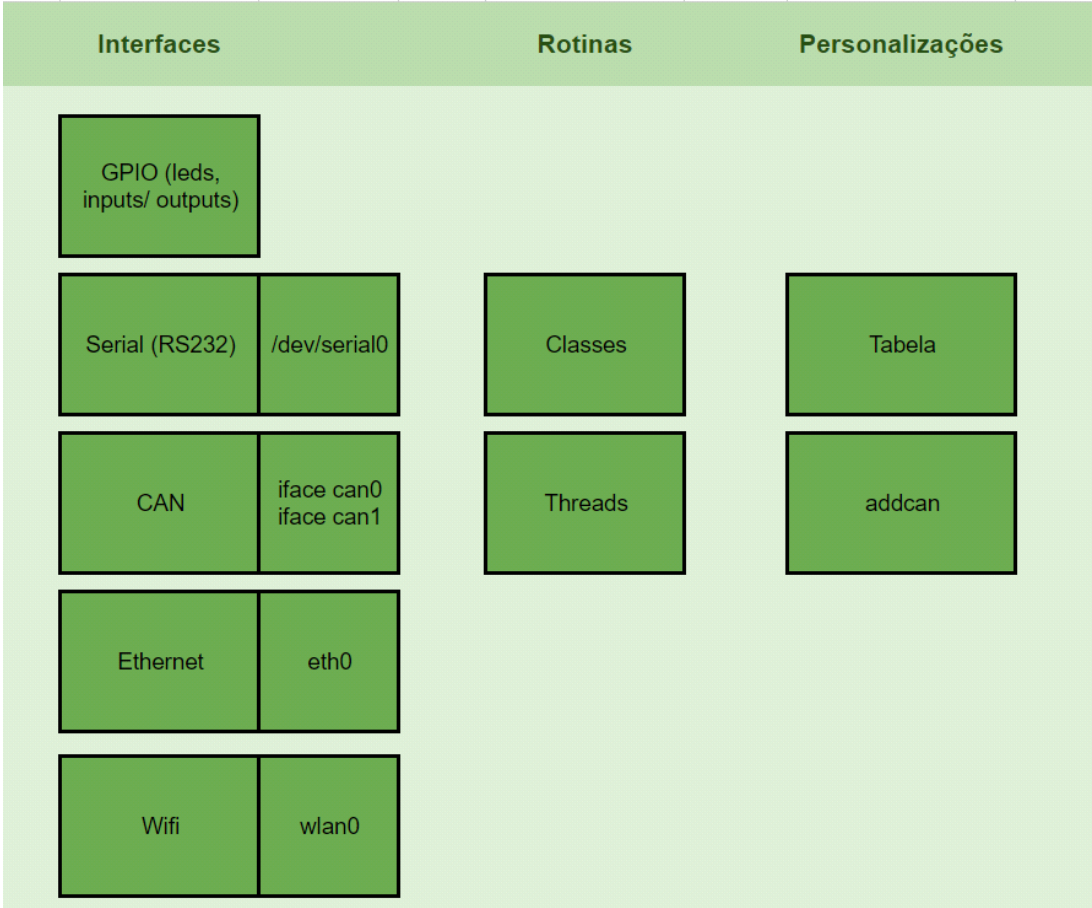
ROADMAP:



Bibliografia:

https://copperhilltech.com/blog/installing-pythoncan-on-the-raspberry-pi/

https://github.com/skpang/PiCAN-Python-examples/blob/master/cluster_rpm.py

https://github.com/skpang/PiCAN-Python-examples/blob/master/simple_rx_test.py

https://www.devmedia.com.br/comandos-importantes-linux/23893

https://gist.github.com/iximiuz/fe49427ac6900f2a590629500981e5ab

https://www.dio.me/articles/entenda-as-formas-de-transmissao-de-dados-atraves-da-internet

https://www.youtube.com/watch?v=S7Yle8clJ30

https://realpython.com/python-sockets/

https://www.activestate.com/blog/how-to-manage-threads-in-python/

https://superfastpython.com/multiprocessing-queue-in-python/#Need_for_a_Queue

https://pimylifeup.com/linux-kill-process/

https://www.w3schools.com/nodejs/nodejs_raspberrypi_webserver_websocket.asp

https://github.com/m3y54m/Embedded-Engineering-Roadmap

Power Shell – comandos:

- **ls**: Lista todos os arquivos do diretório;
- **df**: Mostra a quantidade de espaço usada no disco rígido;
- **top**: Mostra o uso da memória;
- **cd**: Mudar de diretório atual, como por exemplo cd diretório, cd .., cd /;
- **mkdir**: Cria um diretório;
- **rm**: Remove um arquivo/diretório;
- **cat**: Abre um arquivo (mostrando o conteúdo no console);
- **gip**: Get IP;
- **ssh**: Sessão segura, vem de Secure Shell, e permite-nos logar num servidor remoto através do protocolo ssh;
- **passwd**: Mudar a password do nosso utilizador (usuário logado);
- **mv**: Move ou renomeia arquivos ou diretórios;
- **netstat**: Mostra o estado da rede;
- **ifconfig**: Visualizar os IPS da nossa máquina, entre outras funções relacionadas com IPS;
- **ping**: Pingar um determinado host, ou seja, enviar pacotes ICMP para um determinado host e medir tempos de resposta, entre outras coisas;
- **^y**: Suspende o processo no próximo pedido de input;
- **^z**: Suspende o processo atual;
- **history**: Lista os últimos comandos usados, muito útil para lembrar também de que comandos foram usados para fazer determinada ação no passado ou o que foi feito em dada altura;

Exercício 1: Comunicar o Servidor CAN, receber a CAN do Inclinômetro LOHR e printar mensagem.

```
#!/usr/bin/python3.7

import can
import time
import os

try:
    bus = can.interface.Bus(channel='can1', bustype='socketcan')
except OSError:
    print('Cannot find PiCAN board.')
    exit()
```

```
print('Ready')

try:
    while True:
        print('entrou')
        message = bus.recv(0.5) # Wait until a message is received.
        if message is None:
            print('n recebeu mensagem')
        else:
            print('recebeu mensagem')
            c = '{0:f} {1:x} {2:x} '.format(message.timestamp,
message.arbitration_id, message.dlc)
            s=''
            for i in range(message.dlc):
                s +=  '{0:x} '.format(message.data[i])
            print(' {}'.format(c+s))
except KeyboardInterrupt:
    #Catch keyboard interrupt
    print('\n\rKeyboard interrtupt')
```

```
nuc@NUC:~/Python $ python3 /home/nuc/Python/LerInclinômetroServidorCAN.py
Ready
entrou
recebeu mensagem
 1695912689.235215 cf013a2 8 23 7e 8 7e ff ff 0 28
entrou
recebeu mensagem
 1695912689.254719 cf013a2 8 21 7e ff 7d ff ff 0 28
entrou
recebeu mensagem
 1695912689.274159 cf013a2 8 1f 7e 1 7e ff ff 0 28
```

Exercício 2: Comunicar o Servidor CAN 192.168.49.72, receber por CAN o status do LED.

```
#!/usr/bin/python3

import RPi.GPIO as GPIO
import can
import time
import os
from threading import Thread


lc0   = 18
```

```
lc1   = 3
lss   = 23

GPIO.setmode(GPIO.BCM)
GPIO.setwarnings(False)
GPIO.setup(lc0,GPIO.OUT)
GPIO.setup(lc1,GPIO.OUT)
GPIO.setup(lss,GPIO.OUT)

try:
    bus0 = can.interface.Bus(channel='can0', bustype='socketcan')
    bus1 = can.interface.Bus(channel='can1', bustype='socketcan')
    GPIO.output(lc0, False)
    GPIO.output(lc1, False)
    GPIO.output(lss, True)
except OSError:
    print('Cannot find PiCAN board.')
    GPIO.output(lc0, False)
    GPIO.output(lc1, False)
    GPIO.output(lss, False)
    exit()

def can_rx_task():
    tp = 0
    print('entrou can_rx_task()')
    while True:
        message = bus1.recv(0.001)  # Wait until a message is received.
        if message is None:
            #print('n recebeu mensagem')
            tp = tp +1
        else:
            print('recebeu mensagem  (', tp, ')')
            tp = 0
            c = '{0:f} {1:x} {2:x} '.format(message.timestamp,
message.arbitration_id, message.dlc)
            s=''
            for i in range(message.dlc):
                if(message.data[i]=='ff'):
                    GPIO.output(lc1,True)
                else:
                    GPIO.output(lc1,False)
                    s +=  '{0:x} '.format(message.data[i])
            print(' {}'.format(c+s))
        #time.sleep(0.001)
```

```
t = Thread(target = can_rx_task)
t.start()

# Main loop
try:
    while True:
        GPIO.output(lc0,True)
        print('vai enviar a mensagem')
        bus0.flush_tx_buffer()
        msg =
can.Message(arbitration_id=0xC0FFE,data=[0xff,0x01,0xff,0xff,0xff,0x00,0x00,0xff]
)
        bus0.send(msg,1)
        print('enviou mensagem: ')#+msg.data)
        time.sleep(0.1)
        GPIO.output(lc0,False)
        time.sleep(0.1)

except KeyboardInterrupt:
    #Catch keyboard interrupt
    GPIO.output(lc0,False)
    GPIO.output(lc1,False)
    GPIO.output(lss,False)
    #os.system("sudo /sbin/ip link set can0 down")
    #os.system("sudo /sbin/ip link set can1 down")
    print('\n\rKeyboard interrtupt')
```

```
nuc@NUC:~ $ python3 /home/nuc/Python/LuzesServidorCAN-teste2.py
entrou can_rx_task()
vai enviar a mensagem
enviou mensagem:
recebeu mensagem  ( 0 )
 1696006848.394981 c0ffe 8 ff 1 ff ff ff 0 0 ff
vai enviar a mensagem
enviou mensagem:
recebeu mensagem  ( 185 )
 1696006848.595481 c0ffe 8 ff 1 ff ff ff 0 0 ff
```

Exercício 3: Conectar Inclinômetro e CAN 1 para enviar dados, CAN0 receber e filtrar os dados da CAN1 para comandar LEDS 1 e 2, e valor alto no Inclinômetro comandar LED 3.

Ler mensagem da CAN, e definir modos de operação de LED:

Caso byte 1 == 0x01 a 0x7F: Desligado
Caso byte 1 == 0x80 a 0xFF: LED 1 aceso
Caso byte 2 == 0x01 a 0x7F: Desligado
Caso byte 2 == 0x80 a 0xFF: LED 2 aceso
Caso byte 1+2 (inclinômetro) > 0x7F: Manter LED 3 aceso, **else:** desligado
Caso byte 3+4 (inclinômetro) > 0x7F: Manter LED 3 aceso, **else:** desligado

Inclinômetro
ID: 0CF013A2
PGN: 61.459

Dados

Pitch: Byte 1-2
```
Função:
sum = hex(int(message.data[0], 16) + int(message.data[1], 16))
sum = (int(sum[2:],2)-64)/500
```
Roll: Byte 3-4
```
Função:
sum = hex(int(message.data[2], 16) + int(message.data[3], 16))
sum = (int(sum[2:],2)-64)/500
```

Offset: -64
Mult.: 500
Explicação:
1/0,002 = 500 (resolução de 0,002 deg por 1 bit)
64,51/0,002 = 32.255 bits
32.255/500 = 64,51 (Teste da Verdade)

min.: -64 deg || **máx.:** 64,51 deg
64/0,002 = 32.000
64,51/0,002 = 32.255
min.: -32.000 bits || **máx.:** 32.255 bits

```
#!/usr/bin/python3

import RPi.GPIO as GPIO
import can
from threading import Thread
from random import random
import time

lc0   = 18
lc1   = 3
lss   = 23
```

```
GPIO.setmode(GPIO.BCM)
GPIO.setwarnings(False)
GPIO.setup(lc0,GPIO.OUT)
GPIO.setup(lc1,GPIO.OUT)
GPIO.setup(lss,GPIO.OUT)

ID_INCLINOMETRO = 0x0CF013A2
ID_CAN0 = 0xC0FFE

try:
    bus0 = can.interface.Bus(channel='can0', bustype='socketcan')
    bus1 = can.interface.Bus(channel='can1', bustype='socketcan')
    GPIO.output(lc0,False)
    GPIO.output(lc1,False)
    GPIO.output(lss,True)
except OSError:
    print('Cannot find PiCAN board.')
    GPIO.output(lc0,False)
    GPIO.output(lc1,False)
    GPIO.output(lss,False)
    exit()


def controle_leds(message1, message2):
    if(message1 < 127):
        GPIO.output(lc0,False)
        time.sleep(0.01)
    elif(message1 >= 127):
        GPIO.output(lc0,True)
        time.sleep(0.01)

    if(message2 < 127):
        GPIO.output(lc1,False)
        time.sleep(0.01)
    elif(message2 >= 127):
        GPIO.output(lc1,True)
        time.sleep(0.01)

def controle_led_inclinometro(message):
    pitch = (((message.data[0] << 8)+(message.data[1])) - 32000)/500
    roll = (((message.data[0] << 8)+(message.data[1])) - 32000)/500
    if(pitch > 30 or roll > 30):
        GPIO.output(lss,True)
        time.sleep(0.01)
```

```
    else:
        GPIO.output(lss,False)
        time.sleep(0.01)

def can_rx_task():
    tp = 0
    while True:
        message = bus1.recv(0.001)  # Wait until a message is received.
        if message is None:
            tp = tp + 1
        else:
            print('recebeu mensagem  (', tp, ')')
            tp = 0
            c = '{0:f} {1:x} {2:x} '.format(message.timestamp,
message.arbitration_id, message.dlc)
            s=''
            for i in range(message.dlc):
                if(message.arbitration_id==ID_INCLINOMETRO):
                    controle_led_inclinometro(message)
                elif(message.arbitration_id==ID_CAN0):
                    controle_leds(message.data[0],message.data[1])
                s +=  '{0:x} '.format(message.data[i])
            print(' {}'.format(c+s))
        time.sleep(0.001)

def can_tx_task():
    while True:
        msg = can.Message(arbitration_id=ID_CAN0,data=[round(random() *
0xff),round(random() * 0xff),round(random() * 0xff),round(random() *
0xff),0xff,0x00,0x00,0xff])
        bus0.flush_tx_buffer()
        bus0.send(msg,0.001)
        time.sleep(0.1)

# Main loop
try:
    if __name__ =="__main__":
        # creating thread
        t1 = Thread(target = can_rx_task)
        t2 = Thread(target = can_tx_task)
        t3 = Thread(target = can_rx_task)

        # starting thread 1
        t1.start()
```

```
        # starting thread 2
        t2.start()
        # starting thread 2
        t3.start()

        # wait until thread 1 is completely executed
        t1.join()
        # wait until thread 2 is completely executed
        t2.join()
        # wait until thread 2 is completely executed
        t3.join()

        # both threads completely executed
        print("Done!")
except KeyboardInterrupt:
    #Catch keyboard interrupt
    GPIO.output(lc0,False)
    GPIO.output(lc1,False)
    GPIO.output(lss,False)
    #os.system("sudo /sbin/ip link set can0 down")
    #os.system("sudo /sbin/ip link set can1 down")
    print('\n\rKeyboard interrtupt')
```

```
nuc@NUC:~/Python $ python3 Luzes-InclinometroServidorCAN.py ]
recebeu mensagem  ( 0 )
recebeu mensagem  ( 4 )
 1696104730.980586 cf013a2 8 5d 7e f3 7c ff ff 0 28
recebeu mensagem  ( 0 )
 1696104730.973553 c0ffe 8 5b 5c a0 90 ff 0 0 ff
recebeu mensagem  ( 0 )
 1696104731.000004 cf013a2 8 5f 7e f3 7c ff ff 0 28
recebeu mensagem  ( 0 )
 1696104731.019427 cf013a2 8 5d 7e f0 7c ff ff 0 28
recebeu mensagem  ( 0 )
 1696104731.038926 cf013a2 8 54 7e f0 7c ff ff 0 28
recebeu mensagem  ( 0 )
 1696104731.058385 cf013a2 8 4e 7e ec 7c ff ff 0 28
recebeu mensagem  ( 0 )
 1696104731.077665 cf013a2 8 4b 7e ef 7c ff ff 0 28
recebeu mensagem  ( 0 )
 1696104731.073899 c0ffe 8 3c b9 89 c8 ff 0 0 ff
```

**O código está funcional, porém seria melhor se funcionasse através de ciclos ao invés de delay. Pois o delay (“time.sleep()”) para o programa, sendo que não podemos impedir o recebimento de outros ID.

Exercício 4: Enviar dados Ethernet – Raspberry to PC:

```
# Python Server
from socket import *

bufferSize = 1024
msgFromServer = "Howdy Client, happy to be your server"
```

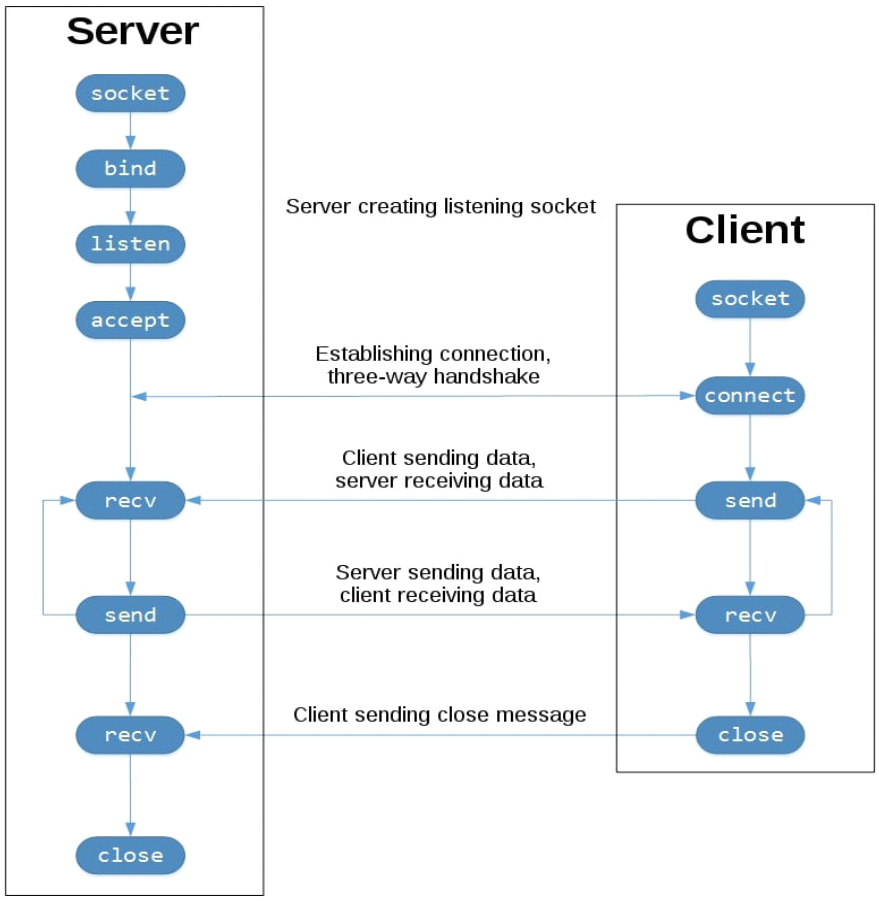
```
ServerPort = 2222
ServerIP = '192.168.49.89'
bytesToSend = msgFromServer.encode('utf-8')
RPIsocket = socket(AF_INET, SOCK_DGRAM)
RPIsocket.bind((ServerIP,ServerPort))
print('Server is Up and Listening...')
cnt = 0

try:
    while True:
        message,address = RPIsocket.recvfrom(bufferSize)
        message = message.decode('utf-8')
        print(message)
        print('Client Adress: ',address[0])
        if message=='INC':
            cnt=cnt+1
        if message=='DEC':
            cnt=cnt-1
        msg=str(cnt)
        print(msg)
        msg=msg.encode('utf-8')
        RPIsocket.sendto(msg,address)
except KeyboardInterrupt:
    socket.close()
    print('\n\rKeyboard interrtupt')

# Python Client
from socket import *

msgFromClient='Howdy Server, from your Client'
bytesToSend = msgFromClient.encode('utf-8')
serverAddress = ('192.168.49.89', 2222)
bufferSize=1024
UDPClient = socket(AF_INET, SOCK_DGRAM)

try:
    while True:
        cmd = input('What do you want to do with counter, INC or DEC?')
        cmd = cmd.encode('utf-8')
        UDPClient.sendto(cmd, serverAddress)
        data, address = UDPClient.recvfrom(bufferSize)
        data = data.decode('utf-8')
        print('Data from Server: ', data)
        print('Server IP Address: ', address[0])
```

```
        print('Server Port: ', address[1])
except KeyboardInterrupt:
    socket.close()
    print('\n\rKeyboard interrtupt')
```

- Como os dados são transferidos pelo cabo de Ethernet?

Enviado bit por bit. Você pensaria que isso seria um processo lento, mas considerando que somos capazes de alterar as tensões com rapidez suficiente para atingir gigabits por segundo, isso realmente faz você pensar.

Sem nos aprofundarmos nos diferentes tipos de modos de transmissão, onde usamos diferentes níveis de tensão para transmitir de maneira mais eficiente do que simplesmente dizer que a tensão próxima ao solo é 0 e +5v é 1, é essencialmente isso que acontece. Digamos que você queira transferir o número 5 pela rede. Se ignorarmos o cabeçalho, o que precisamos fazer é converter o número 5 para o valor ASCII, que é 35, e depois para o binário 0010 0101. Transferiremos primeiro os bits mais à direita. Isso significa que literalmente enviamos 1, 0, 1, 0, 0, 1, 0, 0 com pulsos de eletricidade extremamente rápidos e a extremidade receptora está programada para saber como receber isso porque temos algo chamado protocolos que são essencialmente acordos em transferência de dados, onde concordamos que enviamos primeiro os bits mais corretos e concordamos com o formato do cabeçalho, quanto tempo uma mensagem (pacote) pode durar antes de ser dividida em mensagens menores.
A última coisa a considerar é o cabeçalho que nos permite enviar para mais de uma máquina. Os cabeçalhos contêm endereços de origem e destino, bem como outras informações para gerenciar o pacote. A fonte diz de quem é e o destino diz a quem se destina. O endereço de destino é bastante volátil à medida que se move em uma rede. Por exemplo, se você estiver jogando um jogo online com um amigo na casa dele e ele estiver atrás de um roteador, quando o seu PC enviar uma mensagem para o PC dele (assumindo que o jogo não usa um servidor e usa um protocolo de host), o endereço IP no destino começa como sendo o IP do seu roteador, depois qualquer roteador entre o seu roteador e o roteador dele e, finalmente, o roteador dele, que sabe como enviá-lo para o PC dele.



Server: Raspberry Pi 4– IP 192.168.49.89
Comportamento: Recebe a solicitação do Client e envia a resposta desejada.
Client: PC – IP 192.168.49.20
Comportamento: Envia a solicitação para o Server e recebe a resposta.

Exercício 5: Receber CAN e enviar bytes recebidos por Ethernet

Comportamento Raspberry Pi 4: receber mensagem CAN0 e CAN1, e checar se o Client fez alguma requisição. Se fez, ativa para enviar o próximo dado que corresponda a solicitação. Se não, continua processo de CAN.

Comportamento PC: Laço de repetição onde pergunta ao usuário se deseja fazer a solicitação. Se não, continua perguntando. Se sim, faz a requisição e espera a resposta, depois printa. Ao finalizar, volta a perguntar.

Python

THREADS

Para trabalhar em um processo que exija muito tempo de processamento, threads não são a melhor opção, pois toda vez que a thread é chamada, tem um processamento por trás. A memória precisa ser designada e reivindicada a cada thread executada. Em muitos casos, thread órfãs podem ocorrer quando não são “desligadas” corretamente, ocupando um precioso tempo da CPU.
A melhor opção seria as “queues”, de estrutura First In, First Out (FIFO), é o jeito mais fácil de rodar tarefas tanto síncronas com assíncronas, de uma maneira que evite “race conditions”, “deadlocks” e outros problemas que a threads causam. Custam menos do processamento e da memória.

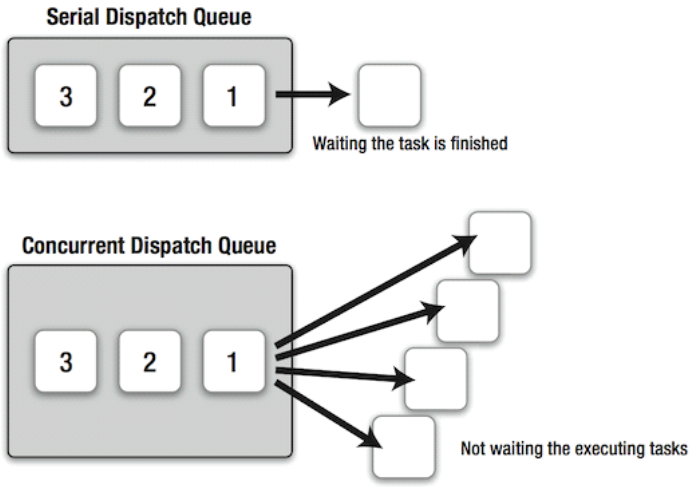
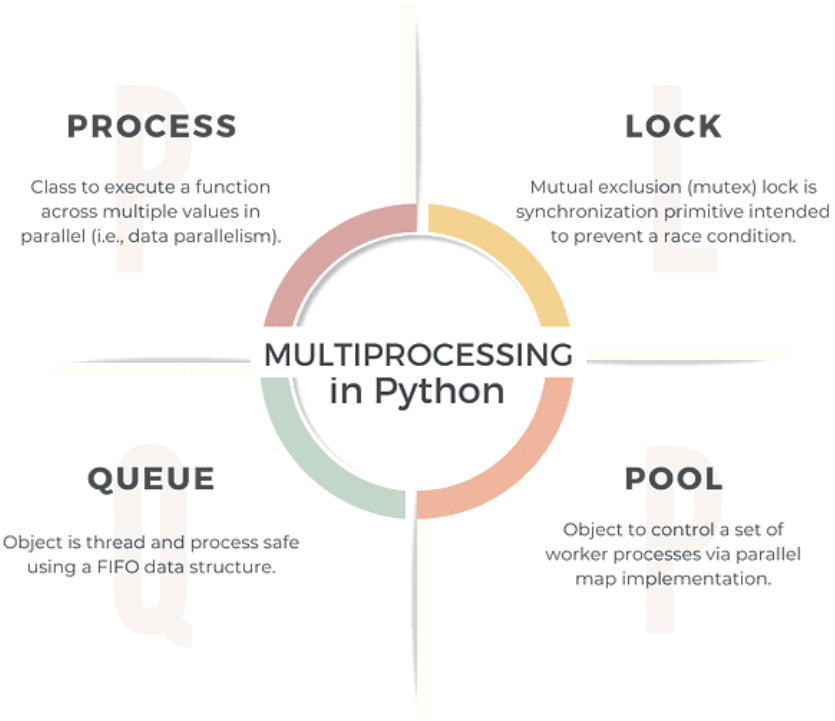


Figure 7–2. *Serial Dispatch Queue and Concurrent Dispatch Queue*

MULTIPROCESSING



- “threading” é adequado para tarefas I/O vinculadas e compartilha memória entre threads, enquanto o multiprocessamento é ideal para tarefas vinculadas à CPU e usa espaços de memória separados.

- “concurrent.futures” fornece uma interface de alto nível para execução simultânea usando threads ou processos, abstraindo as complexidades de gerenciá-los manualmente.

- “asyncio” é uma biblioteca para programação assíncrona, aproveitando corrotinas e loops de eventos para lidar com I/O vinculadas.

C++

Server:

```
#include <iostream>
#include <string>
#include <cstring>
#include <unistd.h>
#include <net/if.h>
#include <sys/ioctl.h>
```

```cpp
#include <sys/socket.h>
#include <linux/can.h>
#include <linux/can/raw.h>
#include <thread>
#include <mutex>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <random>

int s, can0, can1;
bool runThreads = true; // Flag to control thread execution
uint32_t clientSearchIDCAN; // Store the IDCAN received from the client
std::mutex mtx;
struct can_frame frameToSend;
bool enviou = false;

int initCAN(const char* can_interface) {
    struct sockaddr_can addr;
    struct ifreq ifr;

    if ((s = socket(PF_CAN, SOCK_RAW, CAN_RAW)) < 0) {
        perror("Socket creation failed");
        return -1;
    }

    strcpy(ifr.ifr_name, can_interface);
    ioctl(s, SIOCGIFINDEX, &ifr);
    addr.can_family = AF_CAN;
    addr.can_ifindex = ifr.ifr_ifindex;

    if (bind(s, (struct sockaddr*)&addr, sizeof(addr)) < 0) {
        perror("Bind failed");
        return -1;
    }

    return s;
}

void receiveCANFrame() {
    while (runThreads) {
        struct can_frame frame;
        int nbytes = read(can0, &frame, sizeof(struct can_frame));
        if (nbytes < 0) {
            perror("Read error");
```

```
            return;
        }

        std::cout << "Received CAN frame ID: " << std::hex << frame.can_id <<
std::dec << " Data: ";
        for (int i = 0; i < frame.can_dlc; i++) {
            std::cout << "0x" << std::hex << static_cast<int>(frame.data[i]) << "
";
        }
        std::cout << std::endl;
        // Check if the received CAN frame has the IDCAN you are looking for
        if (frame.can_id == clientSearchIDCAN && enviou == false) {
            mtx.lock();
            frameToSend = frame;
            mtx.unlock();
            //std::cout << "clientSearchIDCAN: " << clientSearchIDCAN <<
std::endl;
        }
    }
}

void sendCANFrame(uint32_t can_id, uint8_t* data, uint8_t data_length) {
    while (runThreads) {
        struct can_frame frame;
        frame.can_id = can_id;
        frame.can_dlc = data_length;
        memcpy(frame.data, data, data_length);

        int nbytes = write(can1, &frame, sizeof(struct can_frame));
        if (nbytes < 0) {
            perror("Write error");
            return;
        }

        usleep(1000000); // Sleep for 1 second between transmissions
    }
}

void handleClient(int client_socket) {
   // while (runThreads) {
        if (enviou == false) {
            // Set a maximum number of retry attempts
            const int maxRetryAttempts = 5;
            int retryCount = 0;
```

```
            while (retryCount < maxRetryAttempts) {
                if (frameToSend.can_id == clientSearchIDCAN) {
                    send(client_socket, &frameToSend, sizeof(struct can_frame),
0);
                    enviou = true;
                    std::cout << "Sending CAN frame to Client - ID: " << std::hex
<< frameToSend.can_id << std::dec << " Data: ";
                    for (int i = 0; i < frameToSend.can_dlc; i++) {
                        std::cout << "0x" << std::hex <<
static_cast<int>(frameToSend.data[i]) << " ";
                    }
                    std::cout << std::endl;

                    break; // Response sent successfully, exit the loop
                }
                else {
                    // If the response isn't ready, wait for a short time and
retry
                    std::cout << "retrycount: " << retryCount << std::endl;
                    std::this_thread::sleep_for(std::chrono::seconds(1));
                    retryCount++;
                }
            }

            if (retryCount == maxRetryAttempts) {
                std::cout << "Max retry attempts reached. No response sent to the
client." << std::endl;
            }
        }
        char buffer[4];
        ssize_t bytes_received = recv(client_socket, buffer, sizeof(buffer), 0);
        if (bytes_received == sizeof(uint32_t)) {
            uint32_t receivedIDCAN;
            memcpy(&receivedIDCAN, buffer, sizeof(receivedIDCAN));

            mtx.lock();
            clientSearchIDCAN = receivedIDCAN;
            mtx.unlock();
            enviou = false;
            //Inform the user about the received IDCAN
            std::cout << "Received IDCAN from client: " << std::hex <<
receivedIDCAN << std::dec << std::endl;
        }
        close(client_socket);
```

```
    //}
}

uint32_t randomHex(int size) {
    std::random_device rd;  // Obtain a random seed from the OS entropy device
    std::mt19937 gen(rd()); // Standard Mersenne Twister random number generator
    if (size == 2) {
        std::uniform_int_distribution<uint16_t> dis(0x00, 0xFF); // Define the
range for your hexadecimal number
        return dis(gen);
    }
    else if (size == 6) {
        std::uniform_int_distribution<uint32_t> dis(0x000000, 0xFFFFFF); //
Define the range for your hexadecimal number
        return dis(gen);
    }
    else {
        std::uniform_int_distribution<uint32_t> dis(0x00000000, 0xFFFFFFFF); //
Define the range for your hexadecimal number
        return dis(gen);
    }
}

int main() {
    const char* can_0 = "can0"; // Replace with your CAN interface name
    const char* can_1 = "can1"; // Replace with your CAN interface name

    can0 = initCAN(can_0);
    can1 = initCAN(can_1);

    if (can0 == -1 && can1 == -1) {
        return -1;
    }

    // Create threads for receiving and sending CAN frames
    std::thread receiver(receiveCANFrame); // Pass an initial value of 0
    std::thread sender(sendCANFrame, randomHex(8), new uint8_t[8]{ 0x11, 0x22,
0xFF, 0xEF, 0xAB, 0x6C, 0xFF, 0x00 }, 8); // Example CAN ID and data

    // Set up a TCP server to listen for client connections
    int server_socket = socket(AF_INET, SOCK_STREAM, 0);
    if (server_socket == -1) {
        perror("Socket creation failed");
        return -1;
```

```
    }

    struct sockaddr_in server_address;
    server_address.sin_family = AF_INET;
    server_address.sin_addr.s_addr = INADDR_ANY;
    server_address.sin_port = htons(8888); // Change the port as needed

    if (bind(server_socket, (struct sockaddr*)&server_address,
sizeof(server_address)) < 0) {
        perror("Bind failed");
        return -1;
    }

    listen(server_socket, 5); // Listen for up to 5 connections

    while (true) {
        int client_socket = accept(server_socket, NULL, NULL);
        std::thread client_handler(handleClient, client_socket);
        client_handler.detach(); // Allow the handler thread to run independently
    }

    receiver.join();
    sender.join();

    // Close the socket when done
    close(server_socket);
    runThreads = false; // Set the flag to stop the threads

    close(s);

    return 0;
}
```

Client:

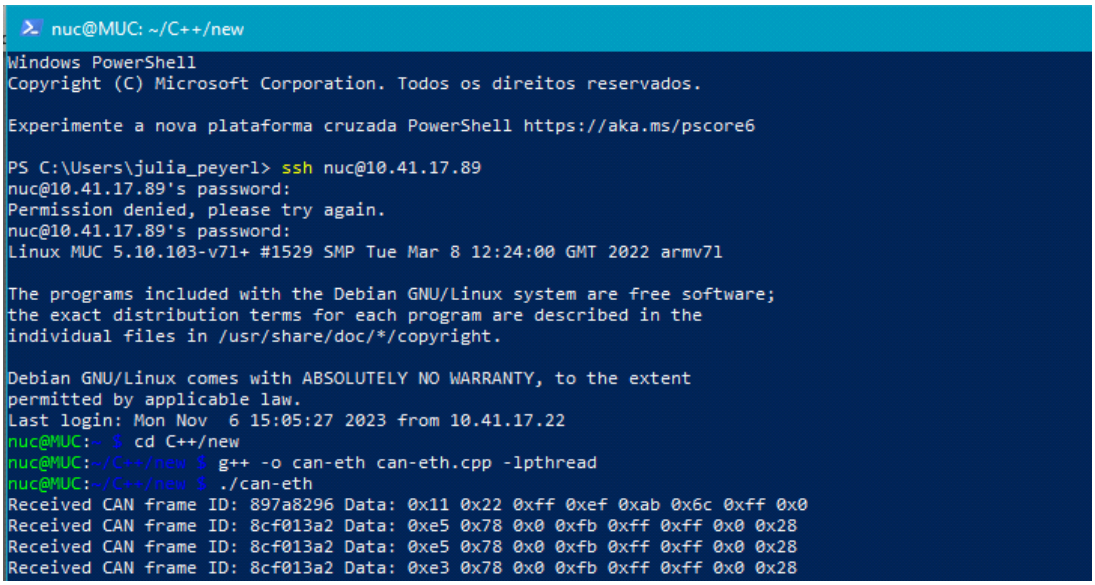
```
#include <iostream>
#include <unistd.h>
#include <WinSock2.h>


#pragma comment(lib, "ws2_32.lib")

typedef uint32_t canid_t;
```

```
struct can_frame {
    union {
        canid_t can_id;
        struct {
            uint32_t id : 29;
            uint32_t err : 1;
            uint32_t rtr : 1;
            uint32_t eff : 1;
        };
    };
    union
        {
        uint8_t can_dlc;
        uint8_t dlc;
        };
    uint8_t __pad;
    uint8_t __res0;
    uint8_t __res1;
    uint8_t data[8] __attribute__((aligned(8)));
};

int main() {

    int maxRetries = 1000;
    int retryCount = 0;

    do {
        WSADATA wsaData;
        if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {
            std::cerr << "WSAStartup failed." << std::endl;
            return 1;
        }

        SOCKET clientSocket = socket(AF_INET, SOCK_STREAM, 0);
        if (clientSocket == INVALID_SOCKET) {
            std::cerr << "Socket creation failed." << std::endl;
            WSACleanup();
            return 1;
        }

        sockaddr_in serverAddress;
        serverAddress.sin_family = AF_INET;
        serverAddress.sin_port = htons(8888);  // Change the port as needed
```

```
        serverAddress.sin_addr.s_addr = inet_addr("10.41.17.189");  // Replace
with the server's IP address

        if (connect(clientSocket, reinterpret_cast<SOCKADDR*>(&serverAddress),
sizeof(serverAddress)) == SOCKET_ERROR) {
            std::cerr << "Connection failed." << std::endl;
            closesocket(clientSocket);
            WSACleanup();
            return 1;
        }

        bool responseReceived = false;
        uint32_t idcan;
        std::cout << "Enter IDCAN (in hexadecimal): ";
        std::cin >> std::hex >> idcan;

        if (send(clientSocket, reinterpret_cast<char*>(&idcan), sizeof(idcan), 0)
== SOCKET_ERROR) {
            std::cerr << "Error sending IDCAN." << std::endl;
            closesocket(clientSocket);
            WSACleanup();
            return 1;
        }

        while (retryCount < maxRetries && responseReceived == false) {
            char buffer[sizeof(struct can_frame)];
            ssize_t bytes_received = recv(clientSocket, buffer, sizeof(buffer),
0);
            if (bytes_received < 0) {
                perror("Read error");
                break;
            }
            else {
                if (bytes_received == sizeof(struct can_frame)) {
                    struct can_frame frame;
                    memcpy(&frame, buffer, sizeof(struct can_frame));
                    memset(buffer, 0, sizeof(buffer));
                    if (frame.can_id == idcan) {
                        responseReceived = true;

                        std::cout << "Received IDCAN from server: " << std::hex
<< frame.can_id << std::dec << std::endl;
                        std::cout << "Received CAN message data: ";
                        for (int i = 0; i < frame.can_dlc; i++) {
```

```
                            std::cout << "0x" << std::hex << (unsigned
int)frame.data[i] << " ";
                        }
                    std::cout << std::endl;
                }
            }
            else {
                std::cerr << "Ainda nao recebi esse dado!" << std::endl;
                responseReceived = false;
                retryCount++;
            }
        }
    }

    if (responseReceived == false) {
        std::cout << "Max retry attempts reached. No response received." <<
std::endl;
    }

    closesocket(clientSocket);

    std::cout << "Do you want to continue? Y=1 | N=0  -->  ";
    std::cin >> retryCount;

} while (retryCount);

WSACleanup();

return 0;
}
```

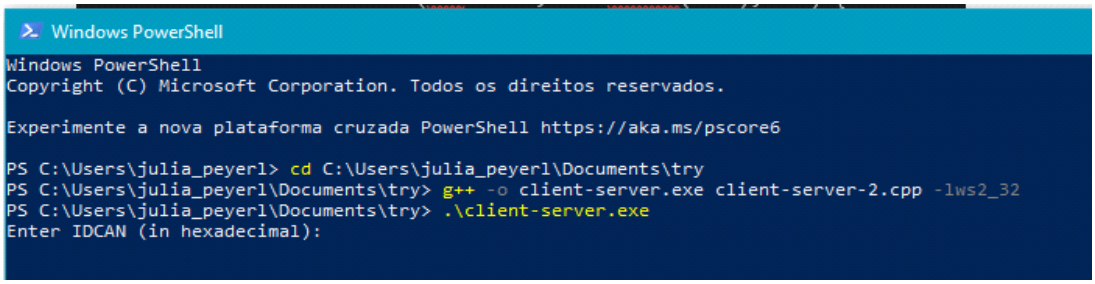
O código do “server” está no ambiente remoto do Servidor Can, sendo acessado e iniciado através do PowerShell da máquina.



Comandos:

- “ssh” Acessar o servidor CAN;
- “cd” Acessar o ambiente/pasta onde está o arquivo;
- “g++ -o can-eth can-eth.cpp -lpthread” “g++” é o compilador do C++, transforma o arquivo “.cpp” em executável no servidor e “-lpthread” flag para criar threads comando para criar o arquivo executável;
- “./” executar a aplicação;

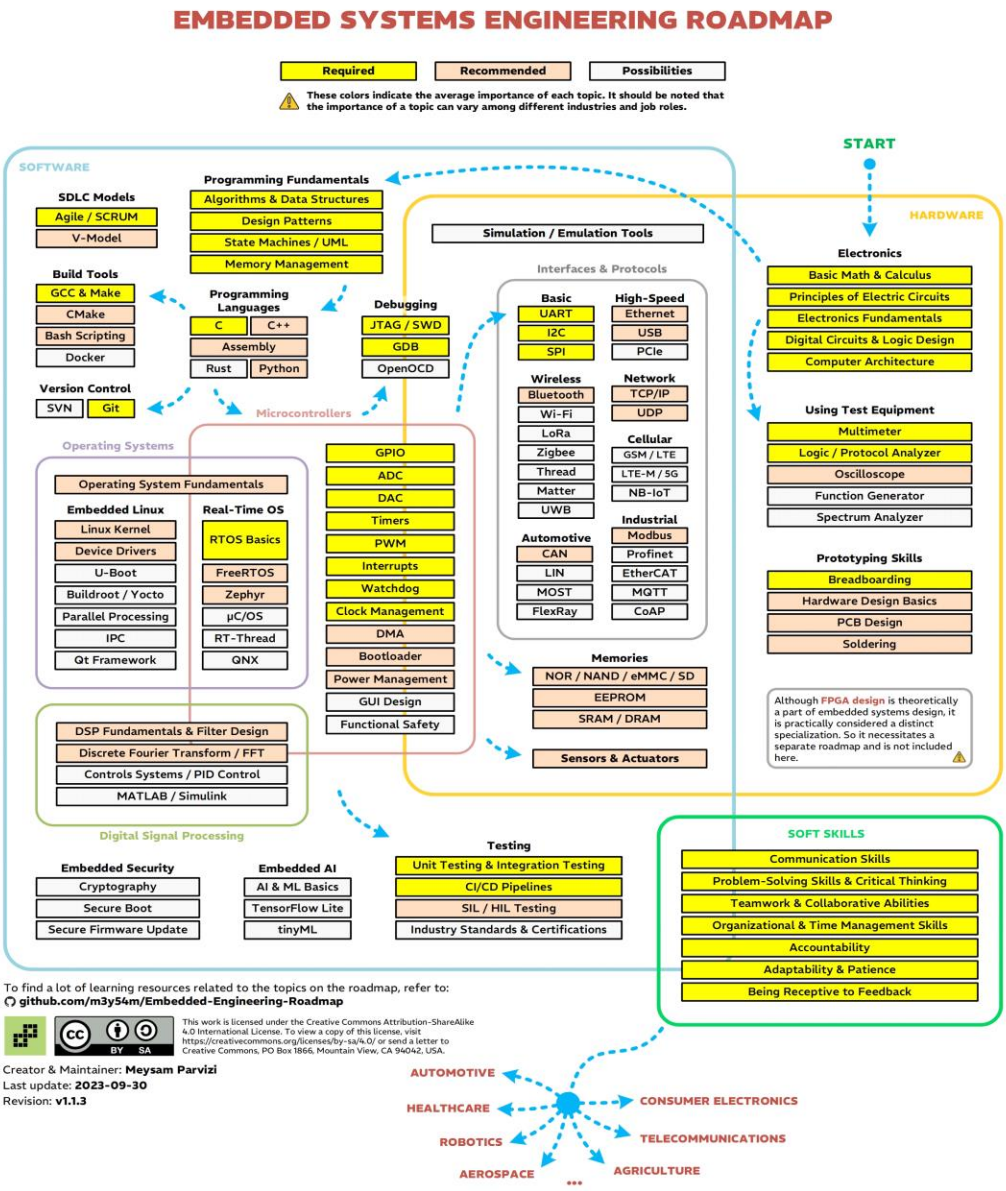
Já o código do “client” está armazenado no PC, mas também roda pelo PowerShell.



Comandos:

- “cd” Acessar o ambiente/pasta onde está o arquivo;
- “g++ -o client-server.exe client-server-2.cpp -lws2_32” “g++” é o compilador do C++ (foi necessário instalar para Windows), transforma o arquivo “.cpp” em executável “.exe” e “-lws2_32” flag para habilitar o WinSock (coloca na linha de comando, devido a não ser um projeto com dependências, se não tinha que incluir na pasta do projeto) para criar o arquivo para executar;
- “.\” executa o “.exe”;

Sistemas Embarcados com C++



Memory Management

O compilador reserva espaço na memória para todos os dados declarados explicitamente, mas se usarmos ponteiros precisamos colocar no ponteiro um endereço de memória existente. Para isto, podemos usar o endereço de uma variável definida previamente ou reservar o espaço necessário no momento que precisemos. Este espaço que precisamos reservar em tempo de execução é chamada de memória alocada dinamicamente.

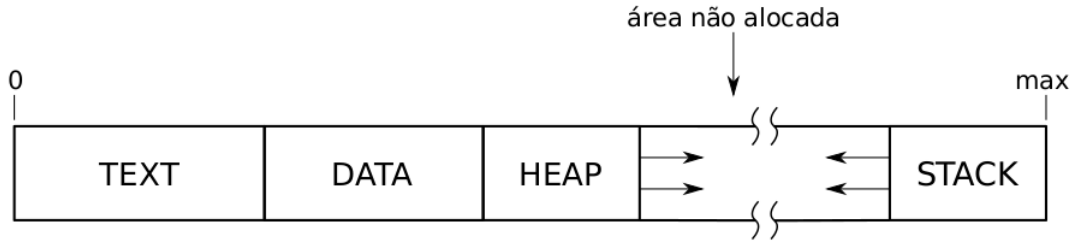
Reservar dinamicamente é o caso em que não sabemos, no momento da programação, a quantidade de dados que deverão ser inseridos quando o programa já está sendo executado. Em

vez de tentarmos prever um limite superior para abarcar todas as situações de uso da memória, temos a possibilidade de reservar memória de modo dinâmico. Os exemplos típicos disto são os processadores de texto, nos quais não sabemos a quantidade de caracteres que o utilizador vai escrever quando o programa estiver sendo executado. Nestes casos podemos, por exemplo, receber a quantidade de caracteres que o usuário digita e depois alocamos a quantidade de memória que precisamos para guardá-lo e depois o armazenamos para uso posterior.

O modo mais comum de alocação de memória é o de alocar certa quantidade de bytes e atribuí-la a um ponteiro, provendo um "array", ou vetor. Nos tópicos a seguir abordaremos estes e outros casos de uso de memória alocada dinamicamente.

O espaço de endereços de um processo em execução é dividido em vários segmentos lógicos. Os mais importantes são:

- ***Text*:** contém o código do programa e suas constantes. Este segmento é alocado durante a criação do processo ("exec") e permanece do mesmo tamanho durante toda a vida do processo.
- ***Data*:** este segmento é a memória de trabalho do processo, aonde ficam alocadas as variáveis globais e estáticas. Tem tamanho fixo ao longo da execução do processo.
- ***Stack*:** contém a pilha de execução, onde são armazenados os parâmetros, endereços de retorno e variáveis locais de funções. Pode variar de tamanho durante a execução do processo.
- ***Heap*:** contém blocos de memória alocadas dinamicamente, a pedido do processo, durante sua execução. Varia de tamanho durante a vida do processo.



Um programa em C suporta três tipos de alocação de memória:

- A **alocação estática** ocorre quando são declaradas variáveis globais ou estáticas; geralmente alocadas em *Data*.
- A **alocação automática** ocorre quando são declaradas variáveis locais e parâmetros de funções. O espaço para a alocação dessas variáveis é reservado quando a função é invocada, e liberado quando a função termina. Geralmente é usada a pilha ("*stack*").
- A **alocação dinâmica**, quando o processo requisita explicitamente um bloco de memória para armazenar dados; o controle das áreas alocadas dinamicamente é manual ou semiautomático: o programador é responsável por liberar as áreas alocadas dinamicamente. A alocação dinâmica geralmente usa a área de "*heap*".

Para manipulação de endereços de memória, a linguagem C/C++ oferece dois operadores:

~ O operador & ("endereço de");

~ O operador * ("conteúdo de");

Quando ponteiros apontam para registros, o acesso aos campos desses registros pode ser realizado de duas formas:

~ usando o operador * ("conteúdo de")

~ usando o operador -> ("seta")

Ponteiros soltos: um ponteiro solto é aquele que contém o endereço de memória de uma região que já foi liberada	**Variáveis dinâmicas perdidas:** uma variável que foi alocada dinamicamente, mas não está mais acessível, pois não ponteiros que apontem para ela.