

MA-CSEL

Conception système Embarqué Linux Mini-Projet

Kirill GOUNDIAEV & Tanguy DIETRICH

June 13, 2023



Contents

1	Introduction	3
2	Module Kernel	3
3	Daemon	3
3.1	IPC	3
3.2	Control des LED	4
3.3	Bouton	5
3.4	Concurrence	6
4	Conclusion	6

1 Introduction

Le projet est composé de 2 parties :

- Un module kernel qui permet de contrôler la fréquence du processeur, et de changer le mode de fonctionnement du module kernel. Et il met à disposition les informations sur la température du CPU.
- Un daemon qui permet d'afficher les informations sur l'écran, et de contrôler le module kernel.

2 Module Kernel

TODO

3 Daemon

Un Daemon est un programme qui s'exécute en arrière plan. Son objectif sera de gérer l'affichage des données sur l'écran, tel que la température, fréquence, et le mode de fonctionnement du module kernel. Pour faire cela, il dispose de 2 interfaces de contrôle :

- Les 3 boutons présents sur la carte pour augmenter/diminuer la fréquence, et changer le mode de fonctionnement.
- un IPC (Inter Process Communication) pour communiquer avec le module kernel. Dans notre cas, nous avons choisi d'utiliser un socket.

Nous allons réutiliser le code fourni dans le cours comme point de départ pour la création du daemon.

Nous avons réparti le code de notre programme dans différentes bibliothèques afin de le rendre plus lisible, voici les fichiers qui le composent, et leur contenu :

- *main.c*
Ce fichier contient le main de notre programme, il initialise le daemon et contient le thread de réception des messages du socket.
- *daemonfanlib.c/h*
Cette bibliothèque contient les fonctions nécessaires à l'initialisation du socket et de la création du daemon. La constante qui indique le port du socket.
- *gpio_utility.c/h*
Cette bibliothèque permet d'accéder au GPIO de la carte, ainsi qu'à l'information fournie par le module kernel. Le header contient diverses constantes qui permettent de définir les GPIO utilisés.
- *ssd1306.c/h*
Cette bibliothèque fournit les fonctions utiles à l'utilisation de l'écran. Cette bibliothèque nous a été fournie dans le cours.

3.1 IPC

La communication entre un processus externe et le daemon se fait par un socket. Ce qui permet de contrôler la carte depuis n'importe quel ordinateur connecté au même réseau. L'initialisation du socket se fait dans la fonction `void initSocket(int *mode, int *freq, pthread_t *thread_id, void* (*threadFunc)(void*))` qui est appelée dans la fonction *main*. La fonction prend un pointeur vers le mode, la fréquence, et l'id du thread, ainsi qu'un pointeur vers la fonction qui sera exécutée par le thread. Afin de traiter les données reçues. Le socket est initialisé avec l'adresse IP de la carte, et le port 8080.

Comme nous étions libres pour le choix du protocole de communication, nous avons choisi d'utiliser un protocole simple, qui permet de contrôler le mode et la fréquence du module kernel. Le protocole est composé de 2 commandes :

- MX : Permet de choisir le mode de fonctionnement du module kernel. X peut prendre 2 valeurs : 0 pour le mode manuel, et 1 pour le mode automatique.
 - FXXX : Permet de changer la fréquence de clignotement de la LED. XXX est un nombre entre 0 et 999.
- Par exemple envoyer "M0", puis "F5" aura pour effet de mettre le mode manuel, et de mettre la fréquence de clignotement à 5 Hz.

Voici le code du thread qui vas s'occuper de traiter les données reçues par le socket :

```

1 static void *threadSocket(void *arg)
2 {
3     int client_fd = 0;
4     char buffer[SOCKET_BUFFER_SIZE] = {0};
5     // get the parameters
6     socketParamThread *param = (socketParamThread*) arg;
7     int addresslen = sizeof(param->address);
8     //listen on the socket
9     if((client_fd = accept(param->server_fd, (struct sockaddr*)&param->address, ((socklen_t) &
10         addresslen))) < 0) {
11         syslog(LOG_ERR, "accept");
12         exit(EXIT_FAILURE);
13     }
14     syslog(LOG_INFO, "threadSocket started\n");
15     while(1) {
16         int valread = read(client_fd, buffer, SOCKET_BUFFER_SIZE);
17         if (valread == 0) {
18             syslog(LOG_INFO, "client disconnected\n");
19             close(client_fd);
20             client_fd = accept(param->server_fd, (struct sockaddr*)&param->address, ((socklen_t) &
21                 addresslen));
22         } else {
23             if (buffer[0] == 'M') {
24                 *param->mode = buffer[1] - '0';
25                 writeMode(*param->mode);
26                 if (*param->mode == 0) {
27                     writeFreq(*param->freq);
28                 }
29             } else if (buffer[0] == 'F') {
30                 *param->freq = atoi(&buffer[1]);
31                 writeFreq(*param->freq);
32             }
33             syslog(LOG_INFO, "received: %s\n", buffer);
34         }
35     }
36     free(param);
37     return NULL;
38 }

```

Le thread va attendre qu'un client se connecte, puis il va lire les données reçues, et les traiter. Si le client se déconnecte, le thread va attendre qu'un nouveau client se connecte. Le code ne permet pas à plusieurs clients de se connecter en même temps, mais il serait possible de le faire en créant un thread qui attend des connexions, et qui crée un nouveau thread pour chaque client qui se connecte.

3.2 Control des LED

L'accès à la LED par le daemon se fait par l'interface sysfs. Pour cela, nous avons créé une fonction qui permet d'initialiser les LED, et une fonction qui permet de choisir l'état de la LED rouge. Le prototype de la fonction d'écriture est : `void writeLed(int value)`. Voici le code d'initialisation des LED :

```

1 void initLeds()
2 {
3     int f = open(GPIO_UNEXPORT, O_WRONLY);
4     write(f, LED, strlen(LED));
5     close(f);
6
7     // export pin to sysfs
8     f = open(GPIO_EXPORT, O_WRONLY);

```

```

9  write(f, LED, strlen(LED));
10 close(f);
11
12 // config pin
13 f = open(GPIO_LED "/direction", O_WRONLY);
14 write(f, "out", 3);
15 close(f);
16 g_led_fd = open(GPIO_LED "/value", O_WRONLY);
17 syslog(LOG_INFO, "leds initialized\n");
18 }

```

Le code commence par exporter les LED, puis il configure les LED en sortie, en écrivant "out" dans le fichier direction. Enfin, il ouvre le fichier value, qui permet d'écrire dans la LED. Le descripteur de fichier est stocké dans la variable globale `g_led_fd`.

Pour finir un simple appel à la fonction `void writeLed(int value)` permet de changer l'état de la LED rouge.

3.3 Bouton

Pour réaliser cette partie du TP, nous avons en partie réutilisé le code que nous avons écrit pour le rendu précédent. Notre code utilise les `epoll` pour gérer les interruptions des boutons. afin de rendre le code main plus lisible, nous avons écrit une fonction `int initButtonsAndTimer()` qui permet d'initialiser les boutons, et un timer pour être utilisé avec les `epoll`. Cette fonction fait appel à la fonction `epoll_create1()`, `epoll_ctl()`, et `timerfd_create()` pour créer les `epoll` et le timer. puis elle retourne un descripteur de fichier qui permet de lire les événements sur les boutons, et le timer.

```

1 // init the buttons S1, S2 and S3, and the timer
2 epfd = initButtonsAndTimer();

```

L'attente des événements sur les boutons s'effectue dans le main avec la fonction `epoll_wait()`. Voici un extrait du code main, qui permet de lire les événements sur les boutons :

```

1 while (1) {
2     struct epoll_event event_arrived[NUM_EVENTS];
3     syslog(LOG_INFO, "waiting for event epoll\n");
4     writeLed(LED_OFF);
5     int nr = epoll_wait(epfd, event_arrived, NUM_EVENTS, -1);
6     syslog(LOG_INFO, "event arrived\n");
7     if (nr == -1) {
8         // printf("error epoll_wait: %s\n", strerror(errno));
9         syslog(LOG_ERR, "epoll_wait");
10        exit(EXIT_FAILURE);
11    }
12    for(int i = 0; i < nr; i++){
13        my_context *ctx = event_arrived[i].data.ptr;
14
15        switch (ctx->ev){
16            case EV_BTN_1: // increase frequency
17                syslog(LOG_INFO, "button 1 pressed\n");
18                if (ctx->first_done == 0){
19                    ctx->first_done = 1;
20                    break;
21                }
22                freq++;
23                writeFreq(freq); // will fail if in auto mode
24                writeLed(LED_ON);
25                break;
26                // ...

```

Dans le `epoll`, nous avons aussi ajouté le timer, qui permet de rafraîchir l'affichage de la température, et de la fréquence en mode automatique. Le timer est initialisé avec une période de 200ms, il est possible de le modifier dans le fichier `gpio_utility.h`, en modifiant la constante `DEFAULT_PERIOD`. Voici un extrait de la gestion de l'événement timer :

```

1 case EV_TIMER:
2     // read the actual mode
3     syslog(LOG_INFO, "timer expired\n");
4     updateTempCPU();
5     if (mode == 1) // if in auto mode
6     {

```

```
7 // read the actual freq
8 freq = readFreq();
9 // show it on the screen
10 writeLCDFreq(freq);
11 }
12 else // if in manual mode
13 {
14     // syslog(LOG_INFO, "manual mode\n");
15 }
16 break;
```

3.4 Concurrence

Comme notre programme est multithreadé (thread socket et thread main), il est possible d'avoir des problèmes de concurrence. Pour éviter cela nous avons ajouté un mutex ces commandes envoyées au LCD. Les variables de mode et de fréquence sont aussi partagées, et peuvent être modifiées par le thread socket, et par le main. Mais dans ce cas, les risques sont assez faibles, il faudrait qu'une requête soit envoyée au moment où quelqu'un écrit sur le socket. Exemple de protection dans la fonction *void writeLCDFreq(int freq)* :

```
1 void writeLCDFreq(int freq)
2 {
3     char str[32] = {0};
4     sprintf(str, "Freq: %03d Hz", freq);
5     pthread_mutex_lock(&g_mutex_lcd);
6     ssd1306_set_position(0, 4);
7     ssd1306_puts(str);
8     pthread_mutex_unlock(&g_mutex_lcd);
9 }
```

Sans cela deux threads essaieraient d'écrire en même temps sur le LCD, et le texte n'était pas affiché correctement.

4 Conclusion

VROOM VROOM ...