Лекція 11
Комунікаційний домен Unix

Призначення адреси сокету

Структура адреси сокета комунікаційного домену Unix `struct sockaddr_un` визначена в `<sys/un.h>`. У цій структурі є принаймні такі поля: `sa_family_t sun_family` (номер комунікаційного домену, має бути `AF_UNIX`), `char sun_path[]` (шляхове ім'я файлу UDS у ФС). POSIX не визначає чи вказане шляхове ім'я має бути відносним або абсолютним, типова практика вказувати абсолютне шляхове ім'я файлу з кінцевим символом NUL у полі `sun_path`. Замість номера комунікаційного домену `AF_UNIX` у деяких системах можна вказувати `AF_LOCAL`, такий макрос у SUSv4 вже не визначений.

Для призначені адреси UDS використовується системний виклик `bind()` так само, як він використовується для призначення адреси TCP-сокету та UDP-сокету. Розмір структури адреси сокета в цьому системному виклику треба вказати як `sizeof(struct sockaddr_un)`. У деяких книгах рекомендують вказувати цей розмір використовуючи нестандартний макрос `SUN_LEN()` (він обчислює реальну довжину шляхового імені, вказану в полі `sun_path`). Таке рішення не відповідає стандарту, макрос `SUN_LEN()` не визначений у POSIX, а самостійно написаний такий макрос не має припускати, що поле `sun_path` у `struct sockaddr_un` є останнім.

Розмір поля `sun_path` у `struct sockaddr_un` зазвичай менший, ніж максимальна довжина шляхового імені в системі. Це пов’язано з першими реалізаціями комунікаційного домену Unix, де була вимога щоб об’єкт типу `struct sockaddr_un` можна було скопіювати в буфер ядра невеликого розміру. Тобто причина історична.

Після успішного призначення адреси UDS у ФС створюється відповідне ім’я файлу. Тип файлу UDS можна отримати за допомогою системного виклику `lstat()`. Значення поля `st_mode` у `struct stat` з маскою `S_IFMT` кодує тип об’єкта ядра. Значення `S_IFSOCK` позначає сокет, також можна використовувати макрос `S_ISSOCK()` для цього поля.

Для забирання файлу UDS з ФС використовується системний виклик `unlink()` або `unlinkat()`, оскільки файл UDS – це об’єкт ФС і до його імені можна застосовувати ті самі системні виклики, які можна застосовувати до імен файлів інших типів. Якщо шляхове ім’я, яке призначається файлу UDS, вже є у ФС, тоді `bind()` завершується з помилкою `EADDRINUSE`. Типовий сценарій програми, яка призначає адресу UDS такий: програма забирає відповідне ім’я файлу з ФС, призначає це ім’я UDS, виконує завдання, забирає відповідне ім’я файлу з

ФС. Правильніше спочатку перевірити тип файлу і якщо цей файл є сокет, тоді забирати його з ФС. Зрозуміло що після між викликами `lstat()` та `unlink*()` є часове вікно, але такий варіант правильніший ніж просто викликати `unlink*()` для імені файлі без перевірки типу файлу.

Для UDS, якому призначене адреса, можна застосовувати системний виклик `getsockname()`. Цей системний виклик у деяких системах повертає розмір структури адреси сокета менший, ніж `sizeof(struct sockaddr_un)`. Поле `sun_path` у `struct sockaddr_un` у цих системах є останнім і ці системи копіюють вміст поля `sun_path`, в якому реально є символи шляхового імені.

```
#include <sys/socket.h>
#include <sys/stat.h>
#include <sys/un.h>
#include <errno.h>
#include <stdint.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

int
main(void)
{
```

```
struct sockaddr_un srv_sun = { 0 };
struct sockaddr_un addr_sun;
struct stat statbuf;
const char *sockpath = "/tmp/socket";
socklen_t addrlen;
int sockfd;

if (strlen(sockpath) > sizeof(srv_sun.sun_path) - 1)
    exit_errx("too long socket pathname %s", sockpath);

if (lstat(sockpath, &statbuf) < 0) {
    if (errno != ENOENT)
        exit_err("lstat() for %s", sockpath);
} else {
    if (!S_ISSOCK(statbuf.st_mode))
        exit_errx("file %s is not a socket", sockpath);
    if (unlink(sockpath) < 0 && errno != ENOENT)
        exit_err("unlink() for %s", sockpath);
}

sockfd = socket(AF_UNIX, SOCK_STREAM, 0);
if (sockfd < 0)
    exit_err("socket()");

srv_sun.sun_family = AF_UNIX;
strncpy(srv_sun.sun_path, sockpath, sizeof(srv_sun.sun_path) - 1);
if (bind(sockfd, (struct sockaddr *)&srv_sun, sizeof(srv_sun)) < 0)
    exit_err("bind()");
```

```
    addrlen = sizeof(addr_sun);
    if (getsockname(sockfd, (struct sockaddr *)&addr_sun, &addrlen) < 0)
        exit_err("getsockname()");

    printf("sockpath %s, its length %zu\n", sockpath, strlen(sockpath));
    printf("sizeof(struct sockaddr_un) %zu, sizeof(sun_path) %zu\n",
        sizeof(addr_sun), sizeof(addr_sun.sun_path));
    printf("getsockname() returned address length %ju\n", (uintmax_t)addrlen);

    if (close(sockfd) < 0)
        exit_err("close()");

    if (unlink(sockpath) < 0)
        exit_err("unlink()");
}
```

Відправлення та отримання даних

Клієнт

```
#include <cstddef>
#include <iostream>
#include <vector>

#include <sys/socket.h>
#include <sys/un.h>
#include <errno.h>
#include <fcntl.h>
#include <stddef.h>
#include <string.h>
#include <unistd.h>

class File {
public:
    const char* name;
    int fd;

    File(const char* name)
        : name(name)
    {
    }
};
```

```
int
main(int argc, char *argv[])
{
    std::vector<File> files;
    struct msghdr msgh;
    struct sockaddr_un sun = {};
    struct iovec iovec;
    const char* socketpath;
    const char* control_end;
    struct cmsghdr* cmsg;
    struct cmsghdr* cmsg_next;
    socklen_t cmsg_len;
    size_t fdnum;
    ssize_t wrsz;
    int opt, sockfd;
    char ch;

    opterr = 0;
    while ((opt = getopt(argc, argv, "f:")) != -1) {
        switch (opt) {
        case 'f':
            try {
                files.push_back(File(optarg));
            } catch (...) {
                exit_err("std::vector::push_back()");
            }
            break;
        default:
```

```
            exit_errx("wrong option -", static_cast<char>(optopt));
        }
    }
    if (files.size() == 0)
        exit_errx("specify at least one -f option");
    if (optind != argc - 1)
        exit_errx("wrong number of command-line arguments");
    socketpath = argv[optind];

    if (strlen(socketpath) > sizeof(sun.sun_path) - 1)
        exit_errx("too long socket pathname");
    sun.sun_family = AF_UNIX;
    strncpy(sun.sun_path, socketpath, sizeof(sun.sun_path) - 1);

    for (auto& file : files) {
        file.fd = open(file.name, O_RDONLY);
        if (file.fd < 0)
            exit_err("open() for ", file.name);
    }

#if defined(CMSG_SPACE) && defined(CMSG_LEN)
    msgh.msg_controllen = (files.size() + 1) * CMSG_SPACE(sizeof(int));
    cmsg_len = CMSG_LEN(sizeof(int));
#else
    auto alignfunc = [](std::size_t x) {
        return (x + (alignof(std::max_align_t) - 1)) &
            ~(alignof(std::max_align_t) - 1);
    };
```

```
    const std::size_t data_space = alignfunc(sizeof(int));
    alignas(struct cmsghdr) char cmsgbuf[
        alignfunc(sizeof(struct cmsghdr)) + data_space + 1024];
    const std::ptrdiff_t cmsg_space = CMSG_DATA((struct cmsghdr*)cmsgbuf) -
        (unsigned char *)cmsgbuf;

    msgh.msg_controllen = (files.size() + 1) * (cmsg_space + data_space);
    cmsg_len = cmsg_space + sizeof(int);
#endif

    msgh.msg_control = malloc(msgh.msg_controllen);
    if (msgh.msg_control == nullptr)
        exit_err("malloc()");

    control_end = (char*)msgh.msg_control + msgh.msg_controllen;
    cmsg = CMSG_FIRSTHDR(&msgh);
    for (auto& file : files) {
        if (cmsg == nullptr)
            exit_errx("cmsg is nullptr");
        if ((size_t)(control_end - (char*)cmsg) < sizeof(*cmsg))
            exit_errx("no space in msg_control");
        cmsg->cmsg_len = cmsg_len;
        cmsg->cmsg_level = SOL_SOCKET;
        cmsg->cmsg_type = SCM_RIGHTS;
        cmsg_next = CMSG_NXTHDR(&msgh, cmsg);
        if (cmsg_next == nullptr)
            exit_errx("cmsg_next is nullptr");
        *(int *)CMSG_DATA(cmsg) = file.fd;
```

```
    std::cout << "Sending fd " << file.fd << '\n';
    cmsg = cmsg_next;
}
msgh.msg_controllen = (char*)cmsg - (char*)msgh.msg_control;

ch = 'x';
iovec.iov_base = &ch;
iovec.iov_len = 1;

msgh.msg_name = nullptr;
msgh.msg_namelen = 0;
msgh.msg_iov = &iovec;
msgh.msg_iovlen = 1;
msgh.msg_flags = 0;

sockfd = socket(AF_UNIX, SOCK_STREAM, 0);
if (sockfd < 0)
    exit_err("socket()");
for (;;) {
    if (connect(sockfd, (struct sockaddr*)&sun, sizeof(sun)) < 0) {
        if (errno != ECONNREFUSED)
            exit_err("connect()");
        warn_err("connect()");
        sleep(1);
    } else {
        break;
    }
}
```

```
    fdnum = files.size();
    wrsz = send(sockfd, &fdnum, sizeof(fdnum), MSG_NOSIGNAL);
    if (wrsz < 0)
        exit_err("send()");
    if (wrsz != sizeof(fdnum))
        exit_errx("short write");

    wrsz = sendmsg(sockfd, &msgh, MSG_NOSIGNAL);
    if (wrsz < 0)
        exit_err("sendmsg()");
    if (wrsz != sizeof(ch))
        exit_errx("short write");

    if (close(sockfd) < 0)
        exit_err("close()");
    for (auto& file : files)
        if (close(file.fd) < 0)
            exit_err("close()");
}
```

Сервер

```
#include <cstddef>
#include <iostream>
#include <vector>

#include <sys/socket.h>
```

```
#include <sys/stat.h>
#include <sys/un.h>
#include <errno.h>
#include <fcntl.h>
#include <stddef.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

int
main(int argc, char *argv[])
{
    std::vector<int> fds;
    struct sockaddr_un sun = {};
    struct stat statbuf;
    struct msghdr msgh;
    struct iovec iovec;
    const char* socketpath;
    struct cmsghdr* cmsg;
    size_t fdnum;
    ssize_t rdsz;
    int connfd, listfd;
    char ch, buf[10];

    if (optind != argc - 1)
        exit_errx("wrong number of command-line arguments");
    socketpath = argv[optind];
```

```
if (lstat(socketpath, &statbuf) < 0) {
    if (errno != ENOENT)
        exit_err("lstat() for ", socketpath);
} else {
    if (!S_ISSOCK(statbuf.st_mode))
        exit_errx("file ", socketpath, " is not a socket");
    if (unlink(socketpath) < 0 && errno != ENOENT)
        exit_err("unlink() for ", socketpath);
}

listfd = socket(AF_UNIX, SOCK_STREAM, 0);
if (listfd < 0)
    exit_err("socket()");
if (strlen(socketpath) > sizeof(sun.sun_path) - 1)
    exit_errx("too long socket pathname");
sun.sun_family = AF_UNIX;
strncpy(sun.sun_path, socketpath, sizeof(sun.sun_path) - 1);
if (bind(listfd, (struct sockaddr*)&sun, sizeof(sun)) < 0)
    exit_err("bind() for ", socketpath);
if (listen(listfd, 10) < 0)
    exit_err("listen()");

connfd = accept(listfd, NULL, NULL);
if (connfd < 0)
    exit_err("accept()");

rdsz = recv(connfd, &fdnum, sizeof(fdnum), MSG_WAITALL);
if (rdsz < 0)
```

```
        exit_err("recv()");
    if (rdsz != sizeof(fdnum))
        exit_errx("wrong size of message with file descriptor numbers");
    if (fdnum == 0)
        exit_errx("wrong number of file descriptors");
    std::cout << "Going to receive " << fdnum << " file descriptor(s)\n";

    iovec.iov_base = &ch;
    iovec.iov_len = 1;

    msgh.msg_name = nullptr;
    msgh.msg_namelen = 0;
    msgh.msg_iov = &iovec;
    msgh.msg_iovlen = 1;
    msgh.msg_flags = 0;

#if defined(CMSG_SPACE) && defined(CMSG_LEN)
    msgh.msg_controllen = fdnum * CMSG_SPACE(sizeof(int));
#else
    auto alignfunc = [](std::size_t x) {
        return (x + (alignof(std::max_align_t) - 1)) &
            ~(alignof(std::max_align_t) - 1);
    };
    const std::size_t data_space = alignfunc(sizeof(int));
    alignas(struct cmsghdr) char cmsgbuf[
        alignfunc(sizeof(struct cmsghdr)) + data_space + 1024];
    const std::ptrdiff_t cmsg_space = CMSG_DATA((struct cmsghdr*)cmsgbuf) -
        (unsigned char *)cmsgbuf;
```

```
    msgh.msg_controllen = fdnum * (cmsg_space + data_space);
#endif

    msgh.msg_control = malloc(msgh.msg_controllen);
    if (msgh.msg_control == nullptr)
        exit_err("malloc()");

    rdsz = recvmsg(connfd, &msgh, MSG_WAITALL);
    if (rdsz < 0)
        exit_err("recvmsg()");
    if (rdsz != 1)
        exit_errx("wrong size of message with ancillary data");
    if (msgh.msg_flags & MSG_CTRUNC)
        exit_errx("ancillary data was truncated");

    cmsg = CMSG_FIRSTHDR(&msgh);
    if (cmsg == nullptr)
        exit_errx("no ancillary data were received");
    do {
        if (cmsg->cmsg_level != SOL_SOCKET || cmsg->cmsg_type != SCM_RIGHTS)
            exit_errx("wrong cmsghdr level or type");
        size_t idx = ((cmsg->cmsg_len -
            (CMSG_DATA(cmsg) - (unsigned char *)cmsg)) / sizeof(int)) - 1;
        do {
            try {
                fds.push_back(((int *)CMSG_DATA(cmsg))[idx]);
            } catch (...) {
                exit_err("std::vector::push_back()");
```

```
            }
        } while (idx--);
        cmsg = CMSG_NXTHDR(&msgh, cmsg);
    } while (cmsg != nullptr);

    if (fds.size() != fdnum)
        exit_errx("received wrong number of file descriptors");

    for (auto& fd : fds) {
        if (lseek(fd, 0, SEEK_SET) < 0)
            exit_err("lseek()");
        rdsz = read(fd, buf, sizeof(buf));
        if (rdsz < 0)
            exit_err("read()");
        std::cout << "Received fd " << fd << ": <";
        for (auto it = std::begin(buf); it != std::begin(buf) + rdsz; ++it)
            std::cout << *it;
        std::cout << ">\n";
    }

    if (close(connfd) < 0 || close(listfd) < 0)
        exit_err("close()");

    if (unlink(socketpath) < 0)
        exit_err("unlink() for ", socketpath);
}
```