

Então, o que é

minitalk?

Parte mandatória e bônus | Caderno de Projeto

**@fegastal | @fgastal-**

# fase 01

## **pesquisa | conceitos gerais**

# THE LINUX PROGRAMMING INTERFACE

A Linux and UNIX\* System Programming Handbook

MICHAEL KERRISK



The Linux Programming Interface - Michael Kerrisk

# 20

## SIGNALS: FUNDAMENTAL CONCEPTS

This chapter and next two chapters discuss signals. Although the fundamental concepts are simple, our discussion is quite lengthy, since there are many details to cover.

This chapter covers the following topics:

- the various different signals and their purposes;
- the circumstances in which the kernel may generate a signal for a process, and the system calls that one process may use to send a signal to another process;
- how a process responds to a signal by default, and the means by which a process can change its response to a signal, in particular, through the use of a signal handler, a programmer-defined function that is automatically invoked on receipt of a signal;
- the use of a process signal mask to block signals, and the associated notion of pending signals; and
- how a process can suspend execution and wait for the delivery of a signal.

**Java**

- | Bitwise OR
- & Bitwise AND
- ~ Bitwise Complement
- ^ Bitwise XOR
- << Left Shift
- >> Right Shift
- >>> Unsigned Right Shift

**JavaScript**

- Bitwise AND a & b
- Bitwise OR a | b
- Bitwise XOR a ^ b
- Bitwise NOT ~ a
- Left shift a << b
- Sign-propagating right shift a >> b
- Zero-fill right shift a >>> b

**Python**

- x<<y Left shift
- x>>y Right shift
- x&y Bitwise AND
- x|y Bitwise OR
- ~x&y Bitwise NOT
- x^y Bitwise XOR



 Practical use of bitwise operations - Implementing a...  
CodeVault  
14 mil visualizações • há 2 anos

Practical uses of bitwise operations - Implementing a...  
CodeVault  
14 mil visualizações • há 2 anos

 Do Addition With No Plus Sign  
Back To Back SWE  
105 mil visualizações • há 3 anos

Add Two Numbers Without The "+" Sign (Bit Shifting Basics)  
Back To Back SWE  
105 mil visualizações • há 3 anos

 Bitwise Operators JavaScript  
techsith  
12 mil visualizações • há 2 anos

Bitwise Operators JavaScript  
techsith  
12 mil visualizações • há 2 anos

 What are Bitwise Operators (And, OR, XOR) in Python |...  
WsCube Tech  
20 mil visualizações • há 1 ano

What are Bitwise Operators (And, OR, XOR) in Python |...  
WsCube Tech  
20 mil visualizações • há 1 ano

 Arduino Workshop - Chapter 4 - Bit Math  
Core Electronics  
52 mil visualizações • há 5 anos

Arduino Workshop - Chapter 4 - Bit Math  
Core Electronics  
52 mil visualizações • há 5 anos

 (Eu, a Patroa e as Crianças) T1E4 Dos Seios e Basquete...  
Cantor VITOR KEVIN - Official C...  
1,6 mi de visualizações • há 3 meses

(Eu, a Patroa e as Crianças) T1E4 Dos Seios e Basquete...  
Cantor VITOR KEVIN - Official C...  
1,6 mi de visualizações • há 3 meses

 Intro to Binary and Bitwise Operators in C++  
The Cherno

Intro to Binary and Bitwise Operators in C++  
The Cherno

## What Are Bitwise Operators And Why Do We Use Them?

48.067 visualizações 15 de nov. de 2017 What are Bitwise operators? Where do they come from? And why would you need to use them? ...mais

1,4 mil Não gostei Compartilhar Salvar ...

▶ YouTube<sup>BR</sup>

Pesquisar



## SET BIT

```
def set_bit(x, position):
    mask = 1 << position
    return x | mask
```

x	00000110	00000110
position	00000101	00100000
mask	00100000	00100110



15:01 / 34:20 • Left Shift &gt;

*Bit Manipulation Basics*



Programming with Processes

## Sending and Handling Signals in C (kill, signal, sigaction)

117.488 visualizações 20 de abr. de 2018 Patreon ►  
<https://www.patreon.com/jacobsorber> ...mais

4 mil Não gostei Compartilhar Salvar

Comentários

Jacob Sorber



Debugging with Core Dumps  
Jacob Sorber  
39 mil visualizações • há 3 anos



Is there Garbage Collection in C and C++?  
Jacob Sorber  
16 mil visualizações • há 3 meses



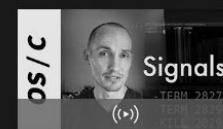
Understanding and implementing a Hash Table (in C)  
Jacob Sorber  
160 mil visualizações • há 2 anos



Understanding fork() system call for new process creation  
Edredo (Formerly Techtud)  
310 mil visualizações • há 5 anos



Reading and Writing Files in C, two ways (fopen vs. open)  
Jacob Sorber  
65 mil visualizações • há 4 anos



Mix de Sending and Handling Signals in C (kill, signal,...)  
Uma playlist personalizada para você



fork() & exec() System Calls  
Neso Academy  
229 mil visualizações • há 3 anos

C main.c > main.c (1)

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <signal.h>
4 #include <unistd.h>
5
6 int main(int argc, char* argv[]) {
7     int pid = fork();
8     if (pid == -1) {
9         return 1;
10    }
11
12    if (pid == 0) {
13
14    } else {
15        printf("What is the result of 3 x 5: ");
16    }
17
18    return 0;
19 }
```

nas.local:1919 1:23 / 11:17 • Creating two processes >  
GCC - Build and debug active file (Tutorial)

Unix Processes in C

## Communicating between processes using signals

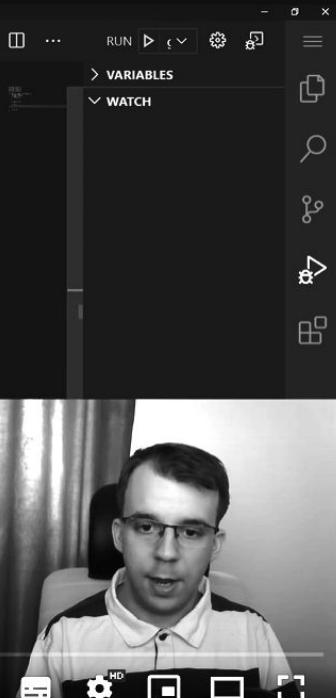
31.046 visualizações 2 de jun. de 2020 Check out our Discord server:  
<https://discord.gg/NFxT8NY...mais>

 CodeVault

 709  Não gostei  Compartilhar  Salvar ...

Comentários  Thanks for making these. I have shared your

Communicating between processes using signals - <https://www.youtube.com/watch?v=PErrlOx3LYE>



 amazon.com.br  
Novas ofertas todos os dias

 Pode acreditar.  
Anúncio · amazon.com.br

 int[] vs int\*  
ARRAYS 11:23

 SIGUSR1 and SIGUSR2  
PROCESSES (leftrightarrow)

 Send array through a PIPE  
PROCESSES 15:19

 Parsing a string  
STRINGS 10:39

Difference between arrays and pointers in C  
CodeVault  
18 mil visualizações · há 3 anos

Mix de CodeVault  
Mais desse canal para você

How to send an array through a pipe  
CodeVault  
10 mil visualizações · há 2 anos

How to parse a string in C (sscanf)  
CodeVault  
41 mil visualizações · há 3 anos



(Eu, a Patroa e as Crianças)  
T1E6 Trabalhando Nisso...  
Cantor VITOR KEVIN - Official C...  
1,1 mi de visualizações · há 3 meses

Next: [Remembering a Signal to Act On Later](#), Previous: [Blocking Signals for a Handler](#), Up: [Blocking Signals](#) [Contents][Index]

## 24.7.6 Checking for Pending Signals

You can find out which signals are pending at any time by calling `sigpending`. This function is declared in `signal.h`.

Function: `int sigpending (sigset_t *set)`

Preliminary: | MT-Safe | AS-Unsafe lock/hurd | AC-Unsafe lock/hurd | See [POSIX Safety Concepts](#).

The `sigpending` function stores information about pending signals in `set`. If there is a pending signal that is blocked from delivery, then that signal is a member of the returned set. (You can test whether a particular signal is a member of this set using `sigismember`; see [Signal Sets](#).)

The return value is `0` if successful, and `-1` on failure.

Testing whether a signal is pending is not often useful. Testing when that signal is not blocked is almost certainly bad design.

Here is an example.

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

sigset_t base_mask, waiting_mask;

sigemptyset (&base_mask);
sigaddset (&base_mask, SIGINT);
sigaddset (&base_mask, SIGTSTP);

/* Block user interrupts while doing other processing. */
sigprocmask (SIG_SETMASK, &base_mask, NULL);
...

/* After a while, check to see whether any signals are pending. */
sigpending (&waiting_mask);
if (sigismember (&waiting_mask, SIGINT)) {
    /* User has tried to kill the process. */
}
else if (sigismember (&waiting_mask, SIGTSTP)) {
    /* User has tried to stop the process. */
}
```

Remember that if there is a particular signal pending for your process, additional signals of that same type that arrive in the meantime might be discarded. For example, if a `SIGINT` signal is pending when another `SIGINT` signal arrives, the system will probably only see one of them when you `unblock` this signal.

# Introduction To Unix Signals Programming

The material on the page was adapted from a tutorial developed by Guy Kerens which can be found [here](#)

## Table Of Contents

1. [What Are Signals?](#)
2. [Sending Signals To Processes](#)
3. [Catching Signals - Signal Handlers](#)
4. [Installing Signal Handlers](#)
5. [Pending Signals](#)
6. [Sigset, Sighold, Sigrelse, Siginore, Sigpause](#)
7. [Avoiding Signal Races - Masking Signals](#)
8. [Implementing Timers Using Signals](#)

## What Are Signals?

Signals, to be short, are various notifications sent to a process in order to notify it of various "important" events. By their nature, they interrupt whatever the process is doing at this minute, and force it to handle them immediately. Each signal has an integer number that represents it (1, 2 and so on), as well as a symbolic name that is usually defined in the file `/usr/include/signal.h` or one of the files included by it directly or indirectly (`HUP`, `INT` and so on). Use the command `'kill -1'` to see a list of signals supported by your system).

Each signal may have a signal handler, which is a function that gets called when the process receives that signal. The function is called in "asynchronous mode", meaning that no where in your program you have code that calls this function directly. Instead, when the signal is sent to the process, the operating system stops the execution of the process, and "forces" it to call the signal handler function. When that signal handler function returns, the process continues execution from wherever it happened to be before the signal was received, as if this interruption never occurred.

Note for "hardwareists": If you are familiar with interrupts (you are, right?), signals are very similar in their behavior. The difference is that while interrupts are sent to the operating system by the hardware, signals are sent to the process by the operating system, or by other processes. Note that signals have nothing to do with software interrupts, which are still sent by the hardware (the CPU itself, in this case).

## Sending Signals To Processes

### Sending Signals Using The Keyboard

The most common way of sending signals to processes is using the keyboard. There are certain key presses that are interpreted by the system as requests to send signals to the process with which we are interacting:

# SIGACTION

Section: □Linux Programmer's Manual□ (2)

Updated: August 2, 2000

[Index](#) [Return to Main Contents](#)

---

## NAME

`sigaction`, `sigprocmask`, `sigpending`, `sigsuspend` - POSIX signal handling functions.

## SYNOPSIS

```
#include <signal.h>

int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);

int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);

int sigpending(sigset_t *set);

int sigsuspend(const sigset_t *mask);
```

## DESCRIPTION

The **sigaction** system call is used to change the action taken by a process on receipt of a specific signal.

*signum* specifies the signal and can be any valid signal except **SIGKILL** and **SIGSTOP**.

If *act* is non-null, the new action for signal *signum* is installed from *act*. If *oldact* is non-null, the previous action is saved in *oldact*.

The **sigaction** structure is defined as something like

```
struct sigaction {
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
```



Search or jump to...

Pull requests Issues Marketplace Explore

Watch Fork Star

mlanca-c/Minitalk Public

Watch 1

Fork 1

Star 37

Code Issues Pull requests Actions Projects Wiki Security Insights

# Home

Maria Sotomayor edited this page on 25 Jun 2021 · 18 revisions

## Intro

Pages 6

The purpose of this project is to code a small data exchange program using Unix signals.

Create a communication program - in C - in the form of a client and server. The server is the first one to be launched, having to display its PID after. The client will take as parameters the server PID and the string to be sent. Once the string has been received by the server, this one should display it.

This communication should only be done using Unix signals. Only two signals can be used **SIGUSR1** and **SIGUSR2**. The executable files are named client and server.

### Functions used in C:

- write
- signal
- sigemptyset

## Contents

1. Intro
2. Unix Processes
3. Unix Signals
4. Used Functions
  - `sigaction()` vs. `signal()`
5. Project
  - 1st Step: Make server receive a signal from client
  - 2nd Step: Convert ASCII character to Binary character
  - 3rd Step: Convert Binary character to ASCII character
  - 4th Step: Sending a string from client to server

# fase 02

## objetivo do projeto

The purpose of this  
project is to code a small  
**data exchange** program  
using **UNIX signals**."

# fase 03

## análise do subject

## REQUERIMENTOS | SUBJECT

Você deve criar um programa de comunicação na forma de um **cliente** e um **servidor**.

O servidor deve ser iniciado primeiro. Após seu lançamento, ele tem que imprimir seu PID.

O cliente toma dois **parâmetros**:

- 1) O **PID** do servidor
- 2) A **string** a ser enviada.

O cliente deve enviar **a string passada como um parâmetro** para o servidor. Uma vez **recebida a sequência de caracteres**, o servidor deve **imprimi-la**.

O servidor tem que **exibir a string muito rapidamente**.

Rapidamente significa que se você acha que leva muito tempo, então provavelmente é muito longo.

Seu servidor deve ser capaz de **receber strings de vários clientes** em uma **fila sem precisar reiniciar**.

A comunicação entre seu cliente e seu servidor tem que ser feita apenas usando **sinais UNIX**.

Você só pode usar estes dois sinais: **SIGUSR1** e **SIGUSR2**.

# SO...WHAT'S THE POINT?



client

The client will send information to the server using UNIX signals



server

The server must correctly receive and display this information

### 3.2 Nota - Comunicação entre servidor e cliente (canal bidirecional)

*"O servidor tem que exibir a string muito rapidamente. Rapidamente significa que se você acha que leva muito tempo, então provavelmente é muito longo."*

Ao pesquisar sobre o tema, surgem duas opções:

1 | usando uma função de atraso como sleep() ou usleep() **evitando a comunicação de sinais para trás e para frente entre servidor/cliente**, o que importa um atraso por char, uma vez que estaríamos enviando de forma cega (sem loop de feedback) chars do cliente para o servidor. Dependendo do atraso definido, **os sinais poderiam ficar em fila de espera.**

2 | a outra opção seria **implementar um loop de feedback** para que **sempre que o cliente enviar um char para o servidor, ele espere até que o servidor envie de volta um sinal informando ao cliente que o próximo bit pode ser enviado**. Esta solução posterior deveria ser muito mais rápida, uma vez que não haveria função de atraso. Esta solução está alinhada com o bônus: O servidor reconhece cada mensagem recebida enviando de volta um sinal para o cliente.

Como observado no subject, não devem ser enviados sinais cegamente, tornando a **comunicação bidirecional** a melhor implementação.

```
coder@fgastal--workspace-5dff7954bc-8j5pt:~/minitalk-final$ valgrind --leak-check=full ./server
==6634== Memcheck, a memory error detector
==6634== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==6634== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==6634== Command: ./server
==6634==
MinTalk [FGASTAL-]
Server PID: 6634
oi`C==6634==
==6634== Process terminating with default action of signal 2 (SIGINT)
==6634== at 0x493DFC7: pause (pause.c:29)
==6634== by 0x4014D7: main (server.c:102)
==6634==
==6634== HEAP SUMMARY:
==6634==     in use at exit: 0 bytes in 0 blocks
==6634==     total heap usage: 2 allocs, 2 frees, 5 bytes allocated
==6634==
==6634== All heap blocks were freed -- no leaks are possible
==6634==
==6634== For lists of detected and suppressed errors, rerun with: -s
==6634== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

```
coder@fgastal--workspace-5dff7954bc-8j5pt:~/minitalk-final$ valgrind --leak-check=full ./client 6634 oi
==6671== Memcheck, a memory error detector
==6671== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==6671== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==6671== Command: ./client 6634 oi
==6671==
```

```
Number of bytes: 1
Number of bytes: 2
>>> MESSAGE WAS SUCCESSFULLY SENT! :) <<<
```

```
==6671==
==6671== HEAP SUMMARY:
==6671==     in use at exit: 0 bytes in 0 blocks
==6671==     total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==6671==
==6671== All heap blocks were freed -- no leaks are possible
==6671==
==6671== For lists of detected and suppressed errors, rerun with: -s
==6671== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
coder@fgastal--workspace-5dff7954bc-8j5pt:~/minitalk-final$
```

panorama geral  
após uma boa **pesquisa** :)

Primeiro: como funciona a  
comunicação entre  
**sinais**?

Um **processo** é simplesmente um **programa em execução**.

Ele pode ser um programa de sistema, com um código de sistema ou um programa de usuário, com código de usuário.

Com um código de sistema, você vai no kernel do linux.

Esse processo pode ter também várias tarefas dentro dele: threads (ou filamentos). A corda de palha tem vários filamentos

que se amarram e formam uma corda grande.

Um processo pode ser grande, formado de **vários filamentos**, threads. Ou um processo com **uma tarefa** só.

Várias tarefas: sistemas multicore. No sistema operacional, todo o processo tem uma série de informações atreladas a ele: **blocos de controle de processos**.

Um processo pode ficar em uma fila para ser executado, como um processo de pronto, que fica na fila aguardando seu momento de ser executado pela CPU e também tem a pilha de dispositivo, que tem uma fila de dispositivo: fila de pronto.

O trabalho do sistema operacional acontece através de um **gerenciador** para analisar todos os processos que estão acontecendo e economizar os gastos da CPU. Uma fila de pronto e fila de dispositivos (I/O) ficam em lugares diferentes, então o ideal é ter tantos processos na fila de pronto quanto nas fila de dispositivos.

○ **sistema operacional vai organizar** esses processos tanto a longo prazo quanto à curto prazo, ou pode ser também um processo / schedule à médio prazo com otimizações intermediárias.

e o processo em si,  
como funciona ?

Ele pode executar uma tarefa ou várias e pode ter, se necessário, outro processo dentro dele. E aí, nisso, você pode inclusive criar uma **árvore de processo**. No MiniTalk, há dois processos em aberto: o *terminal* no lado direito e no lado esquerdo. Há, também, abas de arquivos abertos, abas de navegação... ou seja, um processo pode chamar vários outros processos aí dentro.

Nessa árvore de processo, há concorrência ou espera. Um processo pai que vai gerar um processo filho, eles podem continuar de forma **concorrente** ou pode ter um outro esquema em que **espera** o processo filho a ser executado e aí o processo pai é encerrado junto com o filho.

Um processo também pode ser **independente**, ou seja ele faz as tarefas que foram designadas e não se comunica com nenhum outro. Ou pode ser **interativo**, então dessa maneira ele precisa estabelecer COMUNICAÇÃO entre processos. É aí que entra o pipex e o minitalk.

Essa comunicação se chama de **IPC** (Interprocess Communication). Essa comunicação entre processos cooperativos, podem fazer através de memória compartilhada (um processo pai compartilha memória informações com os processos e o filho acessa essas informações, em vez do processo filho ter uma memória particular individual). Outra forma é você fazer a **transmissão de mensagens** e, para isso, você precisa da assistência do Kernel.

Uma maneira de fazer isso, é pela transmissão de sinais.

# BEHAVIOUR

client



MY PID IS  
7509



server

HOW CAN I SEND  
YOU A MESSAGE?



I LIKE BITWISE  
OPERATIONS



SENDING  
MSG BIT BY BIT



DISPLAYS  
A MESSAGE

Um **PID (Process Identifier)** é um número de identificação (como um RG) que o sistema dá a cada processo. Para cada novo processo (através da getpid()), um novo número deve ser atribuído, ou seja, não se pode ter um único PID para dois ou mais processos ao mesmo tempo.

./server

chamada - getpid()

49 53 50 50 55 50

valor em int (ASCII)

\ 5 2 2 7 2 \0

conversão de int para char

PID: 15227

printf()impressão de arg[1]

Mas eu não tenho como mandar dados através de sinais, a estratégia é mandar uma mensagem **decodificada em bits** (em 0 e 1).

Então, se eu quiser mandar uma frase que contém caracteres que são char, e 1 char tem 1 byte, e 1 byte tem 8 bits, vou

precisar pegar cada caractere dessa frase e mandar os bits de

cada um deles. No meu servidor, vou ter as funções que vão receber os sinais (handler) e vou trabalhar com 2 sinais específicos

que podem ser manipulados por nós:

**SIGUSR1 e SIGURS2.**

Há mais de 30 / 40 opções de sinais, que foram pré-definidas pelo Linux. **SIGUSR1** e **SIGUSR2** são sinais que permitem que nós possamos acoplar funções do jeito que quisermos. Então quando chamamos um sinal, temos atrelado a ele uma função, que é a função de handler, anexada a um sinal. Quando recebo um sinal, o programa vai automaticamente executar a **função de handler**, que vai **decodificar o sinal** que estou recebendo do cliente.

Segundo: quais  
funções eu posso usar  
no **minitalk** ?

```
sig_t signal(int sig, sig_t func);
```

A facilidade de signal() é ter uma interface simplificada para a facilidade de sigaction(2) mais geral. (...) alguns sinais, em vez disso, fazem parar o processo de recepção dos mesmos, ou são simplesmente descartados se o processo não tiver pedido o contrário. Com exceção dos sinais SIGKILL e SIGSTOP, a função signal() permite que um **sinal seja capturado, ignorado, ou para gerar uma interrupção**. Estes sinais são definidos no header <signal.h>:

	Name	Default Action	Description
1	SIGHUP	terminate process	terminal line hangup
2	SIGINT	terminate process	interrupt program
(...)			
30	SIGUSR1	terminate process	User defined signal 1
31	SIGUSR2	terminate process	User defined signal 2

O argumento do signal() **especifica qual sinal foi recebido**.

O procedimento func permite que um usuário escolha a ação após o recebimento de um sinal.

O sinal manipulado é desbloqueado quando a função retorna e o processo continua de onde parou quando o sinal ocorreu.

Para algumas chamadas de sistema, se um sinal for pego enquanto a chamada está sendo executada e a chamada for terminada prematuramente, a chamada é automaticamente reiniciada.

Qualquer manipulador instalado com signal(3) terá a bandeira SA\_RESTART configurada, o que significa que qualquer chamada de sistema reinicializável não retornará ao receber um sinal.

As chamadas de sistema afetadas incluem read(2), write(2), sendto(2), recvfrom(2), sendmsg(2) e recvmsg(2) em um canal de comunicação ou dispositivo de baixa velocidade e durante um ioctl(2) ou wait(2).

Entretanto, as chamadas que já foram comprometidas não são reiniciadas, mas retornam um sucesso parcial (por exemplo, uma breve contagem de leitura). Estas semânticas poderiam ser alteradas com siginterrupt(3).

Veja sigaction(2) para uma lista de funções que são consideradas seguras para uso em manipuladores de sinais.

A ação anterior é devolvida em uma chamada bem sucedida. Caso contrário, o SIG\_ERR é retornado e a variável global errno é definida para indicar o erro.

```
#define sa_handler      __sigaction_u.__sa_handler  
  
#define sa_sigaction  
__sigaction_u.__sa_sigaction  
  
int sigaction(int sig, const struct sigaction  
*restrict act, struct sigaction *restrict  
oact);
```

É mais compatível que o signal (se for trabalhar com diferentes contextos / diferentes sistemas operacionais )

**Um processo pode especificar um handler (manipulador) ao qual um sinal é entregue, ou especificar que um sinal deve ser ignorado.**

**Um processo também pode especificar que uma ação padrão deve ser tomada quando ocorre um sinal.** Um sinal também pode ser bloqueado, caso em que sua entrega seja adiada até que seja desbloqueado.

A ação a ser tomada no momento da entrega é determinada no momento da entrega.

Normalmente, os manipuladores de sinais executam na pilha de corrente do processo.

Isto pode ser mudado, por manipulador, para que os sinais sejam tomados em uma pilha de sinais especial.

A chamada ao sistema sigaction() **atribui uma ação para um sinal especificado pelo sig**.

Se o ato não for zero, ele especifica uma ação (SIG\_DFL, SIG\_IGN, ou uma rotina de manipulador) e a máscara a ser usada ao entregar o sinal especificado.

Se o sinal não for zero, as informações de manuseio anteriores para o sinal são devolvidas ao usuário.

## DESCRIPTION

[top](#)

The **sigaction()** system call is used to change the action taken by a process on receipt of a specific signal. (See [signal\(7\)](#) for an overview of signals.)

*signum* specifies the signal and can be any valid signal except **SIGKILL** and **SIGSTOP**.

If *act* is non-NULL, the new action for signal *signum* is installed from *act*. If *oldact* is non-NULL, the previous action is saved in *oldact*.

The *sigaction* structure is defined as something like:

```
struct sigaction {  
    void      (*sa_handler)(int);  
    void      (*sa_sigaction)(int, siginfo_t *, void *);  
    sigset_t  sa_mask;  
    int       sa_flags;  
    void      (*sa_restorer)(void);  
};
```

As opções podem ser especificadas através da definição de **sa\_flags**. O significado dos vários bits é o seguinte:

(...)

SA_NODEFER	If this bit is set, further occurrences of the delivered signal are not masked during the execution of the handler.
SA_RESETHAND	If this bit is set, the handler is reset back to SIG_DFL at the moment the signal is delivered.
SA_RESTART	See paragraph below.
SA_SIGINFO	If this bit is set, the handler function is assumed to be pointed to by the <code>sa_sigaction</code> member of struct <code>sigaction</code> and should match the prototype shown above or as below in EXAMPLES. This bit should not be set when assigning SIG_DFL or SIG_IGN.

Se um sinal for capturado durante as chamadas do sistema listadas, a chamada pode ser forçada a terminar com o erro EINTR, a chamada pode retornar com um dado de transferência mais curta do que a solicitada, ou a chamada poderá ser reiniciada.

A reinicialização de chamadas pendentes é solicitada definindo o bit SA\_RESTART em sa\_flags.

As chamadas do sistema afetado incluem (...), write(2) e (...) em um canal de comunicação ou em um dispositivo lento (como um terminal, mas não um arquivo regular) e durante uma wait(2) ou (...).

Entretanto, chamadas que já foram comprometidas não são reiniciadas, mas retornam um sucesso parcial (por exemplo, uma breve contagem de leitura).

#### NOTA

O campo **sa\_mask** especificado no act não é permitido bloquear o SIGKILL ou SIGSTOP. Qualquer tentativa de fazer isso será silenciosamente ignorada.

As seguintes funções são ou reentrantes ou não **interrompíveis** por sinais e são **assimétricos** e **seguros**. Portanto, as aplicações podem invocá-las, sem restrições, das funções de captura de sinais:

Interfaces de base:

```
(...), getpid(), (...), kill(), (...), pause(),
(...), sigaction(), sigaddset(), (...),
sigemptyset(), (...), signal(),
(...), sleep(), (...), wait(), (...),
write().
```

Interfaces em tempo real:

```
aio_error(), sigpause(),
aio_return(), aio_suspend(),
sem_post(), sigset() .
```

ANSI C Interfaces:

```
strcpy(), strcat(), strncpy(),
strncat(), and perhaps some
others.
```

Extension Interfaces:

```
strlcpy(), strlcat() .
```

Todas as funções não incluídas nas listas acima são consideradas **inseguras com respeito aos sinais**. Ou seja, o comportamento de tais funções quando chamado de um manipulador de sinais é **indefinido**.

No entanto, em geral, o sinal dos manipuladores devem fazer pouco mais do que colocar uma bandeira; a maioria das outras ações não são seguras.

Além disso, é uma boa prática fazer uma cópia da **variável global errno** e restaurá-la antes de retornar do manipulador de sinais. Isto protege contra o efeito colateral de errno ser definido por funções chamadas de dentro do manipulador de sinais.

# SIGNAL VS SIGACTION

`sigaction()`:

without any of these  
disadvantages

A muscular dogeza dog, a popular internet meme, is shown from the waist up, flexing its well-defined abdominal muscles. An arrow points from the text "without any of these disadvantages" towards this image.

`signal()`:

-does not block other  
signals from arriving

-exact behaviour of  
`signal()` varies between  
systems

-behaviour in  
multi-threaded  
programs is  
undefined



```
int sigemptyset(sigset_t *set);  
int sigaddset(sigset_t *set, int signo);
```

Estas funções **manipulam conjuntos de sinais**, armazenados em um **sigset\_t**.

**sigemptyset()** deve ser chamado para cada objeto do tipo **sigset\_t** antes de qualquer outro uso do objeto.

A função **sigemptyset()** **inicializa um conjunto de sinais a ser vazio**.

A função **sigaddset()** adiciona o sinal “signo” especificado ao conjunto de sinais.

Estas funções são fornecidas como macros no arquivo include <signal.h>.

As funções reais estão disponíveis se seus nomes não estiverem definidos (com #undef name).

As funções **retornam 0**.

macOS 12.4, 4 de junho de 1993 macOS 12.4

Using `kill()` to check the PID server input

## Descrição

"A função `kill()` envia o sinal especificado por `Sig` para `pid`, um processo ou um grupo de processos. Tipicamente, `Sig` será um dos sinais especificados em `sigaction(2)`.

Um valor de **0**, entretanto, fará com que a **verificação de erros seja realizada** (sem que tenha sido enviado nenhum sinal). Isto pode ser usado para **verificar a validade do pid**."

Esta é uma boa descoberta para verificar a validade do PID de uma forma robusta, em vez de apenas verificar se todos os caracteres são dígitos.

Se o pid for > 0:

O `Sig` é enviado para o processo cujo ID é igual a `pid`.

Se o pid == 0:

`Sig` é enviado para todos os processos cujo ID de grupo é igual a `pid` ID de grupo do remetente, e para o qual o processo tem permissão; esta é uma variante do `killpg(2)`.

Se pid é -1:

Se o utilizador tiver privilégios de super-utilizador, o sinal é enviado a todos os processos excluindo os processos do sistema e o processo de envio do sinal.

Se o utilizador não for o super utilizador, o sinal é enviado para todos os processos com o mesmo uid que o utilizador, excluindo o processo enviar o sinal. Nenhum erro é devolvido se algum processo puder ser sinalizado.

```
pid_t getpid() (void);
```

## Descrição

“getpid() retorna o ID do processo de chamada. O ID é garantidamente único e útil para a construção de um temporário arquivo de nomes (file names).”

```
int pause() (void);
```

## Descrição

“A função pause() faz com que **a string de chamada faça uma pausa até que um sinal seja recebido da função kill(2) ou de um timer de intervalo.** (Veja setitimer(2)).

Ao término de um manipulador de sinal iniciado durante um pause(), a chamada de pause() retornará.”

```
int usleep(useconds_t microseconds);
```

## Descrição

**“A função usleep() suspende a execução da thread chamada até microssegundos decorridos ou um sinal é entregue à thread e sua ação é invocar uma função de captura de sinal ou para encerrar o processo.**

As atividades ou limitações do sistema podem prolongar sleep() por uma quantidade indeterminada.

Esta função é implementada utilizando o nanosleep(2) através de uma pausa de microssegundos ou até que ocorra um sinal.

Consequentemente, nesta implementação, o sleep() não tem efeito sobre o estado dos temporizadores do processo, e não há nenhum tratamento especial para o SIGALRM.

Além disso, esta implementação faz não colocar um limite para o valor de microssegundos (além do limitado por o tamanho do tipo useconds\_t); algumas outras plataformas exigem que ele seja menos de um milhão.”

**exit()**

## Descrição

“A função exit() encerra um processo.

Antes de terminar, exit() executa as seguintes funções na ordem listada:

1. Chamar as funções registradas com a função atexit(3), em ordem inversa de seu registro.
2. “Lavar” todos os fluxos de saída abertos.
3. Fechar todos os fluxos abertos.
4. Desbloquear todos os arquivos criados com a função tmpfile(3).

A função torna os oito bits de baixa ordem do argumento de status disponíveis para um processo de pais que chamou uma função wait(2)-family.

A Norma C (ISO/IEC 9899:1999 ("ISO C99")) define os valores 0, EXIT\_SUCCESS, e EXIT\_FAILURE como possíveis valores de status.”

Segundo: como  
esquematizar essa

**comunicação** ?

## PASSOS

**1 |** Abrir o server em loop

**1.1 |** Printar o PID e ficar no aguardo dos sinais

**2 |** Digitar o PID e enviar o/s argumento/s no client

**2.1 |** Traduzir os argumentos em binário

**2.2 |** Enviar os binários para o server por sinais

(correspondentes a 0 e a 1)

**3 |** Receber os sinais no server

**3.1 |** Armazenar os sinais de 8 em 8 e traduzir  
de binário para char de volta (por isso de 8 em 8)

**3.2 |** Armazenar tudo em uma string e imprimir

95	01011111	U+005F	-
96	01100000	U+0060	`
97	01100001	U+0061	a
98	01100010	U+0062	b
99	01100011	U+0063	c
100	01100100	U+0064	d
101	01100101	U+0065	e
102	01100110	U+0066	f
103	01100111	U+0067	g
104	01101000	U+0068	h
105	01101001	U+0069	i
106	01101010	U+006A	j
107	01101011	U+006B	k
108	01101100	U+006C	l
109	01101101	U+006D	m
110	01101110	U+006E	n
111	01101111	U+006F	o
112	01110000	U+0070	p
113	01110001	U+0071	q
114	01110010	U+0072	r
115	01110011	U+0073	s
116	01110100	U+0074	t
117	01110101	U+0075	u
118	01110110	U+0076	v
119	01110111	U+0077	w
120	01111000	U+0078	x
121	01111001	U+0079	y
122	01111010	U+007A	z
123	01111011	U+007B	{

## tabela ASCII

47	00101111	U+002F	/
48	00110000	U+0030	0
49	00110001	U+0031	1
50	00110010	U+0032	2
51	00110011	U+0033	3
52	00110100	U+0034	4
53	00110101	U+0035	5
54	00110110	U+0036	6
55	00110111	U+0037	7
56	00111000	U+0038	8
57	00111001	U+0039	9
58	00111010	U+003A	:

man signal

Signal	x86/ARM most others	Alpha/ SPARC	MIPS	PARISC	Notes
SIGHUP	1	1	1	1	
SIGINT	2	2	2	2	
SIGQUIT	3	3	3	3	
SIGILL	4	4	4	4	
SIGTRAP	5	5	5	5	
SIGABRT	6	6	6	6	
SIGIOT	6	6	6	6	
SIGBUS	7	10	10	10	
SIGEMT	-	7	7	-	
SIGFPE	8	8	8	8	
SIGKILL	9	9	9	9	
SIGUSR1	10	30	16	16	
SIGSEGV	11	11	11	11	
SIGUSR2	12	31	17	17	
SIGPIPE	13	13	13	13	
SIGALRM	14	14	14	14	
SIGTERM	15	15	15	15	
SIGSTKFLT	16	-	-	7	
SIGCHLD	17	20	18	18	
SIGCLD	-	-	18	-	
SIGCONT	18	19	25	26	
SIGSTOP	19	17	23	24	
SIGTSTP	20	18	24	25	
SIGTTIN	21	21	26	27	
SIGTTOU	22	22	27	28	
SIGURG	23	16	21	29	
SIGXCPU	24	24	30	12	
SIGXFSZ	25	25	31	30	
SIGVTALRM	26	26	28	20	
SIGPROF	27	27	29	21	
SIGWINCH	28	28	20	23	
SIGIO	29	23	22	22	
SIGPOLL					Same as SIGIO
SIGPWR	30	29/-	19	19	
SIGINFO	-	29/-	-	-	
SIGLOST	-	-/29	-	-	
SIGSYS	31	12	12	31	

./server

PID: 15227

./client 15227 oi

1 5 2 2 7 2 \0  
0 0 0 0 0 0 0  
1 1 1 1 1 1 0  
1 1 1 1 1 1 0  
0 0 0 0 0 0 0  
0 1 0 1 1 0 0  
1 0 1 1 1 1 0  
1 1 0 0 0 0 0

SIGUSR1  
SIGUSR2

0 0 0 0 0 0 0  
0 0 0 0 0 0 0  
1 1 1 1 1 1 0  
1 0 0 0 0 0 0  
0 1 0 1 1 0 0  
1 0 1 1 1 1 0  
0 0 0 0 0 0 0  
0 1 1 1 1 1 0

152272\0

152272\0

1	00110001	7	00110101
5	00110101	2	00110111
2	00110101	\0	00001010
2	00110111		
o	00110101		
i	00110111		
\0	00001010		

oi\0

oi\0

`./server`

chamada

49 53 50 50 55 50

valor em int (ASCII)

1 5 2 2 7 2 \0

conversão de int para char

PID: 15227

impressão de arg[1]

`./client 15227 oi`

leitura do  
arg[2]

1 5 2 2 7 2 \0

o i \0

conversão de char para int

conversão de char para int

49 53 50 50 55 50

79 73 0

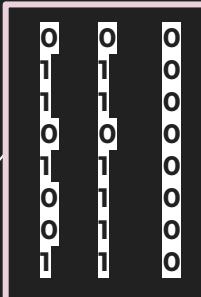
verificação do pid



usleep()

para a  
transmissão  
de

SIGUSR1  
SIGUSR2



00110101  
00110111  
00001010

conversão de binário para int

79 73 0

PID: 15227

oi

impressão de arg[2]

oi\0 —— char para string ——

o i \0

conversão de int para char

`./server`

chamada - getpid()

`49 53 50 50 55 50`

valor em int (ASCII)

`1 5 2 2 7 2 \0`

conversão de int para char

`PID: 15227`

printf() impressão de arg[1]

`./client 15227 oi`

leitura do  
arg[2]

`1 5 2 2 7 2 \0`

conversão de char para int

`49 53 50 50 55 50`

verificação do pid



`o i \0`

conversão de char para binário

0	0	0
1	1	0
1	1	0
0	1	0
1	1	0
0	0	0
0	0	0
1	1	1

`usleep()`

para a  
transmissão  
de

`SIGUSR1`  
`SIGUSR2`

0	0	0
1	1	1
1	1	0
0	0	0
1	1	1
0	0	0
0	0	0
1	1	1

armazena de 8 em 8 bits

`00110101`

`00110111`

`00001010`

conversão de binário para char

`o i \0`

`PID: 15227`

`oi`

impressão de arg[2]

`oi\0`

char para string

analisando **server**  
após um bom **esquema** :D

`./server`

chamada - `getpid()`

49 53 50 50 55 50

valor em int (ASCII)

1 5 2 2 7 2 \0

`atoi()` - conversão de int para char

PID: 15227

`printf()` impressão de arg[1]

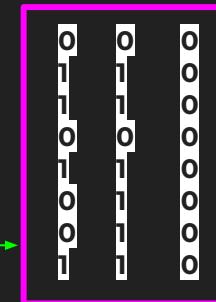
`usleep()`

para a  
transmissão  
de

SIGUSR1  
SIGUSR2

`client.c`

`kill()`  
envia 1  
bit por  
vez



armazena de 8 em 8 bits

00110101

00110111

00001010

conversão de binário para char

o i \0

PID: 15227

oi\0

impressão de arg[2]

oi\0

char para string

# server.c

```
int main(void)
{
    struct sigaction s_server;
    sigset(SIGALRM, sa_mask);

    write(1, "MiniTALK-[FGASTAL-]\n", 20);
    write(1, "Server PID: ", 12);
    ft_putstr(getpid()); 1
    write(1, "\n", 1);
    sigemptyset(&sa_mask); 2
    s_server.sa_flags = SA_SIGINFO | SA_RESTART;
    s_server.sa_mask = sa_mask;
    s_server.sa_sigaction = server_handler; 4
    sigaction(SIGUSR1, &s_server, 0);
    sigaction(SIGUSR2, &s_server, 0); 5
    while (1)
        pause();
    return (0);
}
```

```
int sigemptyset(sigset_t *set); 2
```

```
int sigaction(int sig, const struct
sigaction *restrict act, struct
sigaction *restrict oact); 5
```

```
void ft_putstr(unsigned int n) 1
{
    if (n >= 10)
        ft_putstr((n / 10));
    write(1, &"0123456789"[n % 10], 1);
}
```

```
void server_handler(int sig, siginfo_t *siginfo, void *unused)
{
    static char str = "\0";
    static int i = 0;
    static char c = '\0';
    static pid_t pid = 0;

    (void)unused;
    if (pid)
        c <= 1;
    if (!pid)
        pid = siginfo->si_pid;
    c |= (sig == SIGUSR1);
    if (++i == 8)
    {
        i = 0;
        if (c == 0)
        {
            str = print_free(str);
            pid = 0;
            return ;
        }
        str = add_char(str, c);
        kill(pid, SIGUSR1);
    }
    else
        kill(pid, SIGUSR2);
}
```

SA\_RESTART

See paragraph below.

3

SA\_SIGINFO

If this bit is set, the handler function is assumed to be pointed to by the `sa_sigaction` member of `struct sigaction` and should match the prototype shown above or as below in EXAMPLES. This bit should not be set when assigning `SIG_DFL` or `SIG_IGN`.

# 1 | usando getpid()

```
void ft_putstr(unsigned int n)
{
    if (n >= 10)
        ft_putstr((n / 10));
    write(1, &"0123456789"[n % 10], 1);
}
```

```
int main(void)
{
    struct sigaction s_server;
    sigset(SIGPOLLIN, sa_mask);

    write(1, "Minitalk [FGASTAL-]\n", 20);
    write(1, "Server PID: ", 12);
    ft_putstr(getpid()); 1
    write(1, "\n", 1);
    sigemptyset(&sa_mask);
    s_server.sa_flags = SA_SIGINFO | SA_RESTART;
    s_server.sa_mask = sa_mask;
    s_server.sa_sigaction = server_handler;
    sigaction(SIGUSR1, &s_server, 0);
    sigaction(SIGUSR2, &s_server, 0);
    while (1)
        pause();
    return (0);
}
```

	./client	pid	“string”
arg <b>counter</b>	1	2	3
arg <b>value</b>	[0]	[1]	[2] [1] [++i]

```
pid_t getpid();
```

## DESCRÍÇÃO

“getpid() retorna o ID do processo de chamada. O ID é garantidamente único e útil para a construção de um temporário arquivo de nomes (file names).”

## 2 | usando sigemptyset()

```
int sigemptyset(sigset_t *set);
```

1

Estas funções **manipulam conjuntos de sinais**, armazenados em um **sigset\_t**.

sigemptyset() **deve ser chamado para cada objeto do tipo sigset\_t** antes de qualquer outro uso do objeto. A função sigemptyset() **inicializa um conjunto de sinais a ser vazio**.

Estas funções são fornecidas como macros no arquivo include <signal.h> e **retornam 0**.

```
int main(void)
{
    struct sigaction s_server;
    sigset_t sa_mask;

    write(1, "MiniTalk [FGASTAL-]\n", 20);
    write(1, "Server PID: ", 12);
    ft_putnbr(getpid());
    write(1, "\n", 1);
    sigemptyset(&sa_mask);
    s_server.sa_flags = SA_SIGINFO | SA_RESTART;
    s_server.sa_mask = sa_mask;
    s_server.sa_sigaction = server_handler;
    sigaction(SIGUSR1, &s_server, 0);
    sigaction(SIGUSR2, &s_server, 0);
    while (1)
        pause();
    return (0);
}
```

## 3 | usando flags

### SA\_RESTART

Provide behavior compatible with BSD signal semantics by making certain system calls restartable across signals. This flag is meaningful only when establishing a signal handler. See [signal\(7\)](#) for a discussion of system call restarting.

### SA\_SIGINFO (since Linux 2.2)

The signal handler takes three arguments, not one. In this case, *sa\_sigaction* should be set instead of *sa\_handler*. This flag is meaningful only when establishing a signal handler.

```
int main(void)
{
    struct sigaction s_server;
    sigset(SIGSTOP, sa_mask);

    write(1, "MiniTalk [FGASTAL-]\n", 20);
    write(1, "Server PID: ", 12);
    ft_putnbr(getpid());
    write(1, "\n", 1);
    sigemptyset(&sa_mask);
    s_server.sa_flags = SA_SIGINFO | SA_RESTART;
    s_server.sa_mask = sa_mask;
    s_server.sa_sigaction = server_handler;
    sigaction(SIGUSR1, &s_server, 0);
    sigaction(SIGUSR2, &s_server, 0);
    while (1)
        pause();
    return (0);
}
```

#### 24.3.5 Flags for `sigaction`

The `sa_flags` member of the `sigaction` structure is a catch-all for special features. Most of the time, `SA_RESTART` is a good value to use for this field.

The value of `sa_flags` is interpreted as a bit mask. Thus, you should choose the flags you want to set, OR those flags together, and store the result in the `sa_flags` member of your `sigaction` structure.

Each signal number has its own set of flags. Each call to `sigaction` affects one particular signal number, and the flags that you specify apply only to that particular signal.

In the GNU C Library, establishing a handler with `signal` sets all the flags to zero except for `SA_RESTART`, whose value depends on the settings you have made with `siginterrupt`. See [Primitives Interrupted by Signals](#), to see what this is about.

These macros are defined in the header file `signal.h`.

Macro: `int SA_NOCLDSTOP`

This flag is meaningful only for the `SIGCHLD` signal. When the flag is set, the system delivers the signal for a terminated child process but not for one that is stopped. By default, `SIGCHLD` is delivered for both terminated children and stopped children.

Setting this flag for a signal other than `SIGCHLD` has no effect.

Macro: `int SA_ONSTACK`

If this flag is set for a particular signal number, the system uses the signal stack when delivering that kind of signal. See [Using a Separate Signal Stack](#). If a signal with this flag arrives and you have not set a signal stack, the normal user stack is used instead, as if the flag had not been set.

Macro: `int SA_RESTART`

This flag controls what happens when a signal is delivered during certain primitives (such as `open`, `read` or `write`), and the signal handler returns normally. There are two alternatives: the library function can resume, or it can return failure with error code `EINTR`.

The choice is controlled by the `SA_RESTART` flag for the particular kind of signal that was delivered. If the flag is set, returning from a handler resumes the library function. If the flag is clear, returning from a handler makes the function fail. See [Primitives Interrupted by Signals](#).

As **bandeiras `sa_flags`**, membro da estrutura de `sigaction`, são um elemento de **captura para características especiais**. O valor das `sa_flags` é interpretado como uma **pequena máscara**. Assim, você deve escolher as bandeiras que deseja colocar, OU essas bandeiras juntas, e armazenar o resultado no membro `sa_flags` de sua estrutura de `sigaction`. **SA\_RESTART**: esta bandeira controla o que acontece quando um sinal é entregue durante certos primitivos (como abrir, ler ou escrever), e o manipulador do sinal retorna normalmente. Há duas alternativas: a função de biblioteca pode ser retomada, ou pode retornar falha com o código de erro `EINTR`.

## 4 | usando máscaras

*sa\_mask* specifies a mask of signals which should be blocked (i.e., added to the signal mask of the thread in which the signal handler is invoked) during execution of the signal handler. In addition, the signal which triggered the handler will be blocked, unless the **SA\_NODEFER** flag is used.

*sa\_flags* specifies a set of flags which modify the behavior of the signal. It is formed by the bitwise OR of zero or more of the following:

```
int main(void)
{
    struct sigaction s_server;
    sigset(SIGUSR1, &s_server);
    write(1, "MiniTalk [FGASTAL-]\n", 20);
    write(1, "Server PID: ", 12);
    ft_putnbr(getpid());
    write(1, "\n", 1);
    sigemptyset(&s_server.sa_mask);
    s_server.sa_flags = SA_SIGINFO | SA_RESTART;
    s_server.sa_mask = sa_mask;
    s_server.sa_sigaction = server_handler;
    sigaction(SIGUSR1, &s_server, 0);
    sigaction(SIGUSR2, &s_server, 0);
    while (1)
        pause();
    return (0);
}
```

# 4 | manipulando sinais

## server\_handler

```
void server_handler(int sig, siginfo_t *siginfo, void *unused)
{
    static char c = 0;
    static int i = 0;
    static char *str = NULL;
    static pid_t pid = 0;

    (void)unused;
    if (pid)
        c <= 1;
    if (!pid)
        pid = siginfo->si_pid;
    c |= (sig == SIGUSR1);
    if (++i == 8)
    {
        i = 0;
        if (c == 0)
        {
            str = print_free(str);
            pid = 0;
            return ;
        }
        str = add_char(str, c);
        kill(pid, SIGUSR1); 4
    }
    else
        kill(pid, SIGUSR2);
}
```

Using `kill()` to check the PID server input

A função `kill()` envia o sinal especificado por `Sig` para `pid`, um processo ou um grupo de processos. Tipicamente, `Sig` será um dos sinais especificados em `sigaction(2)`.

Um valor de **0**, entretanto, fará com que a **verificação de erros seja realizada** (sem que tenha sido enviado nenhum sinal). Isto pode ser usado para **verificar o validade do pid.**

```
int main(void)
{
    struct sigaction s_server;
    sigset(SIGALRM, sa_mask);

    write(1, "MiniTalk-[FGASTAL-]\n", 20);
    write(1, "Server-PID: ", 12);
    ft_putstr(getpid());
    write(1, "\n", 1);
    sigemptyset(&sa_mask);
    s_server.sa_flags = SA_SIGINFO | SA_RESTART;
    s_server.sa_mask = sa_mask;
    s_server.sa_sigaction = server_handler; 1
    sigaction(SIGUSR1, &s_server, 0);
    sigaction(SIGUSR2, &s_server, 0);
    while (1)
        pause();
    return (0);
}
```

```
static char *add_char(char *str, char c)
{
    char *new;
    int i;

    if (!str)
        return (start_str(c));
    new = malloc((ft_strlen(str) + 2));
    if (!new)
    {
        free(str);
        return (NULL);
    }
    i = -1;
    while (str[++i])
        new[i] = str[i];
    free(str);
    new[i+1] = c;
    new[i+2] = '\0';
    c = 0;
    return (new);
}
```

```
char *print_free(char *str)
{
    write(1, str, ft_strlen(str));
    free(str);
    return (NULL);
}
```

The `siginfo_t` argument to a `SA_SIGINFO` handler

When the `SA_SIGINFO` flag is specified in `act.sa_flags`, the signal handler address is passed via the `act.sa_sigaction` field. This handler takes three arguments, as follows:

```
void  
handler(int sig, siginfo_t *info, void *ucontext)  
{  
    ...  
}
```

These three arguments are as follows

`sig` The number of the signal that caused invocation of the handler.

`info` A pointer to a `siginfo_t`, which is a structure containing further information about the signal, as described below.

`ucontext` This is a pointer to a `ucontext_t` structure, cast to `void *`. The structure pointed to by this field contains signal context information that was saved on the user-space stack by the kernel; for details, see `sigreturn(2)`. Further information about the `ucontext_t` structure can be found in `getcontext(3)` and `signal(7)`. Commonly, the handler function doesn't make any use of the third argument.

C server.c X

minitalk-ultima-vez > sources > C server.c

```
37
38 void server_handler(int sig, siginfo_t *siginfo, void *unused)
39 {
40     static char c = 0;
41     static int i = 0;
42     static char *str = NULL;
43     static pid_t pid = 0;
44
45     (void)unused;
46     if (pid)
47         c <<= 1;
48     if (!pid)
49         pid = siginfo->si_pid;
50     c |= (sig == SIGUSR1);
51     if (++i == 8)
52     {
53         i = 0;
54         if (c == 0)
55         {
56             pid = finalizing_sending_string(str, pid);
57             return;
58         }
59         str = adding_char(str, c);
60         c = 0;
61         kill_pid(pid, 1);
62     }
63     else
64         kill_pid(pid, 2);
```

Preciso de **variáveis estáticas** porque essa função vai ser chamada para codificar a cada bit, então é necessário salvar o valor dos bits que já foram codificados anteriormente, para não perder a função toda vez que a função terminar de ser executada.

Contexto do tipo void vai conter **outras informações a respeito do sistema**. Deixa void mesmo.

\* Signals sent with kill(2) and sigqueue(3) fill in **si\_pid** and **si\_uid**. In addition, signals sent with sigqueue(3) fill in **si\_int** and **si\_ptr** with the values specified by the sender of the signal; see sigqueue(3) for more details.

Quando mando a mensagem, preciso saber qual processo enviou mensagem pro cliente (em info->si\_pid), quero saber qual sinal me mandou essa informação. Aí pego a variável si\_pid da estrutura info.

Comparação de bit, considerando a formação da letra, vou comparando com 1 ou 0. Pego uma variável de controle, forço que a informação seja sempre 1 para aquela posição e se você conseguir identificar que é 1, manda sig2, se for 0, manda sig1 e aí vai mandando através do sinal. 8 bits fechado > avalia o byte > se for \0 aí finaliza com a frase de sucesso.

The `siginfo_t` data type is a structure with the following fields:

```
siginfo_t {  
    int      si_signo;      /* Signal number */  
    int      si_errno;      /* An errno value */  
    int      si_code;       /* Signal code */  
    int      si_trapno;     /* Trap number that caused  
                           hardware-generated signal  
                           (unused on most architectures) */  
    pid_t    si_pid;        /* Sending process ID */  
    uid_t    si_uid;        /* Real user ID of sending process */  
    int      si_status;     /* Exit value or signal */  
    clock_t  si_utime;     /* User time consumed */  
    clock_t  si_stime;     /* System time consumed */  
    union sigval si_value; /* Signal value */  
    int      si_int;        /* POSIX.1b signal */  
    void    *si_ptr;        /* POSIX.1b signal */  
    int      si_overrun;    /* Timer overrun count;  
                           POSIX.1b timers */  
    int      si_timerid;    /* Timer ID; POSIX.1b timers */  
    void    *si_addr;        /* Memory location which caused fault */  
    long    si_band;        /* Band event (was int in  
                           glibc 2.3.2 and earlier) */  
    int      si_fd;          /* File descriptor */  
    short   si_addr_lsb;    /* Least significant bit of address  
                           (since Linux 2.6.32) */  
    void    *si_lower;       /* Lower bound when address violation  
                           occurred (since Linux 3.19) */  
    void    *si_upper;       /* Upper bound when address violation  
                           occurred (since Linux 3.19) */  
    int      si_pkey;        /* Protection key on PTE that caused  
                           fault (since Linux 4.6) */  
    void    *si_call_addr;   /* Address of system call instruction  
                           (since Linux 3.5) */  
    int      si_syscall;     /* Number of attempted system call  
                           (since Linux 3.5) */  
    unsigned int si_arch;    /* Architecture of attempted system call  
                           (since Linux 3.5) */  
}
```

# 5 | usando sigaction

```
int sigaction(int sig, const struct sigaction  
*restrict act, struct sigaction *restrict oact);
```

5

A chamada ao sistema sigaction()

atribui uma ação para um sinal  
especificado pelo sig.

**Um processo pode especificar um**  
**handler (manipulador) ao qual um sinal**  
**é entregue, ou especificar que um sinal**  
**deve ser ignorado.**

**Um processo também pode**  
**especificar que uma ação**  
**padrão deve ser tomada quando**  
**ocorre um sinal.**

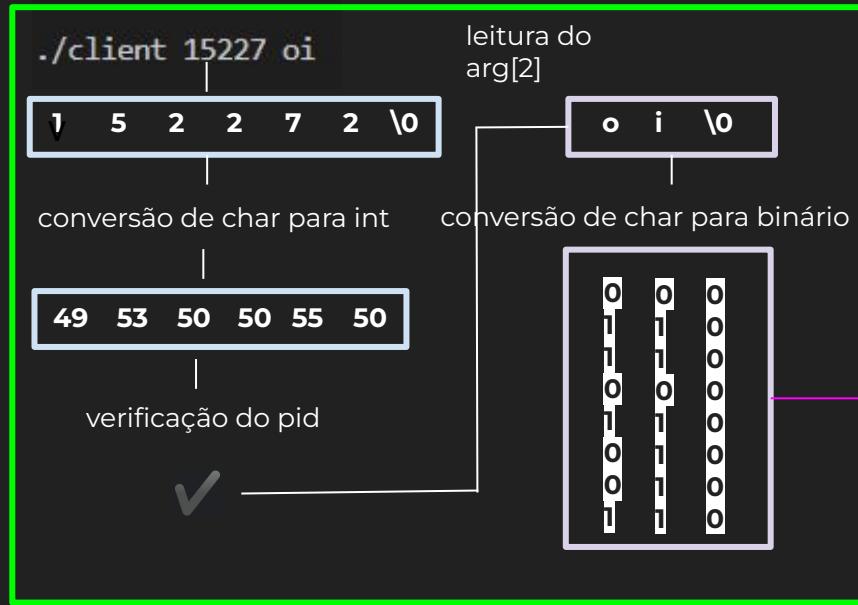
**sigaction()**  
returns 0 on  
success; on  
error, -1 is  
returned, and  
*errno* is set to  
indicate the  
error.

```
int main(void)  
{  
    struct sigaction s_server;  
    sigset(SIGUSR1, &s_server);  
    write(1, "MiniTalk [FGASTAL-]\n", 20);  
    write(1, "Server PID: ", 12);  
    ft_putnbr(getpid());  
    write(1, "\n", 1);  
    sigemptyset(&s_server.sa_mask);  
    s_server.sa_flags = SA_SIGINFO | SA_RESTART;  
    s_server.sa_mask = sa_mask;  
    s_server.sa_sigaction = server_handler;  
    sigaction(SIGUSR1, &s_server, 0);  
    sigaction(SIGUSR2, &s_server, 0);  
    while (1)  
        pause();  
    return (0);  
}
```

1

analizando **client**  
após um bom **esquema** :D

**client.c**



```

int ft_verify_input(int ac, char **av)
{
    int i;
    if (ac != 3)
        return (1);
    while (av[1][++i])
    {
        if (ft_isdigit(av[1][i]) == 0)
            return (1);
    }
    return (0);
}

```

```

int main(int argc, char **argv)
{
    struct sigaction s_client;
    sigset(SIGPOLLIN, mask);
    if (ft_verify_input(argc, argv)) 1
    {
        write(1, "\n>>> ERROR | INVALID INPUT <<<\n\n", 32);
        return (1);
    }
    sigemptyset(&mask);
    s_client.sa_mask = mask;
    s_client.sa_flags = SA_SIGINFO | SA_RESTART;
    s_client.sa_sigaction = client_handler; 2
    sigaction(SIGUSR1, &s_client, NULL);
    sigaction(SIGUSR2, &s_client, NULL);
    if (!ft_strlen(argv[2]))
        exit(0);
    sending_bits(ft_atoi(argv[1]), argv[2]); 3
    write(1, "\n", 1);
    while (1)
        pause();
    return (0);
}

```

```

void client_handler(int sig, siginfo_t *siginfo, void *unused)
{
    static int bytes = 0;
    (void)siginfo;
    (void)unused;
    if (sig == SIGUSR1)
    {
        write(1, "\nNumber of bytes: ", 19);
        ft_putstr(++bytes);
    }
    sending_bits(0, NULL); 3
}

```

```

void end_of_string_message(pid_t static_pid) 4
{
    int i;
    i = 0;
    while (i < 8)
    {
        usleep(WAIT_TIME);
        kill(static_pid, SIGUSR2);
        i++;
    }
    write(1, "\n>>> MESSAGE WAS SUCCESSFULLY SENT! : ) <<<\n\n", 44);
    exit(0);
}

```

```

void sending_bits(pid_t pid, char *s)
{
    static int i = 8;
    static char c;
    static char *str;
    static pid_t static_pid;
    if (s)
    {
        str = s;
        static_pid = pid;
        c = *str;
    }
    if (!i)
    {
        i = 8;
        c = *(++str);
        if (!c)
            end_of_string_message(static_pid); 3
    }
    if (c && c > -i & 1)
        kill(static_pid, SIGUSR1);
    else_if (c)
        kill(static_pid, SIGUSR2);
}

```

client.c

```

int ft_verify_input(int ac, char **av)
{
    int i;
    i = -1;
    if (ac != 3)
        return (1);
    while (av[1][++i])
    {
        if (ft_isdigit(av[1][i]) == 0)
            return (1);
    }
    return (0);
}

```

```

int ft_isdigit(int c)
{
    return (c >= '0' && c <= '9');
}

```

1

## 1 | verificação do input

	./client	pid	“string”	
arg counter	1	2	3	→
arg value	[0]	[1]	[2]	
		[1] [++i]		<pre> if (ft_verify_input(ac, av)) {     write(1, "Error: invalid input\n", 21);     return (1); } </pre>

se tiver um número != de 3 argumentos > erro > retorna (1)  
se o pid for número mas for == 0 > erro > retorna (1)

se nada disso > sucesso > retorna (0)

função que recebe o sinal, conta o número de bytes (conjunto de 8 bits) e envia os bits para o server (mensagem enviada)

```
void client_handler(int sig, siginfo_t *siginfo, void *unused)
{
    static int bytes = 0;

    (void)siginfo;
    (void)unused;
    if (sig == SIGUSR1)
    {
        write(1, "\nNumber of bytes: ", 19);
        ft_putstr(++bytes); 1
    }
    sending_bits(0, NULL); 2
}
```

1

```
void ft_putstr(unsigned int n)
{
    if (n >= 10)
        ft_putstr((n / 10));
    write(1, &"0123456789"[n % 10], 1);
}
```

função put unsigned number que recebe um inteiro que é a **quantidade de bytes** presentes na string enviada que será “printada” no terminal  
“number of bytes:” **“16”**

2

```
void sending_bits(pid_t pid, char *s)
{
    static int i = 8;
    static char c;
    static char *str;
    static pid_t static_pid;

    if (s)
    {
        str = s;
        static_pid = pid;
        c = *str;
    }
    if (!i)
    {
        i = 8;
        c = *(++str);
        if (!c)
            end_of_string_message(static_pid); 3
    }
    if (c && c >>--i & 1)
        kill(static_pid, SIGUSR1);
    else if (c)
        kill(static_pid, SIGUSR2);
}
```

## 2 | manipulação do sinal com client\_handler

```
s_client.sa_sigaction = client_handler;
SIG_BLOCK(SIGUSR1);

./client 208 "oi"
number of bytes: 2
finished successfully
```

3

```
void end_of_string_message(pid_t static_pid)
{
    int i;

    i = 0;
    while (i < 8)
    {
        usleep(WAIT_TIME);
        kill(static_pid, SIGUSR2);
        i++;
    }
    write(1, "\n>>> MESSAGE WAS SUCCESSFULLY SENT! :)<<<\n\n", 44);
    exit(0);
}
```

Trabalho com todos os 8 bits que estão disponíveis pro meu caractere para poder fazer o bônus. Se eu trabalho com 8 bits e não com 7, então estou trabalhando com o caractere sem sinal, ou seja de **0 a 255**.

Estou trabalhando com uma codificação **UTF-8**, que é o sistema de codificação de caracteres abrangentes para linguagens do mundo ocidental, então consigo representar a ASCII, os caracteres especiais, cedilha, acento etc.

character	encoding	bits
A	UTF-8	01000001
A	UTF-16	00000000 01000001
A	UTF-32	00000000 00000000 00000000 01000001
あ	UTF-8	11100011 10000001 10000010
あ	UTF-16	00110000 01000010
あ	UTF-32	00000000 00000000 00110000 01000010

```

void sending_bits(pid_t pid, char *s)    2
{
    static int i = 8;
    static char c;
    static char *str;
    static pid_t static_pid;

    if (s)
    {
        str = s;
        static_pid = pid;
        c = *str;
    }
    if (!i)
    {
        i = 8;
        c = *(++str);
        if (!c)
            end_of_string_message(static_pid);    3
    }
    if (c && c >> --i & 1)
        kill(static_pid, SIGUSR1);
    else if (c)
        kill(static_pid, SIGUSR2);
}

```

retorna valor  
em ASCII do pid

pid

string

```
send_bit(ft_atoi(av[1]), av[2]);
```

```
int ft_atoi(const char *str)
{
    int sinal;
    int num;
    int i;

    sinal = 1;
    num = 0;
    i = 0;
    while ((str[i] >= 9 && str[i] <= 13) || str[i] == 32)
        i++;
    if (str[i] == '+' || str[i] == '-')
    {
        if (str[i] == '-')
            sinal *= -1;
        i++;
    }
    while (str[i] >= 48 && str[i] <= 57)
        num = (num * 10) + (str[i++]-48);
    return (sinal * num);
}
```

### 3 | conversão de char para int no pid (argv[1]) e na string argv[2])

```
void end_prog(pid_t s_pid)
{
    int i;

    i = 8;
    while (i--)
    {
        usleep(50);
        kill(s_pid, SIGUSR2);
    }
    write(1, "\nfinished successfully\n", 22);
    exit(0);
}
```

client.c

```
void send_bit(pid_t pid, char *s)
{
    static int i = 8;
    static char c;
    static char *str;
    static pid_t s_pid;

    if (s)
    {
        str = s;
        s_pid = pid;
        c = *str;
    }
    if (!i)
    {
        i = 8;
        c = *(++str);
        if (!c)
            end_prog(s_pid);
    }
    if (c >--i & 1)
        kill(s_pid, SIGUSR1);
    else
        kill(s_pid, SIGUSR2);
}
```

## 4 | pause()

```
while(1)  
→    pause();
```

client.c

```
int pause(void);
```

### Descrição

“A função pause() faz com que **a string de chamada** faça uma pausa até que um sinal seja recebido da função kill ou de um timer de intervalo. (Veja setitimer(2)).

Ao término de um manipulador de sinal iniciado durante um pause(), a chamada de pause() retornará.”

compilação do projeto

**Valgrind** é um software livre que auxilia o trabalho de depuração de programas criado por Julian Seward. Ele possui ferramentas que **detectam erros decorrentes** do uso incorreto da **memória dinâmica**, como por exemplo os vazamentos de memória, alocação e desalocação incorretas e acessos a áreas inválidas.

O diferencial deste programa está no fato de que usa uma máquina virtual para simular o acesso à memória do programa em teste, eliminando a necessidade de uso de outras bibliotecas auxiliares ou mudanças drásticas no código.

```
valgrind --leak-check=full ./a.out
```

TERMINAL

```
ient
coder@fgastal--workspace-5dff7954bc-8j5pt:~/minitalk-final$ valgrind --leak-check=full ./server
==6634== Memcheck, a memory error detector
==6634== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==6634== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==6634== Command: ./server
==6634==
MinTalk [FGASTAL-]
Server PID: 6634
oi^C==6634==
==6634== Process terminating with default action of signal 2 (SIGINT)
==6634==      at 0x493DFC7: pause (pause.c:29)
==6634==      by 0x4014D7: main (server.c:102)
==6634==      HEAP SUMMARY:
==6634==      in use at exit: 0 bytes in 0 blocks
==6634==      total heap usage: 2 allocs, 2 frees, 5 bytes allocated
==6634==      All heap blocks were freed -- no leaks are possible
==6634==      For lists of detected and suppressed errors, rerun with: -s
==6634==      ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

```
coder@fgastal--workspace-5dff7954bc-8j5pt:~/minitalk-final$ valgrind --leak-check=f
ull ./client 6634 oi
==6671== Memcheck, a memory error detector
==6671== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==6671== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==6671== Command: ./client 6634 oi
==6671==

Number of bytes: 1
Number of bytes: 2
>>> MESSAGE WAS SUCCESSFULLY SENT! :) <<<

==6671==      HEAP SUMMARY:
==6671==      in use at exit: 0 bytes in 0 blocks
==6671==      total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==6671==      All heap blocks were freed -- no leaks are possible
==6671==      For lists of detected and suppressed errors, rerun with: -s
==6671==      ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
coder@fgastal--workspace-5dff7954bc-8j5pt:~/minitalk-final$
```

## TERMINAL

```
coder@fgastal--workspace-5dff7954bc-8j5pt:~/minitalk-final$ valgrind --leak-check=full ./server
==6735== Memcheck, a memory error detector
==6735== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
.
==6735== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright
info
==6735== Command: ./server
6735
[FGASTAL-]
Server PID: 6735
c==6735==
==6735== Process terminating with default action of signal 2 (SIGINT)
==6735==    at 0x493DFC7: pause (pause.c:29)
==6735==    by 0x4014D7: main (server.c:102)
==6735== HEAP SUMMARY:
==6735==     in use at exit: 0 bytes in 0 blocks
==6735==     total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==6735== All heap blocks were freed -- no leaks are possible
==6735== For lists of detected and suppressed errors, rerun with: -s
==6735== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

```
coder@fgastal--workspace-5dff7954bc-8j5pt:~/minitalk-final$ valgrind --leak-check=full ./client 6634
==6759== Memcheck, a memory error detector
==6759== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==6759== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==6759== Command: ./client 6634
==6759==

>>> ERROR | INVALID INPUT <<<

==6759== HEAP SUMMARY:
==6759==     in use at exit: 0 bytes in 0 blocks
==6759==     total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==6759== All heap blocks were freed -- no leaks are possible
==6759== For lists of detected and suppressed errors, rerun with: -s
==6759== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
coder@fgastal--workspace-5dff7954bc-8j5pt:~/minitalk-final$
```

Erro de manipulação, por exemplo, **número incorreto de argumentos**

✓ TERMINAL

```
coder@fgastal--workspace-5dff7954bc-8j5pt:~/minitalk-final$ valgrind --leak-check=full ./server
==6800== Memcheck, a memory error detector
==6800== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
.
==6800== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright
info
==6800== Command: ./server
==6800== MiniTalk [FGASTAL-]
Server PID: 6800
^C==6800==
==6800== Process terminating with default action of signal 2 (SIGINT)
==6800== at 0x493DFC7: pause (pause.c:29)
==6800== by 0x4014D7: main (server.c:102)
==6800==
==6800== HEAP SUMMARY:
==6800==   in use at exit: 0 bytes in 0 blocks
==6800==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==6800==
==6800== All heap blocks were freed -- no leaks are possible
==6800==
==6800== For lists of detected and suppressed errors, rerun with: -s
==6800== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

```
coder@fgastal--workspace-5dff7954bc-8j5pt:~/minitalk-final$ valgrind --leak-check=full ./client 6634 "oi"
==6820== Memcheck, a memory error detector
==6820== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==6820== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==6820== Command: ./client 6634 oi
==6820==
^C==6820==
==6820== Process terminating with default action of signal 2 (SIGINT)
==6820== at 0x493DFC7: pause (pause.c:29)
==6820== by 0x4014A6: main (client.c:91)
==6820==
==6820== HEAP SUMMARY:
==6820==   in use at exit: 0 bytes in 0 blocks
==6820==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==6820==
==6820== All heap blocks were freed -- no leaks are possible
==6820==
==6820== For lists of detected and suppressed errors, rerun with: -s
==6820== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

```
coder@fgastal--workspace-5dff7954bc-8j5pt:~/minitalk-final$ []
```

Erro de manipulação, por exemplo, **servidor PID inválido**: é como se estivesse enviando mensagem para outro servidor, então o meu servidor não recebe nada.

✓ TERMINAL

```
==6979== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright  
info  
==6979== Command: ./server  
==6979==
```

```
MiniTALK [FGASTAL-]  
Server PID: 6979
```

```
Lore ipsum is simply dummy text of the printing and typesetting industry. Lore ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book. It has survived not only five centuries, but also the leap into electronic typesetting, remaining essentially unchanged. It was popularised in the 1960s with the release of Letraset sheets containing Lore ipsum passages, and more recently with desktop publishing software like Aldus PageMaker including versions of Lore ipsum.^C==6979==
```

```
==6979== Process terminating with default action of signal 2 (SIGINT)  
==6979== at 0x493DFC7: pause (pause.c:29)  
==6979== by 0x4014D7: main (server.c:102)  
==6979==
```

```
==6979== HEAP SUMMARY:  
==6979==     in use at exit: 0 bytes in 0 blocks  
==6979== total heap usage: 574 allocs, 574 frees, 165,599 bytes allocated  
==6979==  
==6979== All heap blocks were freed -- no leaks are possible  
==6979==  
==6979== For lists of detected and suppressed errors, rerun with: -s  
==6979== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

```
coder@fgastal--workspace-5dff7954bc-8j5pt:~/minitalk-final$ []
```

```
Number of bytes: 557  
Number of bytes: 558  
Number of bytes: 559  
Number of bytes: 560  
Number of bytes: 561  
Number of bytes: 562  
Number of bytes: 563  
Number of bytes: 564  
Number of bytes: 565  
Number of bytes: 566  
Number of bytes: 567  
Number of bytes: 568  
Number of bytes: 569  
Number of bytes: 570  
Number of bytes: 571  
Number of bytes: 572  
Number of bytes: 573  
Number of bytes: 574
```

```
>>> MESSAGE WAS SUCCESSFULLY SENT! :) <<<
```

```
==7010==  
==7010== HEAP SUMMARY:  
==7010==     in use at exit: 0 bytes in 0 blocks  
==7010== total heap usage: 0 allocs, 0 frees, 0 bytes allocated  
==7010==  
==7010== All heap blocks were freed -- no leaks are possible  
==7010==  
==7010== For lists of detected and suppressed errors, rerun with: -s  
==7010== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Envio de **mensagem de grande porte**

usando o site Lore ipsum

## TERMINAL

```
coder@fgastal--workspace-5dff7954bc-f22bg:~/minitalk-final$ valgrind --leak-check=full ./server
==7612== Memcheck, a memory error detector
==7612== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==7612== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==7612== Command: ./server
==7612==
```

**MiniTalk [FGASTAL-]**

PID: 7612

```
oiola^C==7612==
==7612== Process terminating with default action of signal 2 (SIGINT)
==7612== at 0x493DFC7: pause (pause.c:29)
==7612== by 0x401231: main (in /home/coder/minitalk-final/server)
==7612==

==7612== HEAP SUMMARY:
==7612==     in use at exit: 0 bytes in 0 blocks
==7612==   total heap usage: 6 allocs, 6 frees, 23 bytes allocated
==7612==

==7612== All heap blocks were freed -- no leaks are possible
==7612==

==7612== For lists of detected and suppressed errors, rerun with: -s
==7612== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

coder@fgastal--workspace-5dff7954bc-f22bg:~/minitalk-final$
```

Receber uma mensagem de um **2º cliente**  
após o 1º cliente (em uma fila)

```
==7637== Memcheck, a memory error detector
==7637== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==7637== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==7637== Command: ./client 7612 oi
==7637==

Sending charactere
Sending charactere
>>> MESSAGE WAS SUCCESSFULLY SENT! :) <<<

==7637==
==7637== HEAP SUMMARY:
==7637==     in use at exit: 0 bytes in 0 blocks
==7637==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==7637==

==7637== All heap blocks were freed -- no leaks are possible
==7637==

==7637== For lists of detected and suppressed errors, rerun with: -s
==7637== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
coder@fgastal--workspace-5dff7954bc-f22bg:~/minitalk-final$ valgrind --leak-check=full ./client 7612 "ola"
==7668== Memcheck, a memory error detector
==7668== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==7668== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==7668== Command: ./client 7612 ola
==7668==

Sending charactere
Sending charactere
Sending charactere
>>> MESSAGE WAS SUCCESSFULLY SENT! :) <<<

==7668==
==7668== HEAP SUMMARY:
==7668==     in use at exit: 0 bytes in 0 blocks
==7668==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==7668==

==7668== All heap blocks were freed -- no leaks are possible
==7668==

==7668== For lists of detected and suppressed errors, rerun with: -s
==7668== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
coder@fgastal--workspace-5dff7954bc-f22bg:~/minitalk-final$
```

Twitter [5]	Unicode	Bytes (UTF-8)
无障碍图标	U+1F601	\xF0\x9F\x98\x81
哭脸表情	U+1F602	\xF0\x9F\x98\x82
微笑表情	U+1F603	\xF0\x9F\x98\x83
傻笑表情	U+1F604	\xF0\x9F\x98\x84
偷笑表情	U+1F605	\xF0\x9F\x98\x85
大笑表情	U+1F606	\xF0\x9F\x98\x86
眨眼表情	U+1F609	\xF0\x9F\x98\x89
心形表情	U+1F60A	\xF0\x9F\x98\x8A
傻笑表情	U+1F60B	\xF0\x9F\x98\x8B
微笑表情	U+1F60C	\xF0\x9F\x98\x8C
鄙视表情	U+1F60D	\xF0\x9F\x98\x8D
傻笑表情	U+1F60F	\xF0\x9F\x98\x8F
哭脸表情	U+1F612	\xF0\x9F\x98\x92

#### TERMINAL

```
coder@fgastal--workspace-5dff7954bc-8j5pt:~/minitalk-final$ valgrind --leak-check=full ./server
==7075== Memcheck, a memory error detector
==7075== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==7075== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==7075== Command: ./server
==7075==
Minitalk [FGASTAL-]
Server PID: 7075
C^C==7075== Process terminating with default action of signal 2 (SIGINT)
==7075== at 0x493DFC7: pause (pause.c:29)
==7075== by 0x4014D7: main (server.c:102)
==7075==
==7075== HEAP SUMMARY:
==7075==     in use at exit: 0 bytes in 0 blocks
==7075==     total heap usage: 4 allocs, 4 frees, 14 bytes allocated
==7075==
==7075== All heap blocks were freed -- no leaks are possible
==7075==
==7075== For lists of detected and suppressed errors, rerun with: -s
==7075== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

```
coder@fgastal--workspace-5dff7954bc-8j5pt:~/minitalk-final$ valgrind --leak-check=full ./client 7075 ↵
==7100== Memcheck, a memory error detector
==7100== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==7100== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==7100== Command: ./client 7075
==7100==
```

```
Number of bytes: 1
Number of bytes: 2
Number of bytes: 3
Number of bytes: 4
>>> MESSAGE WAS SUCCESSFULLY SENT! :) <<<
```

```
==7100== HEAP SUMMARY:
==7100==     in use at exit: 0 bytes in 0 blocks
==7100==     total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==7100==
==7100== All heap blocks were freed -- no leaks are possible
==7100==
==7100== For lists of detected and suppressed errors, rerun with: -s
==7100== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Suporte de caracteres **unicode!**

O emoji é como se fosse uma série de letras e números, o código converte para binário. :)