Chapter 6 Implementation details

When we read data from audio data relay, we have to make sure that correct data format is ready for reading. If the data stored in data relay is not complete or its format is wrong, we couldn't parse correct data information from it and fail to write data to sound device. For solving the problem, using semaphore and blocking signals are used in our system. Besides, we create a special mechanism, delay close, for another problem that has been happened.

6.1. Using Semaphore

A semaphore is a general mechanism for controlling access to resources. We know that there are different M-protocol formats for different system calls. If two system calls write their data to audio relay at the same time, the data would be written to the audio relay interactively. Then, the audio receiving daemon couldn’t write correct system call to /dev/dsp, because the data receiving from network is not fit our M-protocol format. We give a simple example: ioctl system call and write system call are writing data to the data relay in turn. The data of ioctl and write system call are alternate stored in data relay. This makes our audio-receiving daemon couldn’t correctly parse these data receiving from audio-sending daemon via network. Therefore, we have to prevent from simultaneously running the same code or accessing the same
data, and a semaphore is used for this situation.

Semaphores in Linux are defined in `<asm/semaphore.h>`. The following call is used when the semaphores are initialized.

```c
#include <asm/semaphore.h>
struct semaphore sema;
sema_init(&sema, 1);
```

A process wishing to enter a section of code protected by a semaphore must first ensure that no other process is already there. In Linux, we call `down` function.

```c
#include <asm/semaphore.h>
struct semaphore sema;
down(&sema);
```

The functions test the value of the semaphore to see if it is greater than 0; if so, they decrement the semaphore and return. If the semaphore is 0, the functions will sleep and try again after some other process.

A process that obtains a semaphore must always release it afterward. Linux uses function `up`. A simple call like

```c
#include <asm/semaphore.h>
struct semaphore sema;
up(&sema);
```

will increment the value of the semaphore and wake up any processes that are waiting for the semaphore to become available.

### 6.2. Forbid Signals

We solve the problem that two or more system calls simultaneously accessing the
same data by using semaphore. But it is interrupted by a signal when we are writing
system call, the data stored in data relay will not be complete M-protocol formats. In
the same way, our audio-receiving daemon couldn’t recognize the data format as well.
For the reason, we have to block signals during when we are writing data to data relay
or reading data from it.

Signal functions are defined in `<linux/signal.h>`, the system call
sys_rt_sigprocmask is used to block or unblock signals.

```c
#include <linux/signal.h>
struct sigset_t oldsig, newsig;
...
/* BLOCK signals */
if ((sys_rt_sigprocmask(SIG_BLOCK, &newsig, &oldsig, sizeof(newsig)) < 0)
    printk(" Blocking signals is failed \n");
...
/* Writing data to audio data relay or Reading data from it. */
...
/* Unblock signals */
if ((sys_rt_sigprocmask(SIG_UNBLOCK, &oldsig, NULL, sizeof(oldsig)) < 0)
    printk(" Unblocking signals is failed \n");
```

6.3. A Special Mechanism : Delay close

A process would be exited, it turns off resources, such as the file descriptor table.
We couldn’t take advantage of files including audio data relay then. So, the close
system calls could not be sent to receiving daemon during the time of terminating, we
delay the close system call sending until a new system call running.