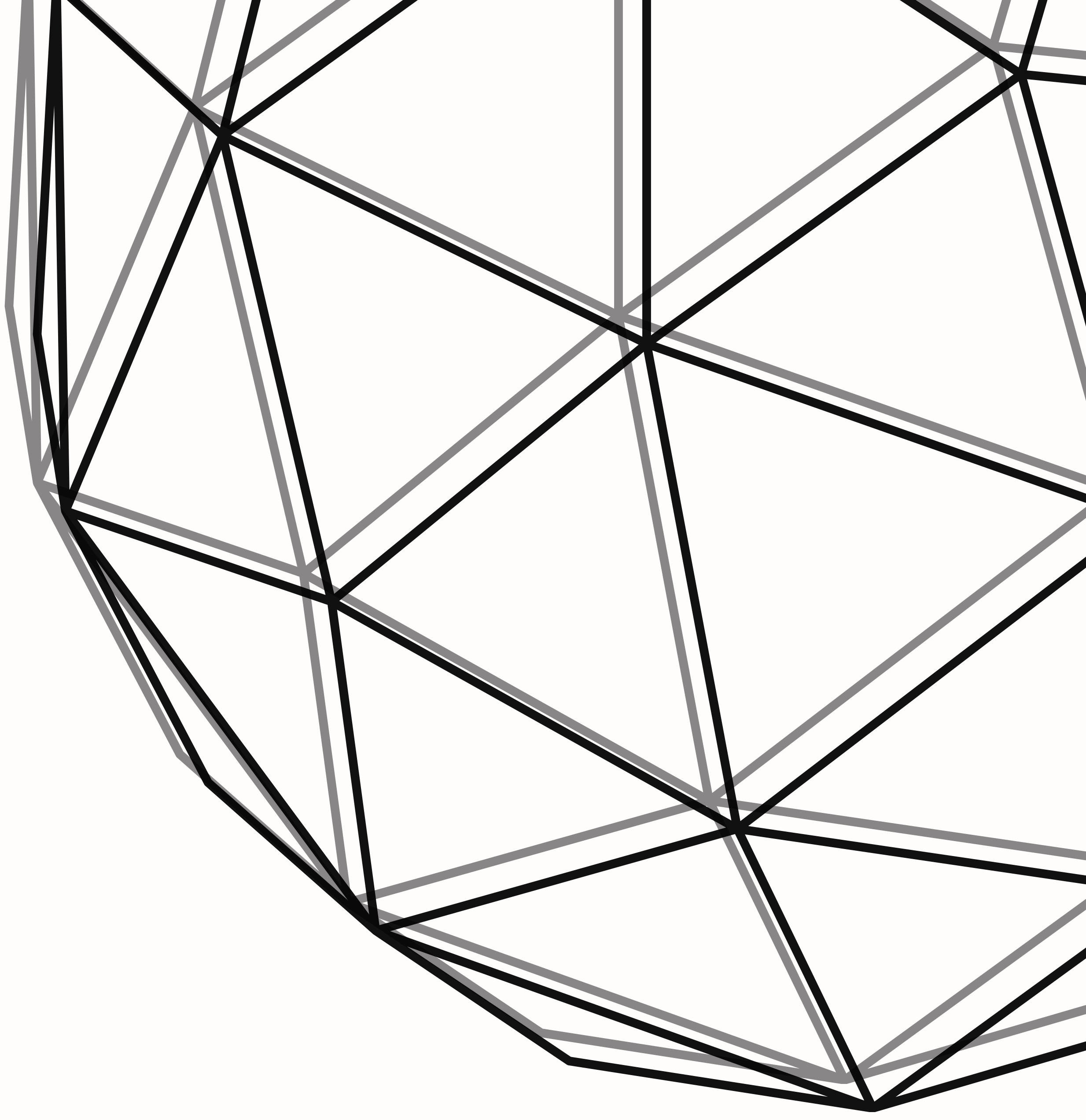
Introduction
to
SIGNALS

MiniTalk

MIKOLAJ JAKOWICKI

Last 3 pages contains my code.
Use it only if it's emergance!



Overviews

The *MiniTalk* project demonstrates inter-process communication using signals in Unix-like operating systems. It consists of a client-server model where:

-Client: *Converts text into binary and sends it to the server using signals.*

-Server: *Listens for signals, reconstructs the binary into text characters, and outputs them.*

Learning Objectives:

-Signal Handling: *Understanding how signals facilitate communication between processes.*

-Binary Representation: *Encoding and decoding text messages into binary format for transmission.*

-Process Identification: *Using Process IDs (PIDs) to identify and communicate between client and server processes.*

-System Calls: *Employing system calls (kill, sigaction, getpid) for signal management and process control.*

Educational Benefits:

-Practical Application: *Applying theoretical knowledge of signals and process management in a real-world project.*

-Hands-on Experience: *Gaining practical experience with low-level programming techniques and Unix system calls.*

-Concept Reinforcement: *Reinforcing understanding of communication protocols and error handling in software development.*

The *MiniTalk* project serves as a foundational exercise in system programming, offering practical insights into process communication and Unix system fundamentals.

Key Elements

Functions:

-**kill**:
Function: *Sends a signal to a process or a group of processes.*
Usage in MiniTalk: *The client uses kill to send SIGUSR1 or SIGUSR2 signals to the server, encoding each bit of the message.*

-**sigaction**:
Function: *Establishes a signal handler for a specific signal.*
Usage in MiniTalk*: The server uses sigaction to set up handlers (sig_user1 and sig_user2) for SIGUSR1 and SIGUSR2 signals respectively. These handlers process incoming signals and reconstruct the transmitted data.*

-**getpid**:
Function: *Retrieves the Process ID (PID) of the calling process.*
Usage in MiniTalk:*The server calls getpid to obtain its own PID.This PID is then typically displayed or communicated to the client, allowing the client to send signals to the correct server process.*

-**sleep and usleep**:
Functions: *Pause execution of a program for a specified amount of time.*
Usage in MiniTalk:*sleep suspends the execution of the program for a specified number of seconds.usleep suspends the execution of the program for a specified number of microseconds (1 microsecond = 1 millionth of a second).In MiniTalk, usleep is used to introduce small delays between sending individual signals to ensure they are processed correctly by the receiving process.*

Elements:

-**Signal Masks**:
Definition: *A set of signals that are temporarily blocked from delivery to a process.*
Usage in MiniTalk: *The sa_mask field in sigaction is used to define a set of signals that should be blocked (masked) while a signal handler is executing. This prevents the handler from being interrupted by certain signals.*

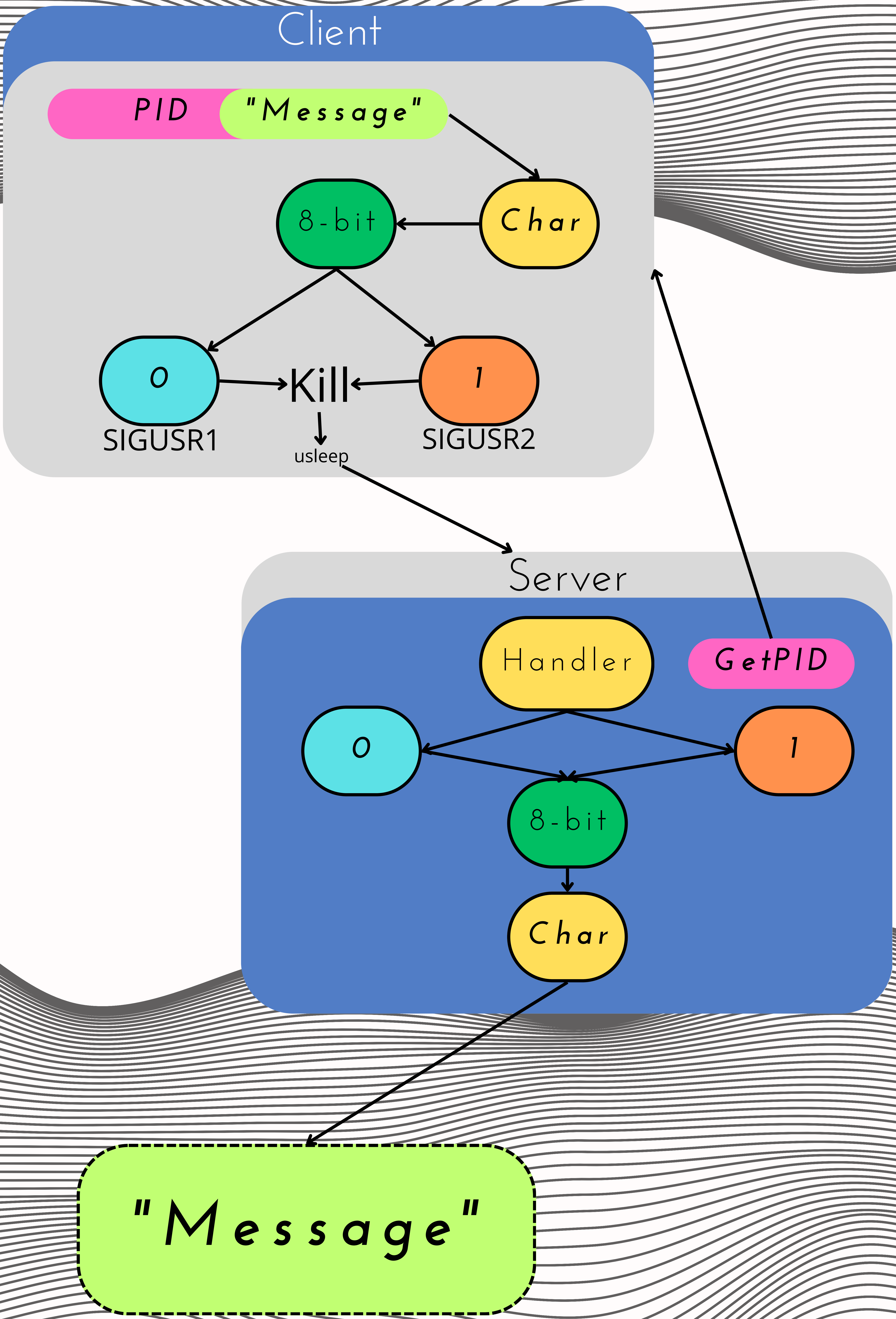
. -**Signal Flags:**
Definition: *Special options that modify the behavior of signal handling.*
Usage in MiniTalk: *The sa_flags field in sigaction specifies various flags that control aspects like whether system calls should be restarted if interrupted by a signal (SA_RESTART flag).*

. -**Special Signals**:
SIGUSR1:Definition: *User-defined signal 1.*
Usage in MiniTalk: *Used by the client to signal bit 1 in the binary representation of each character.*
SIGUSR2:Definition: *User-defined signal 2.*
Usage in MiniTalk: *Used by the client to signal bit 0 in the binary representation of each character.*

Signal Sets:

-**sigemptyset**:
Function: *Initializes an empty signal set.*
Usage in MiniTalk: *Used to initialize a signal set (sa_mask) in sigaction to specify which signals should be blocked while a signal handler is executing.*

Proces Visualization



Client

Simple Explanation of How the MiniTalk Client Code Works:

The MiniTalk client code sends text to a server using special signals. Here's how it works, step by step:

1. ***Including Necessary Files***

The code begins by including necessary functions and definitions from the minitalk.h file, which provides the tools needed for communication.

2. ***The send_sig Function***

This function sends a single character to the server:

Splitting the Character into Bits: Each character is converted into 8 bits (since each character in a computer is represented by 8 bits).

Sending the Bits: For each of the 8 bits:If the bit is 1, it sends the SIGUSR1 signal.If the bit is 0, it sends the SIGUSR2 signal.After sending each bit, the program waits a short time (100 microseconds) to ensure the signals are received correctly.

3. ***The main FunctionThe main part of the program works as follows:***

Checking Arguments: The program checks if exactly two arguments are provided: the server's PID and the message to send. If not, it shows information on how to run the program correctly.

Getting the PID: The program converts the first argument (the server's PID) into a number that identifies the server.

Getting the Message: The second argument is the message to send.

Sending the Message: The program goes through each character in the message:For each character, it calls the send_sig function to send that character to the server.

End-of-Message Signal: After sending all the characters, the program sends two additional newline characters (\n) to mark the end of the message.

Finishing Up: The program finishes and returns 0, indicating success.

Summary

The MiniTalk client code converts the message into bits and sends them as signals to the server. This allows the server to receive and interpret the data one character at a time.

Server

Simple Explanation *of How the MiniTalk Server Code Works*

The MiniTalk server code receives text from a client using special signals. Here's how it works, step by step:

1. ***Including Necessary Files:***The code begins by including the necessary functions and definitions from the minitalk.h file, which provides the tools needed for signal handling and other operations.

2. ***Global State:*** The server maintains a global state to keep track of the character being built and the current bit position.

Character and Bit Position:g_state.c stores the character being constructed.g_state.i tracks the current bit position within the character.

3. ***Signal Handlers***: Two functions handle the signals sent by the client:

Handling SIGUSR1 (Bit 1): When the server receives a SIGUSR1 signal, it sets the current bit of the character to 1.It then increments the bit position.If all 8 bits have been received, it writes the character to the standard output, and resets the character and bit position.

Handling SIGUSR2 (Bit 0): When the server receives a SIGUSR2 signal, it increments the bit position without changing the current bit (leaving it as 0).If all 8 bits have been received, it writes the character to the standard output, and resets the character and bit position.

4. **Setting Up Signal Handlers**: The function setup_signal_handlers configures the server to handle incoming signals.

Signal Handlers Initialization:The server initializes two sigaction structures to specify the handlers for SIGUSR1 and SIGUSR2.

Associating Signals with Handlers:It uses the sigaction function to link the SIGUSR1 signal with the handler for bit 1, and the SIGUSR2 signal with the handler for bit 0.

5. ***Main Function***:

The main part of the program works as follows:

Getting and Printing the PID*:* The server retrieves its own PID (Process ID) and prints it. The client needs this PID to send signals to the server.

Setting Up Signal Handlers: The server sets up the signal handlers so it can correctly interpret incoming signals.

Infinite Loop: The server enters an infinite loop to keep running and listening for signals indefinitely.

Return Statement: Although present, the return statement is never actually reached due to the infinite loop.

Summary

The MiniTalk server code listens for signals from a client and reconstructs characters bit by bit. When all 8 bits of a character have been received, it writes the character to the standard output. The server continuously runs, waiting for signals, making it ready to receive and process messages from the client.

SERVER

```
#include ”minitalk.h”

static t_State        g_state = {0, 0};

void        sig_user1(int sig)
{
        (void)sig;
        g_state.c |= (1 << g_state.i);
        g_state.i++;
        if (g_state.i == 8)
        {
                write(1, &g_state.c, 1);
                g_state.c = 0;
                g_state.i = 0;
        }
}

void        sig_user2(int sig)
{
        (void)sig;
        g_state.i++;
        if (g_state.i == 8)
        {
                write(1, &g_state.c, 1);
                g_state.c = 0;
                g_state.i = 0;
        }
}

void        setup_signal_handlers(void)
{
        struct sigaction        sa1;
        struct sigaction        sa2;

        sa1.sa_handler = sig_user1;
        sigemptyset(&sa1.sa_mask);
        sa1.sa_flags = 0;
        sa2.sa_handler = sig_user2;
        sigemptyset(&sa2.sa_mask);
        sa2.sa_flags = 0;
        sigaction(SIGUSR1, &sa1, NULL);
        sigaction(SIGUSR2, &sa2, NULL);
}

int        main(void)
{
        pid_t        pid;

        pid = getpid();
        ft_printf(”Your PID: %d\n”, pid);
        setup_signal_handlers();
        while (1)
        {
        }
        return (0);
}
```

CLIENT

```
#include ”minitalk.h”

void        send_sig(pid_t pid, char c)
{
        int         i;

        i = 0;
        while (i < 8)
        {
                if (((c >> i) & 1) == 1)
                {
                        kill(pid, SIGUSR1);
                }
                else
                {
                        kill(pid, SIGUSR2);
                }
                usleep(100);
                i++;
        }
}

int         main(int ac, char *av[])
{
        pid_t       pid;
        char        *str;

        if (ac != 3)
        {
                ft_printf(”Usage: %s <pid>
<string>\n”, av[0]);
                return (1);
        }
        pid = ft_atoi(av[1]);
        str = av[2];
        while (*str)
        {
                send_sig(pid, *str);
                str++;
        }
        send_sig(pid, ’\n’);
        send_sig(pid, ’\n’);
        return (0);
}
```

MINITALK.H

```
#ifndef MINITALK_H
# define MINITALK_H

# include ”ft_printf/ft_printf.h”
# include ”libft/libft.h”
# include <signal.h>
# include <stdio.h>
# include <stdlib.h>
# include <sys/types.h>
# include <unistd.h>

typedef struct State
{
        int        c;
        int        i;
}                t_State;

void        send_sig(pid_t pid, char c);
void        sig_user1(int sig);
void        sig_user2(int sig);

#endif
```