Teste de conhecimento em programação Linux

O primeiro grupo de testes consiste de algumas questões teóricas e uma questão de desenvolvimento.

1. Ponteiros são uma ferramenta de programação bastante importante em C/C++. Explique seu funcionamento e crie um exemplo de utilização.

Ponteiros são formas de endereçamento indireto que apontam um determinado endereço na memória. São bastante úteis para referenciar variáveis que podem estar fora do escopo de determinada função ou fazer com que uma função aja sobre (ou acesse) uma variável sem precisar copiá-la para dentro do escopo de uma função (através da passagem de argumentos).

Um exemplo recente de utilização dessa ferramenta (no âmbito de minhas recentes experiências) tem sido para referenciar handle structs na criação de bibliotecas para a implementação de protocolos de comunicação de hardware na plataforma da ST (utilizando a STM32CubeIDE).

Por exemplo, escrevi uma biblioteca em ANSI C para compor todas as funções de operação de um chip seletor de canais I2C - PCA9546A - que mais funcionou como um multiplexador para vários canais I2C. Seu datasheet pode ser encontrado no site da própria fabricante (NXP).

Seu handle struct pôde ser definido como:

```
typedef struct PCA9546A_HandleTypeDef {
    I2C_HandleTypeDef* hi2c;
    uint8_t ctrlreg;
    GPIO_TypeDef* RST_GPIOx;
    uint16_t RST_Pin;
} PCA9546A_HandleTypeDef;
```

E sua função para seleção de canal foi definida como:

```
void PCA9546A_SelectChannel(PCA9546A_HandleTypeDef* hmux, uint8_t channel)
{
    hmux->ctrlreg = channel;
    PCA9546A_Reset(hmux);
    HAL_I2C_Master_Transmit(
        hmux->hi2c, PCA9546A_I2C_ADDRESS_WRITE,
        &(hmux->ctrlreg), 1, PCA9546A_DEFAULT_TIMEOUT);
}
```

Perceba que o primeiro argumento da função é exatamente uma referência a um handle struct definido para o dispositivo em questão. Também pode ser notado que o próprio dispositivo guarda um ponteiro para o handle da interface I2C sendo utilizada pelo próprio dispositivo. Assim, temos uma struct global guardando todos os "atributos" referentes a esse dispositivo do ponto de vista do usuário da biblioteca.

Dessa forma, tudo o que o usuário da biblioteca precisa fazer é declarar um handle struct global e acessá-lo de forma indireta para que todas as funções consigam referenciar esse mesmo handle struct dentro de seu próprio escopo.

```
PCA9546A_HandleTypeDef hmux;
```

E então utilizar um ponteiro para esse handle como argumento das funções, como por exemplo:

```
PCA9546A_SelectChannel(&hmux, (1<<2));
```

Ponteiros são uma ferramenta excelente para o desenvolvimento pleno de todo um projeto em C/C++.

2. Em Linux quais as diferenças e semelhanças entre processos e threads? Indique uma vantagem e uma desvantagem de usar cada um deles.

Um processo se trata de um programa em execução. Um sistema Linux tem vários processos em execução a todo momento. Processos no Linux não interrompem uns aos outros, são isolados (não compartilham a memória um do outro) e compartilham a mesma CPU. Processos podem realizar mais de um unidade de trabalho de forma concorrente, criando uma ou mais threads.

Threads são rotinas pequenas, iniciadas por processos para executar de forma concorrente no sistema operacional. Threads apenas possuem uma pilha de memória e compartilham o heap do processo de onde foram iniciadas.

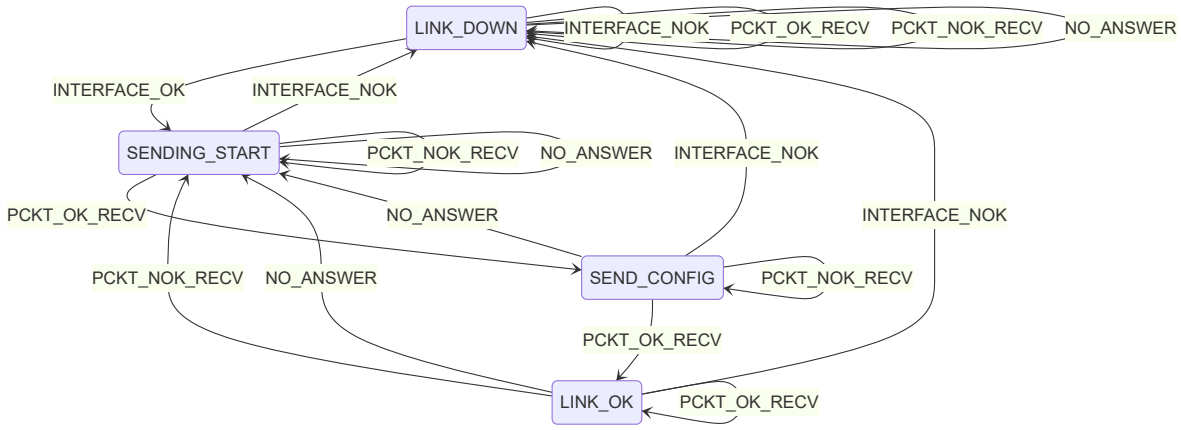
Processo	Thread
Custo computacional maior	Custo computacional menor
Possui sua própria memória	Compartilha memória com o processo de origem e outras threads dentro desse mesmo processo
Comunicação entre processos é lenta devido ao isolamento de memória	Comunicação entre threads é rápida devido à memória compartilhada
Mudança de contexto entre os processos é pesado devido a salvar a antiga e carregar a nova memória da pilha de cada processo	Mudança de contexto é simples devido ao compartilhamento de memória
Uma aplicação com diversos processos para seus componentes promovem uma melhor utilização da memória (nos casos de memória escassa). Pode-se designar baixa prioridade para processos inativos na aplicação. O processo ocioso pode ser mandado para um swap em disco. Isso torna os componentes da aplicação bastante responsivas	Com memória escassa, uma aplicação com múltiplas threads não ajudam no gerenciamento de memória

3. Implemente, usando sua linguagem de preferência, a seguinte máquina de estados, ela representa um protocolo de conexão entre 2 equipamentos (mestre/escravo):

Deve ser implementada uma função que recebe o estado atual e o evento associado. Deverá retornar o próximo estado e realizar as chamadas a funções que realizam as ações referentes a cada estado.

Estado/Evento/Próximo estado	Interface NOK	Interface OK	Pacote OK recebido	Pacote NOK recebido	Não recebeu resposta
1 - Link Down	1	2	1	1	1
2 - Enviando Start	1	-	3	2	2
3 - Start recebido envia Configuração	1	-	4	3	2
4 - Link OK - manda keepalive	1	-	4	2	2

O diagrama de estados referente ao apresentado acima, pode ser desenhado da seguinte forma:



É uma boa prática descrever tanto os eventos quanto os estados através de enumerations. Não somente ajuda na documentação do projeto em código através do header como também define nomes que vão ajudar a documentar o código posteriormente dentro das funções, tornando o código mais legível.

```
typedef enum {
    INTERFACE_NOK,
    INTERFACE_OK,
    PCKT_OK_RECV,
    PCKT_NOK_RECV,
    NO_ANSWER
} Event_TypeDef;

typedef enum {
    LINK_DOWN = 1,
    SENDING_START = 2,
    SEND_CONFIG = 3,
    LINK_OK = 4
```

```
} State_TypeDef;
```

Abaixo também declaro as funções que realizam as funções inerentes a cada estado:

```
void LinkDown();
void SendingStart();
void SendConfig();
void SendKeepAlive();
```

Outra vantagem de usar uma abordagem assim é limitar a imagem e o domínio da função posteriormente desenvolvida, se utilizando da forte tipagem da linguagem na declaração dos argumentos e retornos da função. Isso auxilia no desenvolvimento posterior de vários casos de testes e verificação do código do projeto.

```
State_TypeDef MyFunction(State_TypeDef curstate, Event_TypeDef evt)
{
    State_TypeDef nxtstate = LINK_DOWN;

    switch (curstate) {
        case LINK_DOWN:
            LinkDown();
            //if (evt == INTERFACE_NOK) nxtstate = LINK_DOWN;
            if (evt == INTERFACE_OK) nxtstate = SENDING_START;
            //if (evt == PCKT_OK_RECV) nxtstate = LINK_DOWN;
            //if (evt == PCKT_NOK_RECV) nxtstate = LINK_DOWN;
            //if (evt == NO_ANSWER) nxtstate = LINK_DOWN;
            break;
        case SENDING_START:
            SendingStart();
            //if (evt == INTERFACE_NOK) nxtstate = LINK_DOWN;
            if (evt == PCKT_OK_RECV) nxtstate = SEND_CONFIG;
            if (evt == PCKT_NOK_RECV) nxtstate = SENDING_START;
            if (evt == NO_ANSWER) nxtstate = SENDING_START;
            break;
        case SEND_CONFIG:
            SendConfig();
            //if (evt == INTERFACE_NOK) nxtstate = LINK_DOWN;
            if (evt == PCKT_OK_RECV) nxtstate = LINK_OK;
            if (evt == PCKT_NOK_RECV) nxtstate = SEND_CONFIG;
            if (evt == NO_ANSWER) nxtstate = SENDING_START;
            break;
        case LINK_OK:
            SendKeepAlive();
            //if (evt == INTERFACE_NOK) nxtstate = LINK_DOWN;
            if (evt == PCKT_OK_RECV) nxtstate = LINK_OK;
            if (evt == PCKT_NOK_RECV) nxtstate = SENDING_START;
            if (evt == NO_ANSWER) nxtstate = SENDING_START;
            break;
        default :
            break;
    }

    return nxtstate;
}
```

Como são poucos estados a serem implementados, um simples switch case resolve o problema. Cada estado atual executa sua função inerente e muda seu próximo estado a ser retornado de acordo com o evento recebido. As linhas comentadas não necessariamente precisam ser compiladas junto, mas ajudam a compor a interpretação do código de maneira bem clara, de acordo com o definido na tabela e no diagrama de estados.

Outra forma de implementar máquinas de estado mais robustas (no âmbito de minhas últimas experiências com máquinas de estado bem mais complexas) tem sido utilizando uma biblioteca bastante simples e eficiente para a construção de máquinas de estados finitos `fsm.h`, desenvolvida por Ankur Shrivastava, e disponível na sua própria página do Github.

O segundo grupo de testes apresenta programas com problemas que devem ser apontados e comentados. Entenda-se como problema: bug, código mal estruturado ou confuso. Sugestões de melhoria sempre são bem vindas.

4. Este programa exemplo deve inserir um cabeçalho de 12 caracteres em um pacote apontado por `packet`. Assume-se que a arquitetura é 32 bits e que o parâmetro `length` é o tamanho do pacote em bytes.

```
char *insert_header(int *packet, int *length)
{
    int i;
    char header[12] = { 0x80, 0x52, 0x66, 0x61, 0x77, 0x52 };
    /* os outros 6 são 0 */
    if (packet == NULL || length == NULL) {
        return NULL;
    }
    for (i=0; i < *length; i++)
        packet[i+3] = packet[i];
    for (i=0; i < 12; i++)
        packet[i] = header[i];
        *length += 12;
    return packet;
}
```

Se a arquitetura é 32 bits, `*length` sempre vai ser 4 vezes o tamanho total de `*packet` em bytes, uma vez que cada `int` possui 4 bytes.

Bugs encontrados

1. No primeiro `for` loop, o intuito é mover os dados para 3 posições posteriores para abrir espaço para o cabeçalho a ser inserido. Entretanto, da forma como está implementado, não será esse o resultado obtido. A partir da posição `packet[3]`, todos os dados serão carregados de forma incorreta pois foram sobrescritos nas primeiras iterações da estrutura de repetição. O que resulta na perda da informação do pacote original. Uma possível correção para a linha seria mover cada 4 bytes das últimas posições para a frente, sem perigo de sobrescrita. Por exemplo:

```
for (i = (*length)/4; i == 0; i--) packet[i+3] = packet[i];
```

2. No segundo `for` loop, o intuito é alocar o cabeçalho de 12 bytes no espaço gerado no começo do pacote, após a realocação da linha anterior. Entretanto, não será esse o resultado alcançado. Novamente, haverá perda da informação do pacote original. Isso porque não são 12 iterações a serem realizadas, mas 3. Além disso, o *cast* implícito de um tipo `char` para um tipo `int` seria um desastre para o pacote como um todo. Uma possível correção para a linha de código seria, por exemplo:

```
for (i = 0; i < 3; i++) packet[i] = ((int *) header)[i];
```

3. A função pode retornar um ponteiro para `int`, mas a definição da função resultará em um casting implícito para `char`. Não é bem um erro, mas o aconselhado seria retornar o ponteiro para `char` (ou definir um retorno de `int`) apontando assim para o mesmo tipo que a função possui em sua definição. Uma possível correção seria simplesmente mudar o tipo do ponteiro retornado pela função como, por exemplo:

```
int* insert_header(int* packet, int* length)
{
    ...
}
```

Dessa forma, a função corrigida e adaptada seria algo como:

```
//char* insert_header(int* packet, int* length)
int* insert_header(int* packet, int* length)
{
    int i;
    char header[12] = {0x80, 0x52, 0x66, 0x61, 0x77, 0x52};

    if (packet == NULL || length == NULL) return NULL;

    //for (i = 0; i < *length; i++) packet[i+3] = packet[i];
    for (i = (*length)/4; i == 0; i--) packet[i+3] = packet[i];

    //for (i = 0; i < 12; i++) packet[i] = header[i];
    for (i = 0; i < 3; i++) packet[i] = ((int *) header)[i];

    *length += 12;

    return packet;
}
```

5. A função a seguir funciona? É reentrante? (Porque?) O que a função faz?

```
void removeEntry(int key)
{
    int noPrev=1, noNext=1;
    list_entry *entry;

    for (entry = List_head; entry != NULL; entry = entry->next) {
        if (entry->key != key)
            break;
    }
```

```
    if (entry == NULL)
        return;
    if (entry->previous != NULL) {
        entry->previous->next = entry->next;
        noPrev=0;
    }
    if (entry->next != NULL) {
        entry->next->previous = entry->previous;
        noNext=0;
    }
    free (entry);
    if ((noPrev) && (noNext)) {
        List_head = NULL;
    } else if (noPrev) {
        List_head = entry->next;
    }
}
```

A função não funciona corretamente, possui um pequeno erro de implementação.

Ela também não é reentrante, pois utiliza de uma variável global (`List_head`) para acessar e modificar o ponteiro para o `head` da lista utilizada. Toda função que, quando interrompida, não tem seu processo seguramente autorizado a correr de onde parou, por definição, não pode ser chamada de reentrante. Ao alterar o valor de uma variável global dentro de seu escopo, se houver alguma interrupção no meio dela, corre-se o risco de essa variável global não conter o mesmo valor de antes da interrupção.

A função almeja varrer uma lista e tentar encontrar uma chave específica em algum de seus nodos. Se ela não encontra um nodo com a chave específica (passada como argumento), ela não faz nada mais e termina a função. Se encontrar, ela avalia se o nodo possui algum nodo prévio ou posterior e trata de alterar cada um de seus ponteiros para eliminar o nodo atual da lista. O próximo do anterior, se existir, aponta agora para o próximo do atual e o anterior do próximo []). Por fim, ela carrega a variável global que contém o `head` da lista com o respectivo novo `head` da lista.

Podemos pressupor que, em algum lugar do código, deve existir alguma definição de, no mínimo:

```
typedef struct list_entry{
    int key;
    struct list_entry *next;
    struct list_entry *previous;
} list_entry;

list_entry* List_head; // global
```

Bug encontrado

1. A chamada à função `free()` acontece antes de uma nova utilização da variável que teve seu espaço em memória liberado.

```
...
    free(entry);
    if ((noPrev) && (noNext)) {
        List_head = NULL;
    } else if (noPrev) {
        List_head = entry->next;
    }
...
```

Uma possível solução seria mover a linha da chamada à função `free()` para o final do escopo da função em questão.

```
...
    if ((noPrev) && (noNext)) {
        List_head = NULL;
    } else if (noPrev) {
        List_head = entry->next;
    }
    free(entry);
...
```

Isso garante que o acesso à variável `entry` ainda ocorra antes de liberar a memória.