desenvolvidos utilizando multi-processos permitem executar mais de uma operação simultaneamente, ou até mesmo utilizar programas prontos.
- Existem duas técnicas comuns utilizadas para criar um novo processo. A primeira é relativamente simples, mas deve ser utilizada com certa restrição, devido às ineficiências (uma vez que podem ocorrer falhas na execução), além de possuir um risco considerável de segurança. Já a segunda técnica é mais complexa, porém provê maior flexibilidade, velocidade e segurança.





# Processos e Sinais



- **Criação de sub-processos utilizando `system()`**
  - A função `system()` oriunda da biblioteca padrão do C (`stdlib.h`) permite, de maneira muito simples executar um comando dentro do programa em execução. A partir dele, o sistema cria um sub-processo onde o comando é executado em um shell padrão.

SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...





# Processos e Sinais

- Um exemplo bastante simples:

```
#include <stdlib.h>

#include <stdio.h>

int main ()
{
    int retorna_valor;
    retorna_valor = system ("ls -l /");
    printf("\n retorno = %d", retorna_valor);
    return retorna_valor;
}
```



SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...







# Processos e Sinais



- Retorno da função `system()`
  - A função `system()` retorna em sua saída o status do comando no shell. Se o shell não puder ser executado, o `system()` retorna o valor 127; se um outro erro ocorre, a função retorna -1.

fedora

debian

SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...





# Processos e Sinais

- Observação importante!!!

Como a função `system( )` utiliza o shell para invocar um comando, ela fica sujeita às características, limitações e falhas de segurança inerentes do shell do sistema. Além disso, não se pode garantir que uma versão particular do shell Bourne (por exemplo) esteja disponível. Ou até mesmo, restrições devido aos privilégios do usuário podem inviabilizar o sistema em questão.



SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...





# Processos e Sinais

- Criação de processos

Diferente do que ocorre no DOS e no Windows, que possuem uma família de funções específicas denominada spawn, onde estas funções cujo argumento é o nome do programa a ser executado, criando um novo processo a partir do programa em execução.

O Linux não possui uma função única que inicia e executa um novo processo em um único passo.



SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...





# Processos e Sinais

- Criação de processos utilizando `fork()` e `exec()`

No caso, o Linux disponibiliza uma função denominada `fork()`, que cria um processo filho que é uma cópia exata do processo pai. Além disso, o Linux disponibiliza um outro conjunto de funções, a família `exec()`, que chaveia entre uma instância entre dois processos. Ou seja, para se criar um novo processo, se utiliza um `fork()` para fazer a cópia do processo em questão. Em seguida o `exec` transforma um destes processos em instância de outro programa.

SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...

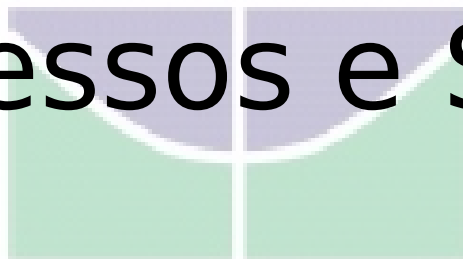




# Processos e Sinais

- **Função `fork()`**

Quando um programa chama o `fork()`, uma duplicação de processos, denominada processo filho (*child process*) é criada. O processo pai continua a executar normalmente o programa de onde o `fork()` foi chamado. Assim como o processo filho também continua a execução desde o `fork()`.



fedora

debian

SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...





# Processos e Sinais

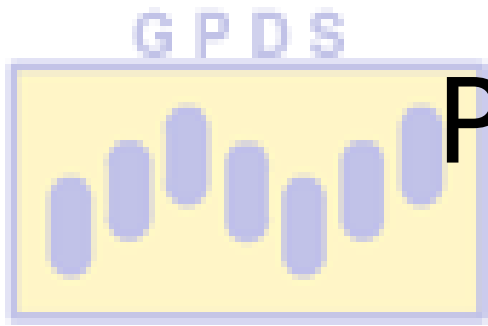
- **Função `fork()`**

Então como é feita a distinção entre estes dois processos? Primeiro, o processo filho é um novo processo e isso implica em um novo PID – diferente de seu pai. Uma maneira de distinguir o filho do pai em um programa é simplesmente fazer uma chamada com a função `getpid()`. Entretanto, a função `fork()` retorna valores distintos. O valor de retorno no processo pai é o PID do processo filho, ou seja, retorna um novo PID. Já o valor do retorno do filho é zero.

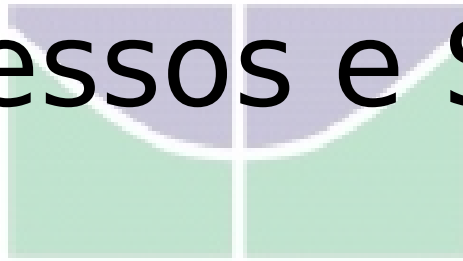
SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...





# Processos e Sinais



Processo Inicial



fork ( )



Retorna um novo PID



Processo Original



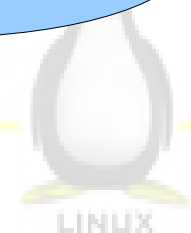
Retorna zero



Novo Processo

SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...



# Processos e Sinais

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main ()
{
    pid_t child_pid;
    printf ("the main program process ID is %d\n", (int) getpid ());
    child_pid = fork ();
    if (child_pid != 0) {
        printf ("this is the parent process, with id %d\n", (int) getpid ());
        printf ("the child's process ID is %d\n", (int) child_pid);
    }
    else
        printf ("this is the child process, with id %d\n", (int) getpid ());
    return 0;
}
```

SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...



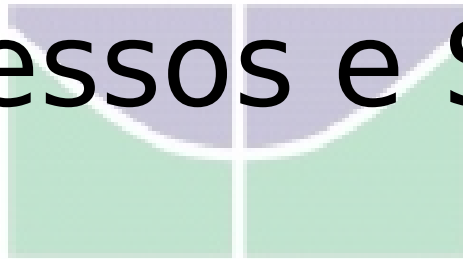




# Processos e Sinais

- **Função `exec()`**

Substitui o programa em execução de um processo por outro programa. Quando um programa chama a função `exec()`, o processo cessa imediatamente a execução do programa corrente e passa a executar um novo programa do início, isso se assumirmos que a chamada não possua ou encontre nenhum erro.



fedora

debian

SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...





# Processos e Sinais

- **Função `exec()`**

A família `exec` são funções que variam sutilmente na sua funcionalidade e também na maneira em que são chamados.

Funções que contêm a letra 'p' em seus nomes (`execvp` e `exec1p`) aceitam que o nome ou procura do programa esteja no *current path*; funções que não possuem o 'p' devem conter o caminho completo do programa a ser executado.

SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...





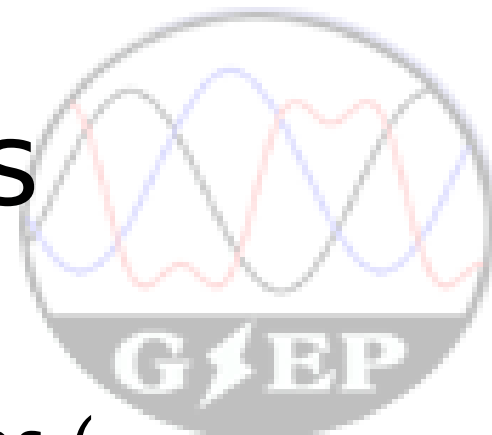
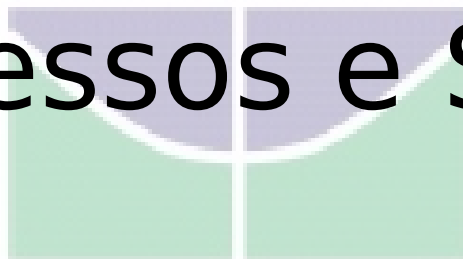
# Processos e Sinais

- **Função `exec()`**

Funções que contêm a letra 'v' em seus nomes (`execv`, `execvp` e `execve`) aceitam que a lista de argumentos do novo programa sejam nulos. Funções que contêm a letra 'l' aceitam em sua lista de argumentos a utilização de mecanismos *varargs* em linguagem C.

Funções que contêm a letra 'e' em seus nomes (`exece` e `execle`) aceitam um argumento adicional.

Como a função `exec` substitui o programa em execução por um outro, ele não retorna valor algum, exceto quando um erro ocorre.



SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...



# Processos e Sinais

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int spawn (char* program, char** arg_list)
{
    pid_t child_pid;
    child_pid = fork ();
    if (child_pid != 0)
        return child_pid;
    else {
        execvp (program, arg_list);
        fprintf (stderr, "an error occurred in execvp\n");
        abort ();
    }
}
```

Exemplo: Parte 1/2

SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...



# Processos e Sinais

## Exemplo: Parte 2/2

```
int main ()
{
    /* Lista de argumentos para o comando "ls". */
    char* arg_list[] = {
        "ls", /* argv[0], nome do programa. */
        "-l",
        "/",
        NULL /* A lista de argumentos deve sempre terminar em NULL. */
    };
    /* Criar um processo filho executando o comando "ls". */
    spawn ("ls", arg_list);
    printf ("done with main program\n");
    return 0;
}
```

SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...



# Processos e Sinais

- **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 (por exemplo, falha de segmentação, violação de memória, erros de entrada e saída, etc). Deve-se notar que este mecanismo possibilita ainda a comunicação e manipulação de processos.



# Processos e Sinais




- **Sinais**

Um sinal (com exceção do SIGKILL) pode ser tratado de três maneiras distintas em UNIX:

1) pode ser simplesmente ignorado. Por exemplo, o programa pode ignorar as interrupções de teclado geradas pelo usuário (e exatamente o que se passa quando um processo é executado em background).

2) pode ser interceptado. Neste caso, na recepção do sinal, a execução de um processo é desviado para o procedimento especificado pelo usuário, para depois retomar a execução no ponto de onde foi interrompido.

3) o comportamento padrão (default) pode ser aplicado à um processo após a recepção de um sinal.



# Processos e Sinais

Os sinais são identificados pelo sistema por um número inteiro. O arquivo `/usr/include/signal.h` contém a lista de sinais acessíveis ao usuário. Cada sinal é caracterizado por um mnemônico. Os sinais mais usados nas aplicações em UNIX são listados a seguir:

**SIGHUP (1) Corte:** sinal emitido aos processos associados a um terminal quando este se “desconecta”. Este sinal também é emitido a cada processo de um grupo quando o chefe termina sua execução.

**SIGINT (2) Interrupção:** sinal emitido aos processos do terminal quando as teclas de interrupção (por exemplo: INTR, CTRL+c) do teclado são acionadas.

**SIGQUIT (3)\* Abandono:** sinal emitido aos processos do terminal quando com a tecla de abandono (QUIT ou CTRL+d) do teclado são acionadas.

**SIGILL (4)\* Instrução ilegal:** emitido quando uma instrução ilegal é detectada.





# Processos e Sinais

**SIGTRAP (5)\*** Problemas com *trace*: emitido após cada intrução em caso de geração de *traces* dos processos (utilização da primitiva `ptrace()`)

**SIGIOT (6)\*** Problemas de instrução de E/S: emitido em caso de problemas de hardware.

**SIGEMT (7)** Problemas de instrução no emulador: emitido em caso de erro material dependente da implementação.

**SIGFPE (8)\*** Emitido em caso de 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.

**SIGKILL (9)** Destruição: “arma absoluta” para matar os processos. Não pode ser ignorada, tampouco interceptada (existe ainda o SIGTERM para uma morte mais “suave” para processos).

**SIGBUS (10)\*** Emitido em caso de erro sobre o barramento.



# Processos e Sinais

**SIGSEGV (11)\*** Emitido em caso de violação da segmentação: tentativa de acesso a um dado fora do domínio de endereçamento do processo.

**SIGSYS (12)\*** Argumento incorreto de uma chamada de sistema.

**SIGPIPE (13)** Escrita sobre um pipe não aberto em leitura.

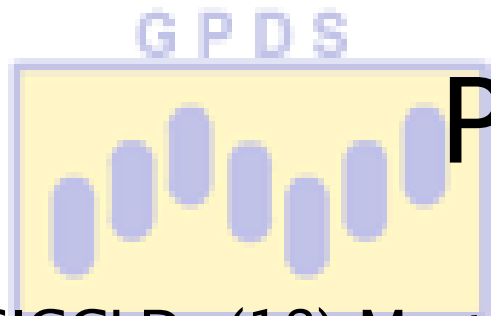
**SIGALRM (14)** Relógio: emitido quando o relógio de um processo pára. O relógio é colocado em funcionamento utilizando a primitiva alarm().

**SIGTERM (15)** Terminação por software: emitido quando o processo termina de maneira normal. Pode ainda ser utilizado quando o sistema quer por fim à execução de todos os processos ativos.

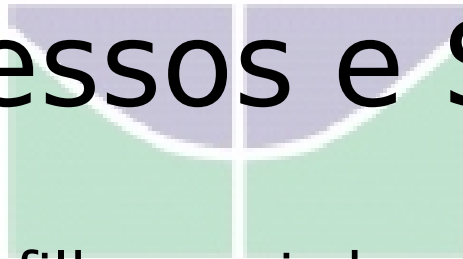
**SIGUSR1 (16)** Primeiro sinal disponível ao usuário: utilizado para a comunicação entre processos.

**SIGUSR2 (17)** Outro sinal disponível ao usuário: utilizado para comunicação interprocessual.





# Processos e Sinais



SIGCLD (18) Morte de um filho: enviado ao pai pela terminação de um processo filho.

SIGPWR (19) Reativação sobre pane elétrica.

\* Os sinais que geram um arquivo *core* no disco quando não são corretamente tratados.

SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...



# Processos e Sinais

O sinal SIGCLD se comporta diferentemente dos outros. Se ele for ignorado, a terminação de um processo filho não irá acarretar na criação de um processo zumbi.

Exemplo: O programa a seguir gera um processo zumbi quando o pai é informado da morte do filho por meio do sinal SIGCLD.

```
#include <stdio.h>
#include <unistd.h>
int main() {
    if (fork() != 0)
        while(1) ;
    return(0);
}
```

Para verificar a criação do processo zumbi basta:

```
$ ./nome_do_executavel &
$ ps
```

SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...



# Processos e Sinais

Nesse caso, o pai ignora o sinal SIGCLD, e seu filho não vai mais se tornar um processo zumbi.

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

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

```
#include <unistd.h>
```

```
int main() {
```

```
    signal(SIGCLD, SIG_IGN) ; /* ignora o sinal  
    SIGCLD */
```

```
    if (fork() != 0)
```

```
        while(1) ;
```

```
    return(0);
```

```
}
```

Para verificar se o processo zumbi  
foi criado, basta:

```
$ ./nome_do_executavel &  
$ ps
```

GPDS

# Processos e Sinais

Emissão de sinais:

`kill()`: Termina um processo

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

```
int kill(pid, sig) /* emissao de um sinal */
```

Valor de retorno: 0 se o sinal foi enviado, -1 se não foi. A primitiva `kill()` emite ao processo de número **pid** o sinal de número **sig**. Além disso, se o valor inteiro **sig** é nulo, nenhum sinal é enviado, e o valor de retorno vai informar se o número de `pid` é um número de um processo ou não.

# Processos e Sinais

## Utilização do parâmetro pid:

Se  $\text{pid} > 1$ : pid designará o processo com ID igual a pid.

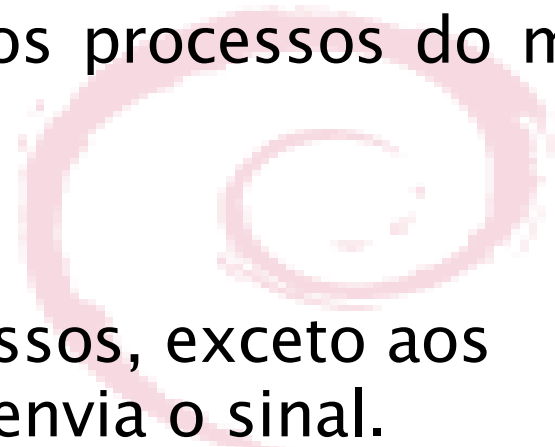
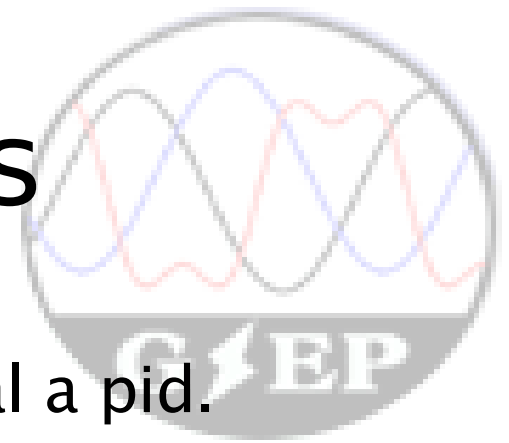
Se  $\text{pid} = 0$ : o sinal é enviado a todos os processos do mesmo grupo que o emissor.

Se  $\text{pid} = 1$ :

**SU**: o sinal é enviado a todos os processos, exceto aos processos do sistema e ao processo que envia o sinal.

**UC**: o sinal é enviado a todos os processos com ID do usuário real igual ao ID do usuário efetivo do processo que envia o sinal (é uma forma de matar todos os processos que se é proprietário, independente do grupo de processos ao qual se pertence).

Se  $\text{pid} < 1$ : o sinal é enviado a todos os processos para os quais o ID do grupo de processos (pgid) é igual ao valor absoluto de pid.



fedorora

Ubuntu

BLACKWARE  
THOSE PENGUINS... THEY SURE AINT NORMAL...



GPDS

# Processos e Sinais

Emissão de sinais:

```
alarm( ):
```

```
#include <unistd.h>
```

```
unsigned int alarm(unsigned int secs)
```

A primitiva `alarm()` envia um sinal `SIGALRM` ao processo chamando após um intervalo de tempo **secs** (em segundos) passado como argumento, depois reinicia o relógio de alarme. Na chamada da primitiva, o relógio é reiniciado a **secs** segundos e é decrementado até 0. Esta primitiva pode ser utilizada, por exemplo, para forçar a leitura do teclado dentro de um dado intervalo de tempo. O tratamento do sinal deve estar previsto no programa, senão o processo será finalizado ao recebê-lo.

THOSE PENGUINS... THEY SURE 'AINT NORMAL...

LINUX



# Processos e Sinais

## Parte 1/2

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

void it_horloge(int sig) /* tratamento do desvio sobre SIGALRM */
{
    printf("recepcao do sinal %d : SIGALRM\n",sig) ;
    printf("atencao, o principal reassume o comando\n") ;
}

void it_quit(int sig) /* tratamento do desvio sobre SIGALRM */
{
    printf("recepcao do sinal %d : SIGINT\n",sig) ;
    printf("Por que eu ?\n") ;
}
```

THOSE PENGUINS... THEY SURE 'AINT NORMAL...





```
int main()
```

```
{
```

```
    unsigned sec ;
```

```
    signal(SIGINT,it_quit); /* interceptacao do ctrl-c */
```

```
    signal(SIGALRM,it_horloge); /* interceptacao do sinal de alarme */
```

```
    printf("Armando o alarme para 10 segundos\n");
```

```
    sec=alarm(10);
```

```
    printf("valor retornado por alarm: %d\n",sec) ;
```

```
    printf("Paciencia... Vamos esperar 3 segundos com sleep\n");
```

```
    sleep(3) ;
```

```
    printf("Rearmando alarm(5) antes de chegar o sinal precedente\n");
```

```
    sec=alarm(5);
```

```
    printf("novo valor retornado por alarm: %d\n",sec);
```

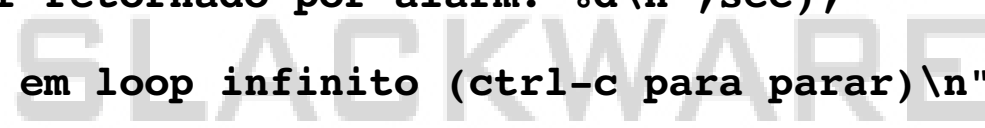
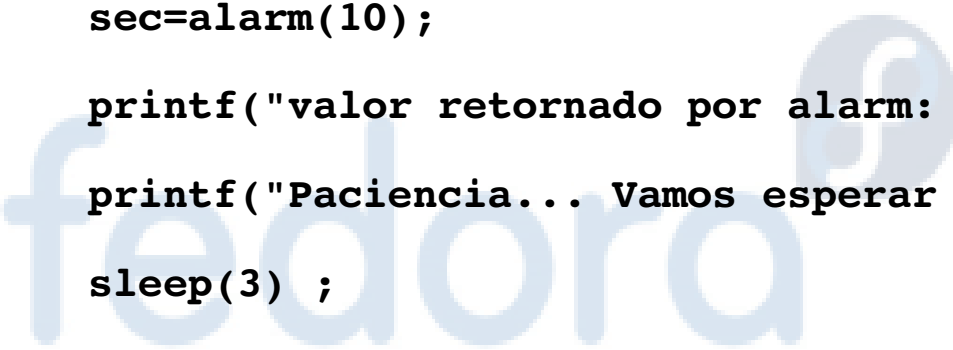
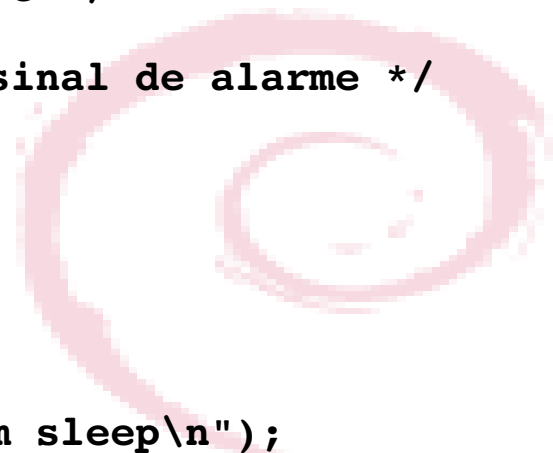
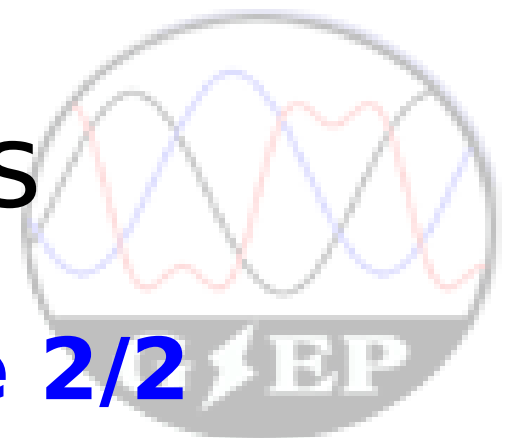
```
    printf("Principal em loop infinito (ctrl-c para parar)\n");
```

```
    for(;;);
```

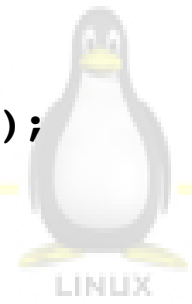
```
}
```

# Processos e Sinais

## Parte 2/2



THOSE PENGUINS... THEY SURE 'AINT NORMAL...



LINUX

# Processos e Sinais

## Recepção de sinais:

`signal():`

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

```
typedef void (*sighandler_t)(int);
```

```
sighandler_t signal(int signum, sighandler_t handler);
```

Valor de retorno: o valor anterior do manipulador do sinal, ou SIG\_ERR (normalmente -1) quando houver erro. A chamada de sistema `signal()` define um novo manipulador (*handler*) para o sinal especificado pelo número `signum`. Em outras palavras, ela intercepta o sinal `signum`. O manipulador do sinal é "setado" para `handler`, que é um ponteiro para uma função que pode assumir um dos três seguintes valores: SIG\_DFL: indica a escolha da ação *default* para o sinal. A recepção de um sinal por um processo provoca a terminação deste processo, menos para SIG\_CLD e SIG\_PWR, que são ignorados por *default*.

# Processos e Sinais

`SIG_IGN`: indica que o sinal deve ser ignorado: o processo é imunizado contra este sinal. Lembrando sempre que o sinal `SIG_KILL` nunca pode ser ignorado.

Um ponteiro para uma função (nome da função): implica na captura do sinal. A função é chamada quando o sinal chega, e após sua execução, o tratamento do processo recomeça onde ele foi interrompido.

Não se pode proceder um desvio na recepção de um sinal `SIG_KILL` pois esse sinal não pode ser interceptado, nem para `SIG_STOP`.

Pode-se notar então que é possível modificar o comportamento de um processo na chegada de um dado sinal. E exatamente isso que se passa para um certo número de processos canônicos do sistema: o shell, por exemplo, ao receber um sinal `SIG_INT` irá escrever na tela o prompt (e não será interrompido).

SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...



GPDS

# Processos e Sinais

## Recepção de sinais:

```
pause()
```

```
#include <unistd.h>
```

```
int pause(void) /* espera de um sinal qualquer */
```

Valor de retorno: sempre retorna -1.

A primitiva `pause()` corresponde a uma espera simples. Ela não faz nada, nem espera nada de particular. Entretanto, uma vez que a chegada de um sinal interrompe toda primitiva bloqueada, pode-se dizer que `pause()` espera simplesmente a chegada de um sinal.

Observe o comportamento de retorno clássico de uma primitiva bloqueada, isto é o posicionamento de `errno` em `EINTR`. Note que, geralmente, o sinal esperado por `pause()` é o relógio de alarm().

# Processos e Sinais

**Parte 1/3** ⚡ EP

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

void it_main(sig) /* tratamento sobre o 1o SIGINT */
int sig ;
{
    printf("recepção do sinal número : %d\n",sig) ;
    printf("vamos retomar o curso ?\n") ;
    printf("é o que o os profs insistem em dizer geralmente!\n")
;
}
```

GPDS

# Processos e Sinais

```
void it_fin(sig) /* tratamento sobre o 2o SIGINT */
```

```
int sig ;
```

```
{
```

```
    printf("recepção do sinal número : %d\n",sig) ;
```

```
    printf("ok, tudo bem, tudo bem ...\n") ;
```

```
    exit(1) ;
```

```
}
```

**Parte 2/3** ⚡ EP

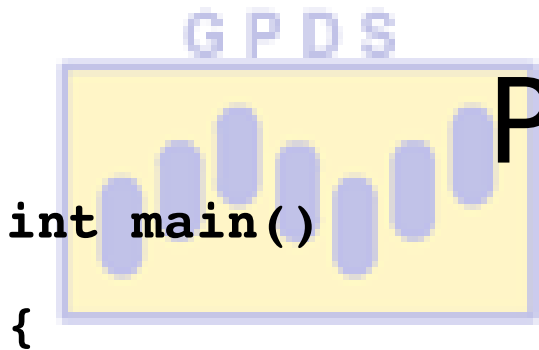
fedora

debian

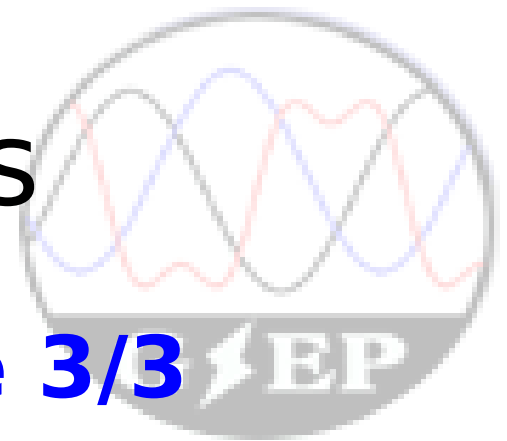
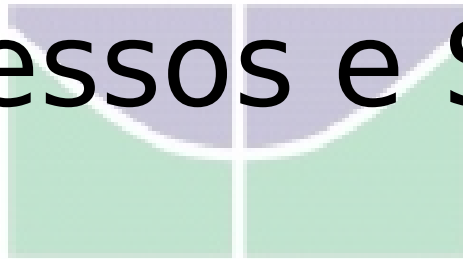
SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...





# Processos e Sinais



## Parte 3/3

```
int main()
{
    signal(SIGINT,it_main) ; /* interceptacao do 1o SIGINT */
    printf("vamos fazer uma pequena pausa (cafe!)\n") ;
    printf("digite CTRL+c para imitar o prof\n") ;
    printf("retorno de pause (com a recepcao do sinal):
%d\n",pause()) ;
    printf("errno = %d\n",errno) ;
    signal(SIGINT,it_fin) ; // rearma a interceptacao: 2o SIGINT
    for(;;) ;
    exit(0) ;
}
```

SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...





GPDS

# Processos e Sinais

## Controle da progressão de um programa:

Todos àqueles que já lançaram programas de simulação ou de cálculo numérico muito longos devem ter pensado numa forma de saber como está progredindo a aplicação durante a sua execução. Isto é perfeitamente possível por meio do envio do comando `shell kill` aos processos associados ao sinal. Os processos podem então, após a recepção deste sinal, apresentar os dados desejados.

SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...



# Processos e Sinais

## Parte 1/2 ⚡ EP

```
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <time.h>
#include <unistd.h>
/* as variaveis a serem editadas devem ser globais */
long somme = 0 ;
void it_verificacao()
{
    long t_date ;
    signal(SIGUSR1, it_verificacao) ;/* reativo SIGUSR1 */
    time(&t_date) ;
    printf("\n Data do teste : %s ", ctime(&t_date)) ;
    printf("valor da soma : %d \n", (int) somme) ;
}
```

# Processos e Sinais

```
int main()  
{  
    signal(SIGUSR1,it_verificacao) ;  
    printf ("Enviar o sinal USR1 para o processo %d  
\n",getpid()) ;  
    while(1) {  
        sleep(1);  
        somme++ ;  
    }  
    exit(0);  
}
```

Para verificar se o processo zumbi  
foi criado, basta:

```
$ ./nome_do_executavel &  
$ kill -USR1 pid
```

**Parte 2/2** ⚡ EP

SLACKWARE

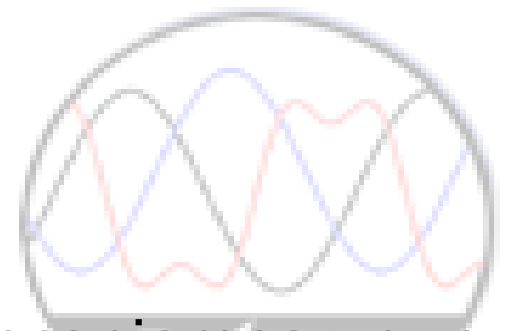
THOSE PENGUINS... THEY SURE 'AINT NORMAL...



LINUX



# Threads



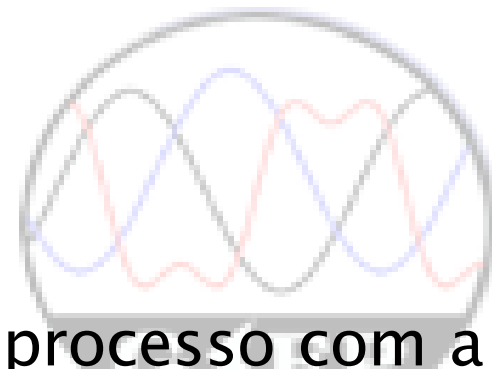
Assim como os processos, threads são mecanismos que permitem um programa realizar mais de uma operação “simultaneamente”. Além de executar threads concorrentemente; o kernel do Linux os organiza assincronamente, interrompendo cada thread de tempos em tempos de forma que todos tenham chance de ser executados.

Conceitualmente as threads co-existem com o processo, pois threads são unidades menores de execução de um processo. Quando um programa é invocado no Linux, um novo processo é criado e neste mesmo processo uma nova thread é criada, que executa o programa sequencialmente. A referida thread possui a capacidade de criar novas threads de modo que essas threads executam o mesmo código (possivelmente em segmentos de código distintos) no mesmo processo de maneira 'simultânea'.





# Threads



Diferente do que ocorre quando se cria um novo processo com a função `fork`, durante a criação de uma nova thread, nada é copiado. Ou seja, as threads existentes e criadas dividem o mesmo espaço de memória, descrição de arquivos e outros resquícios do sistema.

Logo, se em uma dada instância alguma thread muda o valor de alguma variável, a thread subsequente utilizará a variável modificada. Assim como se uma thread fecha um arquivo, outras threads não poderão ler ou escrever nele. Entretanto, podem-se tirar algumas vantagens desta característica da thread, pois não são necessários mecanismos de comunicação e sincronização complexas.

SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...





# Threads POSIX

O GNU/Linux implementa uma API (*application program interface*) conhecida como pthreads um padrão IEEE de threads denominado POSIX (*Portable Operation System Interface*). Todas as funções de threads e tipos de dados são declaradas no arquivo de header `<pthread.h>`.

Contudo as funções do pthread não são incluídas nas bibliotecas padrões do C. Ao invés disso, deve ser incluído a implementação das funcionalidades do libpthread, então é necessário adicionar à linha de comando o argumento `-lpthread` para conectar à compilação do código.

SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...



G P D S



# Threads POSIX



## Criando uma thread:

Cada thread de um processo é identificado por um thread ID, que são referidos nos programas escritos em C e C++ pelo tipo `pthread_t`.

Após a criação, cada thread executa uma “*thread function*”, que é uma função ordinária que contém o código que a thread deve executar. E depois termina a thread quando a mesma retornar o valor da função (termina de executar a função). No GNU/Linux, funções do thread utilizam apenas um parâmetro do tipo `void*`, e seu retorno também é do tipo `void*`.

SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...





# Threads POSIX



## Criando uma thread:

Para criar uma nova thread utiliza-se a função `pthread_create`, que nos disponibiliza os seguintes elementos:

Um ponteiro para variável `pthread_t`, no qual o número de identificação (ID) da nova thread é armazenado.

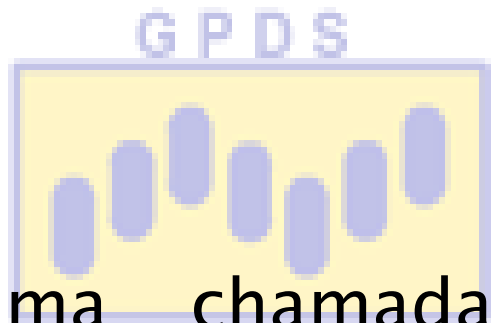
Um ponteiro para o objeto atribuído à thread. Este objeto controla detalhes de como as threads interagem com o resto do programa.

Um ponteiro para a função da thread, que é uma função ordinária do tipo: `void* (*)(void*)`

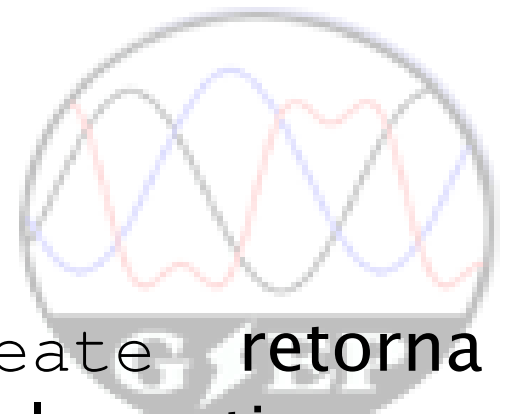
Um valor de argumento para a thread do tipo `void*`; que é passado para a função somente quando a thread é iniciada.







# Threads POSIX



Uma chamada na função `pthread_create` retorna imediatamente, enquanto a thread original continua a execução do programa. Enquanto isso a nova thread começa executando a *thread function*. Como o Linux agenda as threads assíncronamente, o programa não distingue a ordem de execução das instruções das threads. Uma maneira de se verificar isso é executando o seguinte programa:

SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...



# Threads POSIX

```
#include <pthread.h>
#include <stdio.h>
```

```
/* Imprime 'x' em stderr. */
void* print_xs (void* unused)
{
    while (1)
        fputc ('x', stderr);
    return NULL;
}
```

```
int main ()
{
    pthread_t thread_id;
    /* Cria um novo thread. A nova thread irá chamar a função print_xs */
    pthread_create (&thread_id, NULL, &print_xs, NULL);
    /* Imprime 'o' continuamente em stderr. */
    while (1)
        fputc ('o', stderr);
    return 0;
}
```

THOSE PENGUINS... THEY SURE 'AINT NORMAL...





# Threads POSIX



## Passando dados para threads:

O argumento da thread possui um método conveniente de passar dados para a thread. Como o tipo de argumento da thread é um `void*`, não é possível passar um grande conjunto de dados por meio dos argumentos. Mas existem algumas maneiras de contornar esta situação, como passar um ponteiro para uma dada estrutura ou matriz de dados. Uma técnica comum seria definir uma estrutura para cada função thread, que contém os parâmetros que a função thread “espera”.

Utilizar o argumento da thread implica em facilitar o reuso das funções threads em várias threads. Onde todas essas threads executam o mesmo código, mas utilizando dados diferentes. O programa abaixo é similar ao apresentado anteriormente; o programa cria duas threads, onde uma delas imprime 'x' e a outra 'o'. Porém, ao invés de imprimí-los infinitamente, cada um imprime um número de caracteres e depois sai retornando para a função thread. A mesma função thread `char_print`, é utilizado por ambas as thread, mas cada uma é configurada de maneira distinta ao utilizar a estrutura `char_print_parms`.

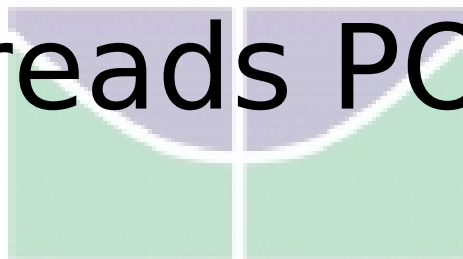
THOSE PENGUINS... THEY SURE 'AINT NORMAL...



LINUX



# Threads POSIX



```
#include <pthread.h>
#include <stdio.h>
```

```
struct char_print_parms
{
    char character;
    int count;
};
```

```
void* char_print (void* parameters)
{
    struct char_print_parms* p = (struct
char_print_parms*) parameters;
    int i;
    for (i = 0; i < p->count; ++i)
        fputc (p->character, stderr);
    return NULL;
}
```



THOSE PENGUINS... THEY SURE 'AINT NORMAL...



GPDS

# Threads POSIX

```
int main ()
{
    pthread_t thread1_id;
    pthread_t thread2_id;

    struct char_print_parms thread1_args;
    struct char_print_parms thread2_args;
    thread1_args.character = 'x';
    thread1_args.count = 30000;

    pthread_create (&thread1_id, NULL, &char_print,
&thread1_args);
    thread2_args.character = 'o';
    thread2_args.count = 20000;

    pthread_create (&thread2_id, NULL, &char_print,
&thread2_args);

    return 0;
}
```



fedora

debian

SLACKWARE

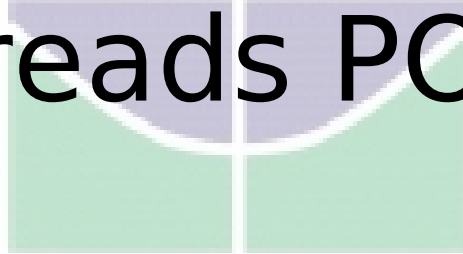
THOSE PENGUINS... THEY SURE 'AINT NORMAL...



G P D S

**Observação:**

# Threads POSIX



**Existe um problema sério neste código!!!** A thread principal (que roda a função main) cria as estruturas de parâmetro da thread (`thread1_args` e `thread2_args`) como variáveis locais, e depois passam esses ponteiros para a estrutura dos threads que foram criados. E o que previne o Linux de agendar as threads de modo que a função main termine a execução antes de finalizar as outras threads? Simplesmente nada. Mas, isso ocorrer a memória que contém o parâmetro das estruturas são desalocadas enquanto as outras threads ainda estão acessando.

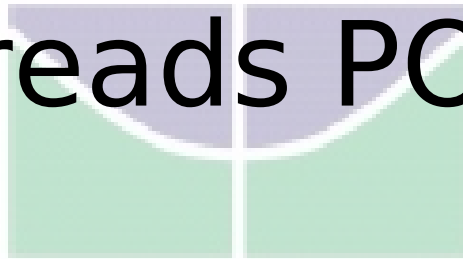
SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...





# Threads POSIX



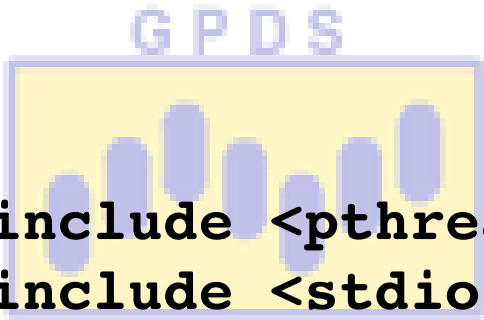
## Juntando threads:

Uma solução para forçar que a função `main()` fique ativa até que as outras threads do programa terminem é utilizar uma função que possui funcionalidade equivalente ao `wait`. Essa função é a `pthread_join()`, que possui dois argumentos: a thread ID da thread que irá esperar e um ponteiro para uma variável `void*` que receberá o valor de retorno da thread.

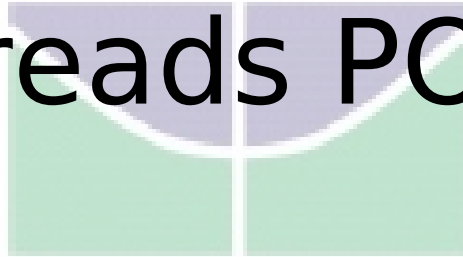
SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...





# Threads POSIX



```
#include <pthread.h>
#include <stdio.h>
```

```
struct char_print_parms
{
    char character;
    int count;
};
```

```
void* char_print (void* parameters)
{
    struct char_print_parms* p = (struct
char_print_parms*) parameters;
    int i;
    for (i = 0; i < p->count; ++i)
        fputc (p->character, stderr);
    return NULL;
}
```



THOSE PENGUINS... THEY SURE 'AINT NORMAL...





GPDS

# Threads POSIX

```
int main ()
{
    pthread_t thread1_id;
    pthread_t thread2_id;
    struct char_print_parms thread1_args;
    struct char_print_parms thread2_args;
    thread1_args.character = 'x';
    thread1_args.count = 30000;
    pthread_create (&thread1_id, NULL, &char_print,
    &thread1_args);
    thread2_args.character = 'o';
    thread2_args.count = 20000;
    pthread_create (&thread2_id, NULL, &char_print,
    &thread2_args);
    pthread_join (thread1_id, NULL);
    pthread_join (thread2_id, NULL);
    return 0;
}
```



THOSE PENGUINS... THEY SURE 'AINT NORMAL...





# Threads POSIX



## Cancelando threads:

Em circunstâncias normais, uma thread termina quando sua execução finaliza, ou por meio do retorno da função thread ou pela chamada da `pthread_exit()`. Mas é possível que uma thread requisiite o término de outra, denomina-se a essa funcionalidade o cancelamento de thread.

Para cancelar uma thread, deve-se chamar a função `pthread_cancel()`, passando como parâmetro a thread ID da thread a ser cancelada.

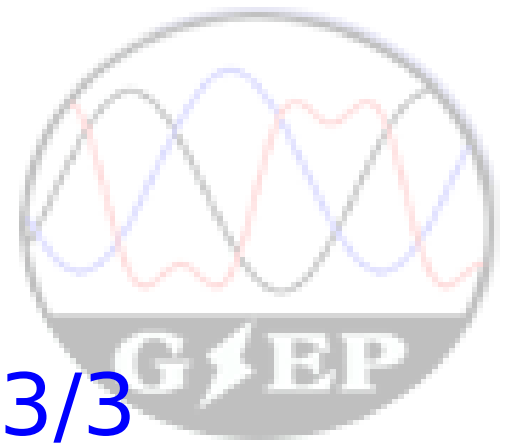
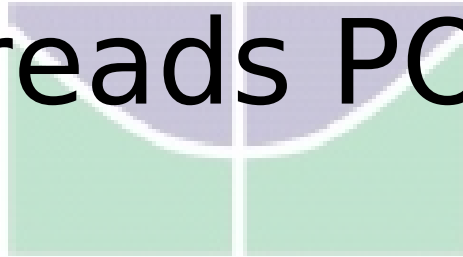
SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...





# Threads POSIX



Parte 3/3

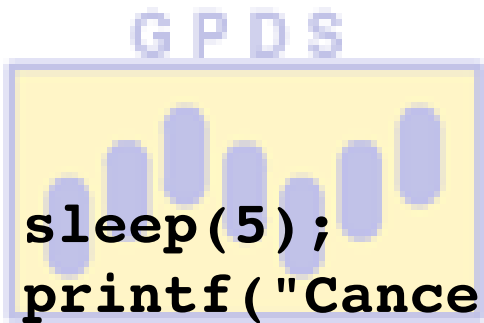
```
# include <stdio.h>
# include <stdlib.h>
# include <pthread.h>
```

```
void *thread_function(void *arg);
int main(){
    int res;
    pthread_t a_thread;
    void *thread_result;
    res = pthread_create(&a_thread, NULL, thread_function,
NULL);
    if (res != 0){
        perror("Não foi possível criar a thread!");
        exit(EXIT_FAILURE);
    }
}
```

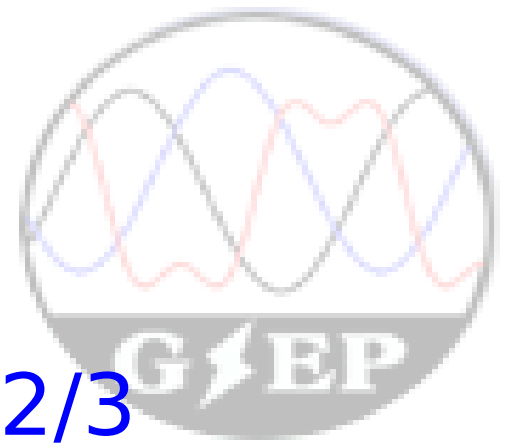
SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...





# Threads POSIX



Parte 2/3

```
sleep(5);
printf("Cancelando a thread ... \n");
res = pthread_cancel(a_thread);
if (res != 0){
    perror("Não foi possível cancelar a thread!");
    exit(EXIT_FAILURE);
}
printf("Esperando o fim da execução da thread ...
\n");
res = pthread_join(a_thread,&thread_result);
if (res != 0){
    perror("Não foi possível juntar as threads!");
    exit(EXIT_FAILURE);
}
exit(EXIT_SUCCESS);
}
```

SLACKWARE

THOSE PENGUINS... THEY SURE 'AINT NORMAL...



GPDS

# Threads POSIX

Parte 3/3

```
void *thread_function(void *arg){
    int i, res;
    res = pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
    if (res != 0){
        perror("Falha na pthread_setcancelstate");
        exit(EXIT_FAILURE);
    }
    res = pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);
    if (res != 0){
        perror("Falha na pthread_setcanceltype");
        exit(EXIT_FAILURE);
    }
    printf("Função thread executando. \n");
    for (i = 0; i < 10; i++){
        printf("Thread em execução (%d) ... \n", i);
        sleep(1);
    }
    pthread_exit(0);
}
```

THOSE PENGUINS... THEY SURE 'AINT NORMAL...

