eBook Gratuit

APPRENEZ POSIX

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#posix

Table des matières

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: posix

It is an unofficial and free POSIX ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official POSIX.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Démarrer avec POSIX

Versions

Version	la norme	Année de sortie
POSIX.1	Norme IEEE 1003.1-1988	1988-01-01
POSIX.1b	Norme IEEE 1003.1b-1993	1993-01-01
POSIX.1c	Norme IEEE 1003.1c-1995	1995-01-01
POSIX.2	Norme IEEE 1003.2-1992	1992-01-01
POSIX.1-2001	Norme IEEE 1003.1-2001	2001-12-06
POSIX.1-2004	Norme IEEE 1003.1-2004	2004-01-01
POSIX.1-2008	IEEE Std 1003.1-2008 (alias " *Base Specifications, Issue 7* ")	2008-12-01
POSIX.1-2013	Norme IEEE 1003.1-2013	2013-04-19
POSIX.1-2016	Norme IEEE 1003.1-2016	2016-09-30

Examples

Qu'est ce que POSIX?

POSIX signifie " *Portable Operating System Interface* " et définit un ensemble de normes pour assurer la compatibilité entre les différentes plates-formes informatiques. La version actuelle de la norme est IEEE 1003.1 2016 et est accessible à partir de la spécification OpenGroup POSIX . Les versions précédentes incluent POSIX 2004 et POSIX 1997 . L'édition POSIX 2016 est essentiellement POSIX 2008 plus des errata (il y a eu une version POSIX 2013 également).

POSIX définit diverses interfaces d'outils, commandes et API pour les systèmes d'exploitation de type UNIX et autres.

Les éléments suivants sont considérés comme relevant de la normalisation POSIX:

- Interface système (fonctions, macros et variables externes)

- Interpréteur de commandes, ou Shell (l'utilitaire *sh*)
- Utilitaires (comme *plus* , *chat* , *ls*)

En dehors de la portée POSIX:

- Interfaces SGBD
- Interfaces Graphiques
- Portabilité du code binaire

Bonjour le monde

Un simple programme `Hello, World` sans vérification d'erreur:

```
#include <unistd.h> /* For write() and STDOUT_FILENO */
#include <stdlib.h> /* For EXIT_SUCCESS and EXIT_FAILURE */

int main(void) {
        char hello[] = "Hello, World\n";

        /* Attempt to write `hello` to standard output file */
        write(STDOUT_FILENO, hello, sizeof(hello) - 1);

        return EXIT_SUCCESS;
}
```

Et avec la vérification d'erreur:

```
#include <unistd.h> /* For write() and STDOUT_FILENO */
#include <stdlib.h> /* For EXIT_SUCCESS and EXIT_FAILURE */

int main(void) {
        char hello[] = "Hello, World\n";
        ssize_t ret = 0;

        /* Attempt to write `hello` to standard output file */
        ret = write(STDOUT_FILENO, hello, sizeof(hello) - 1);

        if (ret == -1) {
                /* write() failed. */
                return EXIT_FAILURE;
        } else if (ret != sizeof(hello) - 1) {
                /* Not all bytes of `hello` were written. */
                return EXIT_FAILURE;
        }

        return EXIT_SUCCESS;
}
```

Compiler et courir

Si le code ci-dessus (l'une ou l'autre version) est stocké dans le fichier `hello.c` , vous pouvez compiler le code dans un programme `hello` utilisant `c99` ou `make` . Par exemple, dans un mode strictement conforme à POSIX, vous pourriez en théorie compiler et exécuter le programme en utilisant:

```
$ make hello
c99 -o hello hello.c
$ ./hello
Hello, World
$
```

La plupart des implémentations de `make` utilisent un compilateur C différent (peut-être `cc` , peut-être `gcc` , `clang` , `xlc` ou un autre nom), et beaucoup utiliseront plus d'options pour le compilateur. De toute évidence, vous pouvez simplement taper la commande qui s'exécute directement sur la ligne de commande. `make`

Lire Démarrer avec POSIX en ligne: https://riptutorial.com/fr/posix/topic/4495/demarrer-avec-posix

Chapitre 2: Des filets

Examples

Filetage simple sans arguments

Cet exemple de base compte à des vitesses différentes sur deux threads que nous appelons sync (main) et async (new thread). Le thread principal compte à 15 à 1 Hz (1 s) tandis que le second compte à 10 à 0,5 Hz (2 s). Comme le thread principal se termine plus tôt, nous utilisons `pthread_join` pour le faire attendre à la fin de son async.

```
#include <pthread.h>
#include <unistd.h>
#include <errno.h>
#include <stdlib.h>
#include <stdio.h>

/* This is the function that will run in the new thread. */
void * async_counter(void * pv_unused) {
    int j = 0;
    while (j < 10) {
        printf("async_counter: %d\n", j);
        sleep(2);
        j++;
    }

    return NULL;
}

int main(void) {
    pthread_t async_counter_t;
    int i;
    /* Create the new thread with the default flags and without passing
     * any data to the function. */
    if (0 != (errno = pthread_create(&async_counter_t, NULL, async_counter, NULL))) {
        perror("pthread_create() failed");
        return EXIT_FAILURE;
    }

    i = 0;
    while (i < 15) {
        printf("sync_counter: %d\n", i);
        sleep(1);
        i++;
    }

    printf("Waiting for async counter to finish ...\n");

    if (0 != (errno = pthread_join(async_counter_t, NULL))) {
        perror("pthread_join() failed");
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

Copié ici: http://stackoverflow.com/documentation/c/3873/posix-threads/13405/simple-thread-without-arguments où il avait été initialement créé par M. Rubio-Roy .

Simple Mutex Usage

La bibliothèque de threads POSIX fournit l'implémentation de la primitive de mutex, utilisée pour l'exclusion mutuelle. Mutex est créé à l'aide de `pthread_mutex_init` et détruit à l'aide de `pthread_mutex_destroy` . L'obtention d'un mutex peut être effectuée à l'aide de `pthread_mutex_lock` ou `pthread_mutex_trylock` (selon le délai souhaité) et la libération d'un mutex s'effectue via `pthread_mutex_unlock` .

Un exemple simple utilisant un mutex pour sérialiser l'accès à la section critique suit. Tout d'abord, l'exemple sans utiliser un mutex. Notez que ce programme a *une course de données en* raison d'un accès non synchronisé à `global_resource` par les deux threads. Par conséquent, ce programme a un comportement indéfini :

```
#include <pthread.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>

// Global resource accessible to all threads
int global_resource;

// Threading routine which increments the resource 10 times and prints
// it after every increment
void* thread_inc (void* arg)
{
    for (int i = 0; i < 10; i++)
    {
        global_resource++;
        printf("Increment: %d\n", global_resource);
        // Make this thread slower, so the other one
        // can do more work
        sleep(1);
    }

    printf("Thread inc finished.\n");

    return NULL;
}

// Threading routine which decrements the resource 10 times and prints
// it after every decrement
void* thread_dec (void* arg)
{
    for (int i = 0; i < 10; i++)
    {
        global_resource--;
        printf("Decrement: %d\n", global_resource);
    }

    printf("Thread dec finished.\n");

    return NULL;
}
```

```
int main (int argc, char** argv)
{
    pthread_t threads[2];

    if (0 != (errno = pthread_create(&threads[0], NULL, thread_inc, NULL)))
    {
        perror("pthread_create() failed");
        return EXIT_FAILURE;
    }

    if (0 != (errno = pthread_create(&threads[1], NULL, thread_dec, NULL)))
    {
        perror("pthread_create() failed");
        return EXIT_FAILURE;
    }

    // Wait for threads to finish
    for (int i = 0; i < 2; i++)
    {
        if (0 != (errno = pthread_join(threads[i], NULL))) {
            perror("pthread_join() failed");
            return EXIT_FAILURE;
        }
    }

    return EXIT_SUCCESS;
}
```

Une sortie possible est:

```
Increment: 1
Decrement: 0
Decrement: -1
Decrement: -2
Decrement: -3
Decrement: -4
Decrement: -5
Decrement: -6
Decrement: -7
Decrement: -8
Decrement: -9
Thread dec finished.
Increment: -8
Increment: -7
Increment: -6
Increment: -5
Increment: -4
Increment: -3
Increment: -2
Increment: -1
Increment: 0
Thread inc finished.
```

Maintenant, si nous voulons synchroniser ces threads pour que nous voulions d'abord les incrémenter ou les décrémenter, puis les faire différemment, nous devons utiliser une primitive de synchronisation, telle que mutex:

```
#include <pthread.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>

// Global resource accessible to all threads
int global_resource;
// Mutex protecting the resource
pthread_mutex_t mutex;

// Threading routine which increments the resource 10 times and prints
// it after every increment
void* thread_inc (void* arg)
{
    // Pointer to mutex is passed as an argument
    pthread_mutex_t* mutex = arg;

    // Execute the following code without interrupts, all the way to the
    // point B
    if (0 != (errno = pthread_mutex_lock(mutex)))
    {
        perror("pthread_mutex_lock failed");
        exit(EXIT_FAILURE);
    }

    for (int i = 0; i < 10; i++)
    {
        global_resource++;
        printf("Increment: %d\n", global_resource);
        // Make this thread slower, so the other one
        // can do more work
        sleep(1);
    }

    printf("Thread inc finished.\n");

    // Point B:
    if (0 != (errno = pthread_mutex_unlock(mutex)))
    {
        perror("pthread_mutex_unlock failed");
        exit(EXIT_FAILURE);
    }

    return NULL;
}

// Threading routine which decrements the resource 10 times and prints
// it after every decrement
void* thread_dec (void* arg)
{
    // Pointer to mutex is passed as an argument
    pthread_mutex_t* mutex = arg;

    if (0 != (errno = pthread_mutex_lock(mutex)))
    {
        perror("pthread_mutex_lock failed");
        exit(EXIT_FAILURE);
    }

    for (int i = 0; i < 10; i++)
```

```
    {
        global_resource--;
        printf("Decrement: %d\n", global_resource);
    }

    printf("Thread dec finished.\n");

    // Point B:
    if (0 != (errno = pthread_mutex_unlock(mutex)))
    {
        perror("pthread_mutex_unlock failed");
        exit(EXIT_FAILURE);
    }

    return NULL;
}

int main (int argc, char** argv)
{
    pthread_t threads[2];
    pthread_mutex_t mutex;

    // Create a mutex with the default parameters
    if (0 != (errno = pthread_mutex_init(&mutex, NULL)))
    {
        perror("pthread_mutex_init() failed");
        return EXIT_FAILURE;
    }

    if (0 != (errno = pthread_create(&threads[0], NULL, thread_inc, &mutex)))
    {
        perror("pthread_create() failed");
        return EXIT_FAILURE;
    }

    if (0 != (errno = pthread_create(&threads[1], NULL, thread_dec, &mutex)))
    {
        perror("pthread_create() failed");
        return EXIT_FAILURE;
    }

    // Wait for threads to finish
    for (int i = 0; i < 2; i++)
    {
        if (0 != (errno = pthread_join(threads[i], NULL))) {
            perror("pthread_join() failed");
            return EXIT_FAILURE;
        }
    }

    // Both threads are guaranteed to be finished here, so we can safely
    // destroy the mutex
    if (0 != (errno = pthread_mutex_destroy(&mutex)))
    {
        perror("pthread_mutex_destroy() failed");
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

L'une des sorties possibles est

```
Increment: 1
Increment: 2
Increment: 3
Increment: 4
Increment: 5
Increment: 6
Increment: 7
Increment: 8
Increment: 9
Increment: 10
Thread inc finished.
Decrement: 9
Decrement: 8
Decrement: 7
Decrement: 6
Decrement: 5
Decrement: 4
Decrement: 3
Decrement: 2
Decrement: 1
Decrement: 0
Thread dec finished.
```

L'autre sortie possible serait inverse, au cas où `thread_dec` obtiendrait le mutex en premier.

Lire Des filets en ligne: https://riptutorial.com/fr/posix/topic/4508/des-filets

Chapitre 3: Des minuteries

Examples

POSIX Timer avec notification SIGEV_THREAD

Cet exemple illustre l' utilisation avec minuterie POSIX `CLOCK_REALTIME` horloge et `SIGEV_THREAD` méthode de notification.

```
#include <stdio.h>  /* for puts() */
#include <string.h> /* for memset() */
#include <unistd.h> /* for sleep() */
#include <stdlib.h> /* for EXIT_SUCCESS */

#include <signal.h> /* for `struct sigevent` and SIGEV_THREAD */
#include <time.h>   /* for timer_create(), `struct itimerspec`,
                     * timer_t and CLOCK_REALTIME
                     */

void thread_handler(union sigval sv) {
        char *s = sv.sival_ptr;

        /* Will print "5 seconds elapsed." */
        puts(s);
}

int main(void) {
        char info[] = "5 seconds elapsed.";
        timer_t timerid;
        struct sigevent sev;
        struct itimerspec trigger;

        /* Set all `sev` and `trigger` memory to 0 */
        memset(&sev, 0, sizeof(struct sigevent));
        memset(&trigger, 0, sizeof(struct itimerspec));

        /*
         * Set the notification method as SIGEV_THREAD:
         *
         * Upon timer expiration, `sigev_notify_function` (thread_handler()),
         * will be invoked as if it were the start function of a new thread.
         *
         */
        sev.sigev_notify = SIGEV_THREAD;
        sev.sigev_notify_function = &thread_handler;
        sev.sigev_value.sival_ptr = &info;

        /* Create the timer. In this example, CLOCK_REALTIME is used as the
         * clock, meaning that we're using a system-wide real-time clock for
         * this timer.
         */
        timer_create(CLOCK_REALTIME, &sev, &timerid);

        /* Timer expiration will occur withing 5 seconds after being armed
         * by timer_settime().
         */
```

```
        trigger.it_value.tv_sec = 5;

        /* Arm the timer. No flags are set and no old_value will be retrieved.
         */
        timer_settime(timerid, 0, &trigger, NULL);

        /* Wait 10 seconds under the main thread. In 5 seconds (when the
         * timer expires), a message will be printed to the standard output
         * by the newly created notification thread.
         */
        sleep(10);

        /* Delete (destroy) the timer */
        timer_delete(timerid);

        return EXIT_SUCCESS;
}
```

Lire Des minuteries en ligne: https://riptutorial.com/fr/posix/topic/4644/des-minuteries

Chapitre 4: Douilles

Examples

Serveur d'écho simultané TCP

Dans cet exemple, nous allons créer un serveur d'écho simple qui écoutera sur le port spécifié et sera capable de gérer les nouvelles connexions:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <time.h>

/**
  Connection handler - this will be executed in
  the new process, after forking, and it will read
  all the data from the socket, while available and
  to echo it on the local terminal.

  Params:
    sd = socket to the client
*/
#define BUF_SIZE (1024)

int echo_client(int sd)
{
    int result = 0;

    char buf[BUF_SIZE + 1] = {0};

    ssize_t n_read;
    while (0 < (n_read = read(sd, buf, BUF_SIZE)))
    {
        buf[n_read] = '\0';
        printf("%s\n", buf);
    }

    if (0 > n_read)
    {
      perror("read() failed");
      result = -1;
    }
    else
    {
      fprintf(stderr, "The other side orderly shut down the connection.\n");
    }

    close(sd);

    return result;
}
```

```
int main(void)
{
    // Create a listening socket
    int listening_socket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (listening_socket == -1)
    {
        perror("socket() failed");
        return EXIT_FAILURE;
    }

    // Bind it to port 15000.
    unsigned short listening_port = 15000;

    struct sockaddr_in addr = {0};
    addr.sin_family = AF_INET;
    addr.sin_port = htons(listening_port);

    socklen_t sock_len = sizeof(addr);

    if (0 > (bind(listening_socket, (const struct sockaddr*) &addr, sock_len)))
    {
        perror("bind() failed");
        return EXIT_FAILURE;
    }

    // Start listening
    if (0 > listen(listening_socket, 0))
    {
        perror("listen() failed");
        return EXIT_FAILURE;
    }

    // Accept new connections, fork the new process for handling
    // and handle the connection in the new process, while the parent
    // is waiting for another connection to arrive.
    int accepted_socket = 0;
    while (0 < (accepted_socket =
                  accept(listening_socket, (struct sockaddr*) &addr, &sock_len)))
    {
        pid_t pid_child = fork();

        if (0 > pid_child)
        {
            perror("fork() failed");
            return EXIT_FAILURE;
        }
        else if (0 == pid_child)
        {
            // inside the forked child here
            close(listening_socket); // The child does not need this any more.

            echo_client(accepted_socket);

            return EXIT_SUCCESS;
        }

        // Inside parent process, since file descriptors are reference
        // counted, we need to close the client socket
        close(accepted_socket);
```

```
    }

    if (0 > accepted_socket)
    {
        perror("accept() failed");
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

Activation de TCP keepalive côté serveur

Ceci est un exemple client-serveur. Le processus transforme et exécute le client dans le processus parent et le serveur dans le processus enfant:

- le client se connecte au serveur et attend que le serveur se ferme;
- le serveur accepte la connexion du client, active le keepalive et attend tout signal.

Keepalive est configuré en utilisant les options suivantes décrites dans les pages de manuel `socket(7)` et `tcp(7)` :

- `SO_KEEPALIVE` - permet d'envoyer des messages `SO_KEEPALIVE`
- `TCP_KEEPIDLE` - le temps (en secondes) `TCP_KEEPIDLE` la connexion doit rester inactive avant que TCP ne commence à envoyer des sondes keepalive
- `TCP_KEEPINTVL` - le temps (en secondes) entre les sondes individuelles de keepalive
- `TCP_KEEPCNT` - le nombre maximal de sondes keepalive que TCP doit envoyer avant de supprimer la connexion

Code source:

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/signal.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <netinet/tcp.h>

#define check(expr) if (!(expr)) { perror(#expr); kill(0, SIGTERM); }

void enable_keepalive(int sock) {
    int yes = 1;
    check(setsockopt(
        sock, SOL_SOCKET, SO_KEEPALIVE, &yes, sizeof(int)) != -1);

    int idle = 1;
    check(setsockopt(
        sock, IPPROTO_TCP, TCP_KEEPIDLE, &idle, sizeof(int)) != -1);

    int interval = 1;
    check(setsockopt(
        sock, IPPROTO_TCP, TCP_KEEPINTVL, &interval, sizeof(int)) != -1);
```

```
    int maxpkt = 10;
    check(setsockopt(
        sock, IPPROTO_TCP, TCP_KEEPCNT, &maxpkt, sizeof(int)) != -1);
}

int main(int argc, char** argv) {
    check(argc == 2);

    struct sockaddr_in addr;
    addr.sin_family = AF_INET;
    addr.sin_port = htons(12345);
    check(inet_pton(AF_INET, argv[1], &addr.sin_addr) != -1);

    int server = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    check(server != -1);

    int yes = 1;
    check(setsockopt(server, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)) != -1);

    check(bind(server, (struct sockaddr*)&addr, sizeof(addr)) != -1);
    check(listen(server, 1) != -1);

    if (fork() == 0) {
        int client = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
        check(client != -1);
        check(connect(client, (struct sockaddr*)&addr, sizeof(addr)) != -1);
        printf("connected\n");
        pause();
    }
    else {
        int client = accept(server, NULL, NULL);
        check(client != -1);
        enable_keepalive(client);
        printf("accepted\n");
        wait(NULL);
    }

    return 0;
}
```

Les paquets Keepalive peuvent être surveillés à l'aide de `tcpdump` .

Exemple d'utilisation:

```
$ ./a.out 127.0.0.1 &
[1] 14010
connected
accepted

$ tcpdump -n -c4 -ilo port 12345
dropped privs to tcpdump
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on lo, link-type EN10MB (Ethernet), capture size 262144 bytes
18:00:35.173892 IP 127.0.0.1.12345 > 127.0.0.1.60998: Flags [.], ack 510307430, win 342,
options [nop,nop,TS val 389745775 ecr 389745675], length 0
18:00:35.173903 IP 127.0.0.1.60998 > 127.0.0.1.12345: Flags [.], ack 1, win 342, options
[nop,nop,TS val 389745775 ecr 389745075], length 0
18:00:36.173886 IP 127.0.0.1.12345 > 127.0.0.1.60998: Flags [.], ack 1, win 342, options
[nop,nop,TS val 389745875 ecr 389745775], length 0
```

```
18:00:36.173898 IP 127.0.0.1.60998 > 127.0.0.1.12345: Flags [.], ack 1, win 342, options
[nop,nop,TS val 389745875 ecr 389745075], length 0
4 packets captured
8 packets received by filter
0 packets dropped by kernel
```

Serveur itératif de jour TCP

Ceci est un serveur itératif TCP de jour gardé aussi simple que possible.

```
#include <sys/types.h>   /* predefined types */
#include <unistd.h>      /* unix standard library */
#include <arpa/inet.h>   /* IP addresses conversion utilities */
#include <netinet/in.h>  /* sockaddr_in structure definition */
#include <sys/socket.h>  /* berkley socket library */
#include <stdio.h>       /* standard I/O library */
#include <string.h>      /* include to have memset */
#include <stdlib.h>      /* include to have exit */
#include <time.h>        /* time manipulation primitives */

#define MAXLINE 80
#define BACKLOG 10

int main(int argc, char *argv[])
{
    int list_fd, conn_fd;
    struct sockaddr_in serv_add;
    char buffer[MAXLINE];
    time_t timeval;

    /* socket creation third parameter should be IPPROTO_TCP but 0 is an
     * accepted value */
    list_fd = socket(AF_INET, SOCK_STREAM, 0);

    /* address initialization */
    memset(&serv_add, 0, sizeof(serv_add));        /* init the server address */
    serv_add.sin_family = AF_INET;                 /* address type is IPV4 */
    serv_add.sin_port = htons(13);                 /* daytime port is 13 */
    serv_add.sin_addr.s_addr = htonl(INADDR_ANY);  /* connect from anywhere */

    /* bind socket */
    bind(list_fd, (struct sockaddr *)&serv_add, sizeof(serv_add));

    /* listen on socket */
    listen(list_fd, BACKLOG);

    while (1)
    {
        /* accept connection */
        conn_fd = accept(list_fd, (struct sockaddr *) NULL, NULL);

        timeval = time(NULL);
        snprintf(buffer, sizeof(buffer), "%.24s\r\n", ctime(&timeval));

        write(conn_fd, buffer, strlen(buffer)); /* write daytime to client */

        close(conn_fd);
    }
```

```
    /* normal exit */
    close(list_fd);
    exit(0);
}
```

TCP Client de jour

Ceci est un client de jour TCP gardé aussi simple que possible.

```
#include <unistd.h>      /* unix standard library */
#include <arpa/inet.h>   /* IP addresses manipulation utilities */
#include <netinet/in.h>  /* sockaddr_in structure definition */
#include <sys/socket.h>  /* berkley socket library */
#include <stdio.h>       /* standard I/O library */
#include <string.h>      /* include to have memset*/
#include <stdlib.h>      /* include to have exit*/

#define MAXLINE 1024

int main(int argc, char *argv[])
{
    int sock_fd;
    int nread;
    struct sockaddr_in serv_add;
    char buffer[MAXLINE];

    /* socket creation third parameter should be IPPROTO_TCP but 0 is an
     * accepted value*/
    sock_fd = socket(AF_INET, SOCK_STREAM, 0);

    /* address initialization */
    memset(&serv_add, 0, sizeof(serv_add));     /* init the server address */
    serv_add.sin_family = AF_INET;              /* address type is IPV4 */
    serv_add.sin_port = htons(13);              /* daytime post is 13 */

    /* using inet_pton to build address */
    inet_pton(AF_INET, argv[1], &serv_add.sin_addr);

    /* connect to the server */
    connect(sock_fd, (struct sockaddr *)&serv_add, sizeof(serv_add));

    /* read daytime from server */
    while ((nread = read(sock_fd, buffer, MAXLINE)) > 0)
    {
        buffer[nread] = 0;
        if (fputs(buffer, stdout) == EOF)
        {
            perror("fputs error"); /* write daytime on stdout */
            return -1;
        }
    }

    close(sock_fd);
    exit(0);
}
```

Bases Socket

Il existe quatre types de sockets disponibles dans les API POSIX: TCP, UDP, UNIX et (éventuellement) RAW. Les sockets de domaine Unix peuvent agir comme des sockets de flux ou comme des sockets de datagrammes.

Certains types de terminaux:

1. `struct sockaddr` - type de point final universel. En règle générale, les autres types de points de terminaison concrets ne sont convertis dans ce type que dans les appels posix.
2. `struct sockaddr_in` - noeud final IPv4

```
struct sockaddr_in {
    sa_family_t sin_family;
    in_port_t sin_port;            /* Port number.  */
    struct in_addr sin_addr;       /* Internet address.  */
};
struct in_addr {
    in_addr_t s_addr;
};
```

3. `struct sockaddr_in6` - Point de terminaison IPv6

```
struct sockaddr_in6 {
    sa_family_t sin6_family;
    in_port_t sin6_port;    /* Transport layer port # */
    uint32_t sin6_flowinfo;    /* IPv6 flow information */
    struct in6_addr sin6_addr;    /* IPv6 address */
    uint32_t sin6_scope_id;    /* IPv6 scope-id */
};
```

4. `struct sockaddr_un` .

```
struct sockaddr_un {
    sa_family_t sun_family;               /* AF_UNIX */
    char        sun_path[108];            /* pathname */
};
```

Programme entier

```
#include <arpa/inet.h>
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <unistd.h>

#define DESIRED_ADDRESS "127.0.0.1"
#define DESIRED_PORT 3500
#define BUFSIZE 512

int main()
{
    // ADDRESS PART
    // MAIN PART
```

```
    close(sock);
    return EXIT_SUCCESS;
}
```

Création d'un noeud final IPv4

```
struct sockaddr_in addr = {0};
addr.sin_family = AF_INET;
addr.sin_port = htons(DESIRED_PORT); /*converts short to
                                       short with network byte order*/
addr.sin_addr.s_addr = inet_addr(DESIRED_ADDRESS);
```

Extrait de serveur TCP

```
int sock = socket (AF_INET, SOCK_STREAM, IPPROTO_TCP);
if (sock == -1) {
    perror("Socket creation error");
    return EXIT_FAILURE;
}

if (bind(sock, (struct sockaddr*) &addr, sizeof(addr)) == -1) {
    perror("Bind error");
    close(sock);
    return EXIT_FAILURE;
}

if (listen(sock, 1/*length of connections queue*/) == -1) {
    perror("Listen error");
    close(sock);
    return EXIT_FAILURE;
}

socklen_t socklen = sizeof addr;
int client_sock = accept(sock, &addr, &socklen); /* 2nd and 3rd argument may be NULL. */
if (client_sock == -1) {
    perror("Accept error");
    close(sock);
    return EXIT_FAILURE;
}

printf("Client with IP %s connected\n", inet_ntoa(addr.sin_addr));

char buf[BUFSIZE];
if (send(sock, "hello", 5, 0) == -1) {
    perror("Send error");
    close(client_sock);
    close(sock);
    return EXIT_FAILURE;
}

ssize_t readden = recv(sock, buf, BUFSIZE, 0);
if (readden < 0) {
    perror("Receive error");
    close(client_sock);
```

```
    close(sock);
    return EXIT_FAILURE;
}
else if (readden == 0) {
  fprintf(stderr, "Client orderly shut down the connection.\n");
}
else {readden > 0) {
    if (readden < BUFSIZE)
    {
      fprintf(stderr, "Received less bytes (%zd) then requested (%d).\n",
        readden, BUFSIZE);
    }

    write (STDOUT_FILENO, buf, readden);
}
```

Extrait de client TCP

```
int sock = socket (AF_INET, SOCK_STREAM, IPPROTO_TCP);
if (sock == -1) {
    perror("Socket creation error");
    return EXIT_FAILURE;
}
if (connect(sock, (struct sockaddr*) &addr, sizeof(addr)) == -1) {
    perror("Connection error");
    close(sock);
    return EXIT_FAILURE;
}

char buf[BUFSIZE];
if (send(sock, "hello", 5, 0); /*write may be also used*/ == -1) {
    perror("Send error");
    close(client_sock);
    close(sock);
    return EXIT_FAILURE;
}

ssize_t readden = recv(sock, buf, BUFSIZE, 0); /*read may be also used*/
if (readden < 0) {
    perror("Receive error");
    close(client_sock);
    close(sock);
    return EXIT_FAILURE;
}
else if (readden == 0)
{
    fprintf(stderr, "Client orderly shut down the connection.\n");
}
else /* if (readden > 0) */ {
    if (readden < BUFSIZE)
    {
      fprintf(stderr, "Received less bytes (%zd) then requested (%d).\n",
        readden, BUFSIZE);
    }

    write (STDOUT_FILENO, buf, readden);
}
```

Extrait de serveur UDP

```
int sock = socket (AF_INET, SOCK_DGRAM, IPPROTO_UDP);
if (sock == -1) {
    perror("Socket creation error");
    return EXIT_FAILURE;
}
if (bind(sock, (struct sockaddr*) &addr, sizeof(addr)) == -1) {
    perror("Bind error");
    close(sock);
    return EXIT_FAILURE;
}

char buf[BUFSIZE];
ssize_t readden = recvfrom(sock, buf, BUFSIZE, 0, &addr, sizeof(addr));
if (readden > 0) {
    printf("Client with IP %s sent datagram\n", inet_ntoa(addr.sin_addr));
    write (STDOUT_FILENO, buf, readden);
}
sendto(sock, "hello", 5, 0, &addr, sizeof(addr));
```

Accepter des connexions sur une prise de blocage

Programme AC qui souhaite accepter les connexions réseau (agir comme un « serveur ») doit d' abord créer un socket lié à l'adresse « INADDR_ANY » et appeler `listen` là - dessus. Ensuite, il peut appeler `accept` sur le socket du serveur pour bloquer jusqu'à ce qu'un client se connecte.

```
//Create the server socket
int servsock = socket(AF_INET, SOCK_STREAM, 0);
if(servsock < 0) perror("Failed to create a socket");

int enable = 1;
setsockopt(servsock, SOL_SOCKET, SO_REUSEADDR, (char*)&enable, sizeof(int));

//Bind to "any" address with a specific port to listen on that port
int port = 12345;
sockaddr_in serv_addr;
memset(&serv_addr, 0, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = INADDR_ANY;
serv_addr.sin_port = htons(port);

if(bind(servsock, (sockaddr*)&serv_addr, sizeof(serv_addr)) < 0)
    perror("Error binding to socket");

listen(servsock, 5);

//Accept a client
struct sockaddr_storage client_addr_info;
socklen_t len = sizeof client_addr_info;

int clientsock = accept(servsock, (struct sockaddr*)&client_addr_info, &len);

//Now you can call read, write, etc. on the client socket
```

La structure `sockaddr_storage` qui est transmise à `accept` peut être utilisée pour récupérer des informations sur le client connecté. Par exemple, voici comment déterminer l'adresse IP du client:

```
char client_ip_str[INET6_ADDRSTRLEN + 1];
if(client_addr_info.ss_family == AF_INET) {
    // Client has an IPv4 address
    struct sockaddr_in *s = (struct sockaddr_in *)&client_addr_info;
    inet_ntop(AF_INET, &s->sin_addr, client_ip_str, sizeof(client_ip_str));
} else {  // AF_INET6
    // Client has an IPv6 address
    struct sockaddr_in6 *s = (struct sockaddr_in6 *)&client_addr_info;
    inet_ntop(AF_INET6, &s->sin6_addr, client_ip_str, sizeof(client_ip_str));
}
```

Connexion à un hôte distant

POSIX.1-2008

Étant donné le nom d'un serveur en tant que chaîne, `char* servername` et un numéro de port, `int port` , le code suivant crée et ouvre un socket connecté à ce serveur. Le "nom" du serveur peut être un nom DNS, tel que "www.stackoverflow.com", ou une adresse IP en notation standard, telle que "192.30.253.113"; l'un ou l'autre des formats d'entrée est valide pour `gethostbyname` (qui a été supprimé de POSIX.1-2008).

```
char * server = "www.example.com";

int sock = socket(AF_INET, SOCK_STREAM, 0);
if(sock < 0)
  perror("Failed to create a socket");

hostent *server = gethostbyname(servername);
if (server == NULL)
  perror("Host lookup failed");

char server_ip_str[server->h_length];
inet_ntop(AF_INET, server->h_addr, server_ip_str, server->h_length);

sockaddr_in serv_addr;
memset(&serv_addr, 0, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
serv_addr.sin_port = htons(port);
memcpy(&serv_addr.sin_addr.s_addr, server->h_addr, server->h_length);

if (connect(sock, (sockaddr*)&serv_addr, sizeof(serv_addr)) < 0)
    perror("Failed to connect");

// Now you can call read, write, etc. on the socket.

close(sock);
```

Lire et écrire sur une prise de blocage

Même lorsque les sockets sont en mode "blocage", les opérations de `read` et d' `write` sur celles-ci ne lisent et ne écrivent pas nécessairement toutes les données disponibles pour être lues ou

écrites. Pour écrire un tampon entier dans un socket ou lire une quantité connue de données depuis un socket, il faut les appeler en boucle.

```
/*
 * Writes all bytes from buffer into sock. Returns true on success, false on failure.
 */
bool write_to_socket(int sock, const char* buffer, size_t size) {
    size_t total_bytes = 0;
    while(total_bytes < size) {
        ssize_t bytes_written = write(sock, buffer + total_bytes, size - total_bytes);
        if(bytes_written >= 0) {
            total_bytes += bytes_written;
        } else if(bytes_written == -1 && errno != EINTR) {
            return false;
        }
    }
    return true;
}
```

```
/*
 * Reads size bytes from sock into buffer. Returns true on success; false if
 * the socket returns EOF before size bytes can be read, or if there is an
 * error while reading.
 */
bool read_from_socket(int sock, char* buffer, size_t size) {
    size_t total_bytes = 0;
    while(total_bytes < size) {
        ssize_t new_bytes = read(sock, buffer + total_bytes, size - total_bytes);
        if(new_bytes > 0) {
            total_bytes += new_bytes;
        } else if(new_bytes == 0 || (new_bytes == -1 && errno != EINTR)) {
            return false;
        }
    }
    return true;
}
```

Lire Douilles en ligne: https://riptutorial.com/fr/posix/topic/4706/douilles

Chapitre 5: Les signaux

Syntaxe

- alarme non signée (secondes non signées);
- int kill (pid_t pid, int sig);

Paramètres

Fonction, paramètre (s), valeur de retour	La description
`alarm()`	nom de la fonction
`unsigned seconds`	Secondes pour déclencher une alarme ou 0 pour annuler toute alarme en attente
> = 0	0 si aucune autre alarme n'était en attente, sinon le nombre de secondes pendant lesquelles l'alarme en attente était encore ouverte. Cette fonction n'échouera pas.
-	-
`kill()`	nom de la fonction
`pid_t pid`	.
`int sig`	0 ou ID du signal
0, -1	En cas de succès, 0 est renvoyé, -1 en cas d'échec avec la définition d' `errno` sur `EINVAL` , `EPERM` ou `ESRCH` .

Examples

Relever SIGALARM avec l'action par défaut

En utilisant l' `alarm` , l'utilisateur peut programmer le signal `SIGALARM` pour qu'il soit levé après l'intervalle spécifié. Si l'utilisateur n'a pas bloqué, ignoré ou spécifié un gestionnaire de signal explicite pour ce signal, l'action par défaut pour ce signal sera effectuée à l'arrivée. L'action par défaut par spécification pour `SIGALARM` consiste à terminer le processus:

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
```

```
int main (int argc, char** argv)
{
    printf("Hello!\n");

    // Set the alarm for five second
    alarm(5); // Per POSIX, this cannot fail

    // Now sleep for 15 seconds
    for (int i = 1; i <= 15; i++)
    {
        printf("%d\n", i);
        sleep(1);
    }

    // And print the message before successful exit
    printf("Goodbye!\n");

    return EXIT_SUCCESS;
}
```

Cela produit:

```
Hello!
1
2
3
4
5
[2]    35086 alarm      ./a.out
```

Réglage du gestionnaire de signaux à l'aide de sigaction et de signaux de relance à l'aide de raise

Pour qu'un programme réagisse à un certain signal, autre qu'une action par défaut, un gestionnaire de signal personnalisé peut être installé à l'aide de `sigaction` . `sigaction` reçoit trois arguments - signal sur lequel agir, pointeur sur la structure `sigaction_t` qui, sinon `NULL` , décrit le nouveau comportement et le pointeur sur `sigaction_t` qui, si `NULL` ne sera pas rempli avec l'ancien comportement (on peut donc le restaurer). Augmenter les signaux dans le même processus peut être fait avec la méthode `raise` . Si plus de contrôle est nécessaire (pour envoyer le signal à un autre processus, `kill` ou `pthread_kill` peut être utilisé, qui accepte l'identifiant du processus de destination ou l'identifiant du thread).

```
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Signals are numbered from 1, signal 0 doesn't exist
volatile sig_atomic_t last_received_signal = 0;

// Signal handler, will set the global variable
// to indicate what is the last signal received.
// There should be as less work as possible inside
// signal handler routine, and one must take care only
```

```
// to call reentrant functions (in case of signal arriving
// while program is already executing same function)
void signal_catcher(int signo, siginfo_t *info, void *context)
{
    last_received_signal = info->si_signo;
}

int main (int argc, char** argv)
{
    // Setup a signal handler for SIGUSR1 and SIGUSR2
    struct sigaction act;
    memset(&act, 0, sizeof act);

    // sigact structure holding old configuration
    // (will be filled by sigaction):
    struct sigaction old_1;
    memset(&old_1, 0, sizeof old_1);
    struct sigaction old_2;
    memset(&old_2, 0, sizeof old_2);

    act.sa_sigaction = signal_catcher;
    // When passing sa_sigaction, SA_SIGINFO flag
    // must be specified. Otherwise, function pointed
    // by act.sa_handler will be invoked
    act.sa_flags = SA_SIGINFO;

    if (0 != sigaction(SIGUSR1, &act, &old_1))
    {
        perror("sigaction () failed installing SIGUSR1 handler");
        return EXIT_FAILURE;
    }

    if (0 != sigaction(SIGUSR2, &act, &old_2))
    {
        perror("sigaction() failed installing SIGUSR2 handler");
        return EXIT_FAILURE;
    }

    // Main body of "work" during which two signals
    // will be raised, after 5 and 10 seconds, and which
    // will print last received signal
    for (int i = 1; i <= 15; i++)
    {
        if (i == 5)
        {
            if (0 != raise(SIGUSR1))
            {
                perror("Can't raise SIGUSR1");
                return EXIT_FAILURE;
            }
        }

        if (i == 10)
        {
            if (0 != raise(SIGUSR2))
            {
                perror("Can't raise SIGUSR2");
                return EXIT_FAILURE;
            }
        }
```

```
        printf("Tick #%d, last caught signal: %d\n",
            i, last_received_signal);

        sleep(1);
    }

    // Restore old signal handlers
    if (0 != sigaction(SIGUSR1, &old_1, NULL))
    {
        perror("sigaction() failed restoring SIGUSR1 handler");
        return EXIT_FAILURE;
    }

    if (0 != sigaction(SIGUSR2, &old_2, NULL))
    {
        perror("sigaction() failed restoring SIGUSR2 handler");
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

Cela produit:

```
Tick #1, last caught signal: 0
Tick #2, last caught signal: 0
Tick #3, last caught signal: 0
Tick #4, last caught signal: 0
Tick #5, last caught signal: 30
Tick #6, last caught signal: 30
Tick #7, last caught signal: 30
Tick #8, last caught signal: 30
Tick #9, last caught signal: 30
Tick #10, last caught signal: 31
Tick #11, last caught signal: 31
Tick #12, last caught signal: 31
Tick #13, last caught signal: 31
Tick #14, last caught signal: 31
Tick #15, last caught signal: 31
```

Un processus qui se suicide en utilisant kill ()

Un processus peut (essayer de) envoyer un signal à tout autre processus en utilisant la fonction `kill()` .

Pour ce faire, le processus d'envoi doit connaître le PID du processus de réception. Comme, sans introduire de race, un processus ne peut être sûr que de son propre PID (et des PID de ses enfants), l'exemple le plus simple pour démontrer l'utilisation de `kill()` consiste à envoyer un signal à lui-même.

Ci-dessous, un exemple de processus initiant sa propre terminaison en s'envoyant un kill-signal (`SIGKILL`):

```
#define _POSIX_C_SOURCE 1
```

```
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>
#include <stdio.h>


int main (void)
{
  pid_t pid = getpid(); /* Get my iown process ID. */

  kill(pid, SIGKILL); /* Send myself a KILL signal. */

  puts("Signal delivery initiated.");  /* Although not guaranteed,
                                  practically the program never gets here. */

  pause(); /* Wait to die. */

  puts("This never gets printed.");
}
```

Sortie:

```
Killed
```

(... ou similaire, selon l'implémentation)

Gérer SIGPIPE généré par write () d'une manière thread-safe

Lorsque `write()` est appelé pour un socket ou un socket de flux nommé ou non nommé dont la fin de lecture est fermée, deux choses se produisent:

POSIX.1-2001

1. `SIGPIPE` signal `SIGPIPE` est envoyé au *processus* appelé `write()`

POSIX.1-2004

1. `SIGPIPE` signal `SIGPIPE` est envoyé au *thread* qui a appelé `write()`
2. `EPIPE` erreur `EPIPE` est renvoyée par `write()`

Il y a plusieurs façons de traiter avec `SIGPIPE` :

- Pour les sockets, `SIGPIPE` peut être désactivé en définissant des options spécifiques à la plate-forme telles que `MSG_NOSIGNAL` sous Linux et `SO_NOSIGPIPE` dans BSD (fonctionne uniquement pour l' `send` , mais pas pour l' `write`). Ce n'est pas portable.
- Pour les FIFO (canaux nommés), `SIGPIPE` ne sera pas généré si writer utilise `O_RDWR` au lieu de `O_WRONLY` , afin que la fin de la lecture soit toujours ouverte. Cependant, cela désactive `EPIPE` aussi.

- Nous pouvons ignorer `SIGPIPE` ou définir le gestionnaire global. C'est une bonne solution, mais ce n'est pas acceptable si vous ne contrôlez pas toute l'application (par exemple, vous écrivez une bibliothèque).

- Avec les versions récentes de POSIX, nous pouvons utiliser le fait que `SIGPIPE` est envoyé au thread qui a appelé `write()` et le gère en utilisant la technique de traitement du signal synchrone.

Le code ci-dessous illustre la gestion `SIGPIPE` sans risque pour POSIX.1-2004 et les versions ultérieures.

C'est inspiré par ce post :

- Tout d'abord, ajoutez `SIGPIPE` au signal de masque du thread en cours en utilisant `pthread_sigmask()` .
- Vérifiez si `SIGPIPE` est déjà en attente à l'aide de `sigpending()` .
- Appelez `write()` . Si la fin de la lecture est fermée, `SIGPIPE` sera ajouté au masque de signaux en attente et `EPIPE` sera renvoyé.
- Si `write()` renvoyé `EPIPE` et que `SIGPIPE` n'était pas déjà en attente avant `write()` , supprimez-le du masque de signaux en attente à l'aide de `sigtimedwait()` .
- Restaurer le masque de signal original en utilisant `pthread_sigmask()` .

Code source:

```
#include <unistd.h>
#include <time.h>
#include <errno.h>
#include <sys/signal.h>

ssize_t safe_write(int fd, const void* buf, size_t bufsz)
{
    sigset_t sig_block, sig_restore, sig_pending;

    sigemptyset(&sig_block);
    sigaddset(&sig_block, SIGPIPE);

    /* Block SIGPIPE for this thread.
     *
     * This works since kernel sends SIGPIPE to the thread that called write(),
     * not to the whole process.
     */
    if (pthread_sigmask(SIG_BLOCK, &sig_block, &sig_restore) != 0) {
        return -1;
    }

    /* Check if SIGPIPE is already pending.
     */
    int sigpipe_pending = -1;
    if (sigpending(&sig_pending) != -1) {
        sigpipe_pending = sigismember(&sig_pending, SIGPIPE);
    }

    if (sigpipe_pending == -1) {
        pthread_sigmask(SIG_SETMASK, &sig_restore, NULL);
```

```
        return -1;
    }

    ssize_t ret;
    while ((ret = write(fd, buf, bufsz)) == -1) {
        if (errno != EINTR)
            break;
    }

    /* Fetch generated SIGPIPE if write() failed with EPIPE.
     *
     * However, if SIGPIPE was already pending before calling write(), it was
     * also generated and blocked by caller, and caller may expect that it can
     * fetch it later. Since signals are not queued, we don't fetch it in this
     * case.
     */
    if (ret == -1 && errno == EPIPE && sigpipe_pending == 0) {
        struct timespec ts;
        ts.tv_sec = 0;
        ts.tv_nsec = 0;

        int sig;
        while ((sig = sigtimedwait(&sig_block, 0, &ts)) == -1) {
            if (errno != EINTR)
                break;
        }
    }

    pthread_sigmask(SIG_SETMASK, &sig_restore, NULL);
    return ret;
}
```

Lire Les signaux en ligne: https://riptutorial.com/fr/posix/topic/4532/les-signaux

Chapitre 6: Multiplexage d'entrée / sortie

Introduction

IO peut être bloquant / non bloquant et synchrone / asynchrone. L'API POSIX fournit une API de blocage synchrone (par exemple, appels classiques en lecture, écriture, envoi, recv), API synchrone non bloquante (mêmes fonctions, descripteurs de fichiers ouverts avec les appels `O_NONBLOCK` et IO-multiplexage) et API asynchrone (fonctions commençant par `aio_`).

L'API synchrone est généralement utilisée avec le style "un thread / processus par fd". C'est terrible pour les ressources. L'API non bloquante permet de fonctionner avec un ensemble de fds dans un thread.

Examples

Sondage

Dans cet exemple, nous créons une paire de sockets connectés et envoyons 4 chaînes de l'un à l'autre et imprimons les chaînes reçues sur la console. Notez que le nombre de fois que nous appellerons send peut ne pas être égal au nombre de fois que nous appelons recv

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <poll.h>

#define BUFSIZE 512

int main()
{
    #define CKERR(msg) {if(ret < 0) { perror(msg); \
        close(sockp[0]); close(sockp[1]); exit(EXIT_FAILURE); } }
    const char* strs_to_write[] = {"hello ", "from ", "other ", "side "};

    int sockp[2] = {-1, -1};
    ssize_t ret = socketpair (AF_UNIX, SOCK_STREAM, 0, sockp);
    CKERR("Socket pair creation error")

    struct pollfd pfds[2];
    for(int i=0; i<2; ++i) {
        pfds[i] = (struct pollfd){sockp[i], POLLIN|POLLOUT, 0};
        fcntl(sockp[i], F_SETFL|O_NONBLOCK); // nonblocking fds are
                // literally mandatory for IO multiplexing; non-portable
    }
    char buf[BUFSIZE];

    size_t snt = 0, msgs = sizeof(strs_to_write)/sizeof(char*);
    while(1) {
```

```
        int ret = poll(pfds,
            2 /*length of pollfd array*/,
            5 /*milliseconds to wait*/);
        CKERR("Poll error")

        if (pfds[0].revents & POLLOUT && snt < msgs) {
            // Checking POLLOUT before writing to ensure there is space
            // available in socket's kernel buffer to write, otherwise we
            // may face EWOULDBLOCK / EAGAIN error
            ssize_t ret = send(sockp[0], strs_to_write[snt], strlen(strs_to_write[snt]), 0);
            if(++snt >= msgs)
                close(sockp[0]);
            CKERR("send error")
            if (ret == 0) {
                puts("Connection closed");
                break;
            }
            if (ret > 0) {
                // assuming that all bytes were written
                // if ret != %sent bytes number%, send other bytes later
            }
        }
        if (pfds[1].revents & POLLIN) {
            // There is something to read
            ssize_t ret = recv(sockp[1], buf, BUFSIZE, 0);
            CKERR("receive error")
            if (ret == 0) {
                puts("Connection closed");
                break;
            }
            if (ret > 0) {
                printf("received str: %.*s\n", (int)ret, buf);
            }
        }

    }
    close(sockp[1]);
    return EXIT_SUCCESS;
}
```

Sélectionner

Select est un autre moyen de faire du multiplexage d'E / S. L'un de ses avantages est l'existence d'une API WinSock. De plus, sous Linux, select () modifie le délai d'expiration pour refléter le temps écoulé; la plupart des autres implémentations ne le font pas. (POSIX.1 autorise les deux comportements)

Les deux sondages et sélections ont des alternatives ppoll et pselect, qui permettent de gérer les signaux entrants pendant l'attente d'un événement. Et les deux deviennent lents avec une quantité énorme de descripteurs de fichiers (cent et plus), il serait donc judicieux de choisir un appel spécifique à la plate-forme, par exemple `epoll` sur Linux et `kqueue` sur FreeBSD. Ou passer à une API asynchrone (POSIX `aio` par exemple, ou quelque chose de spécifique comme les ports IO Completion).

Sélectionnez l'appel a le prototype suivant:

```
int select(int nfds, fd_set *readfds, fd_set *writefds,
       fd_set *exceptfds, struct timeval *timeout);
```

`fd_set` est un tableau de descripteurs de fichiers `fd_set` ,

`nfds` est le nombre maximal de tous les descripteurs de fichiers de l'ensemble + 1.

Extrait de travail avec select:

```
fd_set active_fd_set, read_fd_set;
FD_ZERO (&active_fd_set); // set fd_set to zeros
FD_SET (sock, &active_fd_set); // add sock to the set
// # define FD_SETSIZE sock + 1
while (1) {
    /* Block until input arrives on one or more active sockets. */
    read_fd_set = active_fd_set; // read_fd_set gets overriden each time
    if (select (FD_SETSIZE, &read_fd_set, NULL, NULL, NULL) < 0) {
        // handle error
    }
    // Service all file descriptors with input pending.
    for (i = 0; i < FD_SETSIZE; ++i) {
        if (FD_ISSET (i, &read_fd_set)) {
            // there is data for i
    }
}
```

Notez que sur la plupart des implemetations POSIX, les descripteurs de fichiers associés aux fichiers sur le disque sont bloquants. Donc, écrire dans un fichier, même si ce fichier était défini dans `writefds` , bloquerait jusqu'à ce que tous les octets ne soient pas vidés sur le disque

Lire Multiplexage d'entrée / sortie en ligne: https://riptutorial.com/fr/posix/topic/6547/multiplexage-d-entree---sortie

Chapitre 7: Pipes

Introduction

Les tuyaux constituent un mécanisme de communication unidirectionnelle interprocessus ou interthread dans le cadre d'une seule machine. Logiquement, un tube se compose de deux extrémités connectées, une sur laquelle des données peuvent être écrites et une autre à partir de laquelle les données peuvent être lues ultérieurement, avec un tampon de données entre lesquelles des écritures et des lectures ne doivent pas nécessairement être synchrones. Les tuyaux doivent être distingués des *conduites en* coque, qui sont une application de tuyaux.

Examples

Création et utilisation de base

Les pipes anonymes, ou simplement pipes, sont des objets gérés par le noyau exposés à des processus sous la forme d'une paire de descripteurs de fichiers, un pour le terminal de lecture et un pour le terminus d'écriture. Ils sont créés via la fonction `pipe` (2):

```
int pipefds[2];
int result;

result = pipe(pipefds);
```

En cas de succès, `pipe()` enregistre le descripteur de l'extrémité de lecture du tube à l'index 0 du tableau fourni, et le descripteur de la fin de l'écriture à l'index 1; ces indices sont analogues aux numéros de descripteur de fichier classiques pour les flux standard.

Après avoir créé un tube, on utilise les fonctions d'E / S POSIX pour écrire à la fin de l'écriture ou à la lecture:

Processus 1:

```
ssize_t bytes_written = write(pipefds[1], "Hello, World!", 14);
```

Processus 2:

```
char buffer[256];
ssize_t bytes_read = read(pipefds[0], buffer, sizeof(buffer));
```

Alternativement, on peut utiliser `fdopen()` pour envelopper une des deux extrémités de tube dans une structure `FILE` pour les utiliser avec les fonctions de base:

```
FILE *write_end = fdopen(pipefds[1]);

if (write_end) {
```

```
    fputs("Hello, World!");
}
```

Les tuyaux ont des tampons d'E / S finis, et les écritures ordinaires sur les tuyaux avec des tampons complets seront bloquées. Les tuyaux ne sont donc pas un mécanisme sûr pour qu'un thread puisse communiquer avec lui-même, comme si le thread qui écrit dans un tube est également le seul à en lire, puis dès qu'un écriture bloque ce thread est bloqué.

Un canal persiste tant qu'un processus a une description de fichier ouverte pour l'une des extrémités du tuyau. La fonction `close` (2) peut être utilisée pour fermer une extrémité de tuyau représentée par un descripteur de fichier, et `fclose` (3) peut être utilisé pour fermer une extrémité de tuyau via un `FILE` entouré de celui-ci.

Établissement d'un canal vers un processus enfant

Les descripteurs de fichiers et les objets `FILE` sont des ressources par processus qui ne peuvent pas être échangées entre processus via des E / S ordinaires. Par conséquent, pour que deux processus distincts puissent communiquer via un canal anonyme, un ou les deux processus participants doivent hériter d'un tube ouvert du processus qui a créé le canal, qui doit donc être le processus parent ou un ancêtre plus distant. Le cas le plus simple est celui dans lequel un processus parent veut communiquer avec un processus enfant.

Étant donné que le processus enfant doit hériter de la description de fichier ouverte requise de son parent, le canal doit être créé en premier. Le parent fourche alors. Normalement, chaque processus sera strictement un lecteur ou strictement un écrivain; dans ce cas, chacun devrait fermer l'extrémité du tuyau qu'il n'a pas l'intention d'utiliser.

```
void demo() {
    int pipefds[2];
    pid_t pid;

    // Create the pipe
    if (pipe(pipefds)) {
        // error - abort ...
    }

    switch (pid = fork()) {
        case -1:
            // error - abort ...
            break;
        case 0:   /* child */
            close(pipefds[0]);
            write(pipefds[1], "Goodbye, and thanks for all the fish!", 37);
            exit(0);
        default:  /* parent */
            close(pipefds[1]);

            char buffer[256];
            ssize_t nread = read(pipefds[0], sizeof(buffer) - 1);

            if (nread >= 0) {
                buffer[nread] = '\0';
                printf("My child said '%s'\n", buffer);
```

```
            }

            // collect the child
            wait(NULL);

            break;
    }
}
```

Connexion de deux processus enfants via un tube

La connexion de deux processus enfants via un canal est effectuée en connectant chacun des deux enfants au parent via différentes extrémités du même tuyau. En règle générale, le parent ne participera pas à la conversation entre les enfants, il ferme donc ses copies des deux extrémités du tuyau.

```
int demo() {
    int pipefds[2];
    pid_t child1, child2;

    if (pipe(pipefds)) {
        // error - abort ...
    }

    switch (child1 = fork()) {
        case -1:
            // error - abort
            break;
        case 0:   /* child 1 */
            close(pipefds[0]);
            write(pipefds[1], "Hello, brother!", 15);
            exit(0);
        default:  /* parent */
            // nothing
    }

    switch (child1 = fork()) {
        case -1:
            // error - abort
            break;
        case 0:   /* child 2 */
            char buffer[256];
            ssize_t nread;

            close(pipefds[1]);
            nread = read(pipefds[0], buffer, sizeof(buffer) - 1);
            if (nread < 0) {
                // handle error
            } else {
                buffer[nread] = '\0';
                printf("My brother told me '%s'\n", buffer);
            }
            exit(0);
        default:  /* parent */
            // nothing
    }

    // Only the parent reaches this point
```

```
    close(pipefds[0]);
    close(pipefds[1]);
    if (child1 >= 0) {
        wait(NULL);
        if (child2 >= 0) {
            wait(NULL);
        }
    }
}
```

Créer un pipeline de style shell

Un pipeline de type shell se compose de deux processus ou plus, chacun avec sa sortie standard connectée à l'entrée standard de la suivante. Les connexions de sortie en entrée sont construites sur des tuyaux. Pour établir un ensemble de processus semblable à un pipeline, on crée des processus enfants connectés aux tuyaux, comme décrit dans un autre exemple, et utilise en outre la fonction `dup2` (2) pour dupliquer chaque bout de tuyau sur le descripteur de fichier standard approprié de son processus. Il est généralement recommandé de fermer ensuite le descripteur de fichier d'origine du tube, en particulier si, comme c'est souvent le cas, l'enfant doit exécuter une commande différente.

```
// Most error handling omitted for brevity, including ensuring that pipe ends are
// always closed and child processes are always collected as needed; see other
// examples for more detail.
int demo() {
    int pipefds[2];
    pid_t child1 = -1, child2 = -1;

    pipe(pipefds);

    switch (child1 = fork()) {
        case -1:
            // handle error ...
            break;
        case 0:   /* child 1 */
            close(pipefds[0]);
            dup2(pipefds[1], STDOUT_FILENO);
            close(pipefds[1]);
            execl("/bin/cat", "cat", "/etc/motd", NULL);
            exit(1);  // execl() returns only on error
        default:  /* parent */
            // nothing
    }

    switch (child2 = fork()) {
        case -1:
            // handle error ...
            break;
        case 0:   /* child 2 */
            close(pipefds[1]);
            dup2(pipefds[0], STDIN_FILENO);
            close(pipefds[0]);
            execl("/bin/grep", "grep", "[Ww]ombat", NULL);
            exit(1);  // execl() returns only on error
        default:  /* parent */
            // nothing
    }
```

```
    // Only the parent reaches this point
    close(pipefds[0]);
    close(pipefds[1]);
    wait(NULL);
    wait(NULL);
}
```

L'une des erreurs les plus courantes dans la configuration d'un pipeline est d'oublier qu'un ou plusieurs des processus impliqués ferment les extrémités des tuyaux qu'il n'utilise pas. En règle générale, le résultat final devrait être que chaque extrémité de tuyau appartient à un seul processus et n'est ouverte que dans ce processus.

Dans un cas comme la fonction de démonstration ci-dessus, si le deuxième enfant ou le parent ne ferme pas `pipefds[1]` , le deuxième enfant sera suspendu, car `grep` continuera d'attendre la saisie jusqu'à ce qu'il détecte EOF, ce qui ne sera pas le cas. observé tant que la fin de l'écriture du canal reste ouverte dans n'importe quel processus, tel que le processus parent ou le deuxième enfant lui-même.

Lire Pipes en ligne: https://riptutorial.com/fr/posix/topic/8082/pipes

Chapitre 8: Processus

Syntaxe

- pid_t getpid (void);
- pid_t getppid (void);
- pid_t fork (void);
- pid_t waitpid (pid_t pid, int * wstatus, options int);
- int execv (const char * path, char * const argv []);

Paramètres

Fonction, paramètre (s), valeur de retour	La description
`fork()`	nom de la fonction
aucun	n / a
Retourne PID, 0 ou -1	Le processus appelant reçoit le PID du processus nouvellement créé ou -1 en cas d'échec. L'enfant (le processus nouvellement créé) reçoit 0. En cas d'échec, définissez `errno` sur `EAGAIN` ou `ENOMEM`
-	-
`execv()`	nom de la fonction
`const char *path`	Chaîne contenant le nom de l'exécutable (peut inclure un chemin)
`char *const argv[]`	Tableau de pointeur de chaîne comme arguments
Retourne -1 en cas d'échec	En cas de succès, cette fonction ne revient pas.
-	-

Examples

Créer un processus enfant et attendre qu'il se ferme

Ce programme montre comment exécuter un autre processus à l'aide de `fork()` et attendre sa fin en utilisant `waitpid()` :

- `fork()` crée une copie identique du processus en cours. Le processus d'origine est le

processus parent, tandis que le processus nouvellement créé est le processus enfant. Les deux processus continuent exactement après le `fork()` .

- `waitpid()` bloque jusqu'à ce que le processus fils se termine ou se termine et renvoie son code de sortie et sa raison de résiliation.

```
#include <unistd.h>     /* for fork(), getpid() */
#include <sys/types.h>  /* for waitpid() */
#include <sys/wait.h>   /* for waitpid() */
#include <stdlib.h>     /* for exit() */
#include <stdio.h>      /* for printf(), perror() */

int
main(int argc, char *argv[])
{
    /* Create child process.
     *
     * On success, fork() returns the process ID of the child (> 0) to the
     * parent and 0 to the child. On error, -1 is returned.
     */
    pid_t child_pid = fork();

    if (child_pid < 0) {
        perror("fork() failed");
        exit(EXIT_FAILURE);
    } else if (child_pid == 0) {
        /* Print message from child process.
         *
         * getpid() returns the PID (process identifier) of current process,
         * which is typically int but doesn't have to be, so we cast it.
         *
         * getppid() returns the PID of the parent process.
         */
        printf("from child: pid=%d, parent_pid=%d\n",
                 (int)getpid(), (int)getppid());

        /* We can do something here, e.g. load another program using exec().
         */
        exit(33);
    } else if (child_pid > 0) {
        /* Print message from parent process.
         */
        printf("from parent: pid=%d child_pid=%d\n",
                  (int)getpid(), (int)child_pid);

        /* Wait until child process exits or terminates.
         *
         * The return value of waitpid() is PID of the child process, while
         * its argument is filled with exit code and termination reason.
         */
        int status;
        pid_t waited_pid = waitpid(child_pid, &status, 0);

        if (waited_pid < 0) {
            perror("waitpid() failed");
            exit(EXIT_FAILURE);
        } else if (waited_pid == child_pid) {
            if (WIFEXITED(status)) {
                /* WIFEXITED(status) returns true if the child has terminated
                 * normally. In this case WEXITSTATUS(status) returns child's
```

```
                 * exit code.
                 */
                printf("from parent: child exited with code %d\n",
                          WEXITSTATUS(status));
            }
        }
    }

    exit(EXIT_SUCCESS);
}
```

Exemple de sortie:

```
from parent: pid=2486 child_pid=2487
from child: pid=2487, parent_pid=2486
from parent: child exited with code 33
```

Copié à partir d' ici , créé à l'origine par M.Geiger.

Lire Processus en ligne: https://riptutorial.com/fr/posix/topic/4534/processus

Chapitre 9: Système de fichiers

Examples

Nombre de fichiers texte dans le répertoire

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <dirent.h>
#include <ctype.h>

int main(int argc, char **argv)
{
    const char *dname = (argc > 1 ? argv[1] : ".");
    DIR* pdir = opendir(dname);
    if(pdir == NULL) {
        perror("Can't open directory");
        return EXIT_FAILURE;
    }
    int textfiles=0;
    struct dirent* dent;
    while ((dent=readdir(pdir))!=NULL) {
        int d_name_len = strlen(dent->d_name);
        if (d_name_len > 4 &&
                strcmp(dent->d_name + (d_name_len - 4), ".txt") == 0) {
            textfiles++;
        }
    }
    printf("Number of text files: %i\n", textfiles);
    closedir(pdir);
    return 0;
}
```

Supprimer les fichiers récursivement (nftw, pas thread-safe)

```
#define _XOPEN_SOURCE 500
#include <stdlib.h>  /* for exit() */
#include <stdio.h>   /* for remove() */
#include <ftw.h>     /* for nftw() */

int unlink_cb(
    const char *fpath, const struct stat *sb, int typeflag, struct FTW *ftwbuf)
{
    return remove(fpath);
}

int rm_rf(const char *path)
{
    return nftw(path,
                unlink_cb,
                64 /* number of simultaneously opened fds, up to OPEN_MAX */,
                FTW_DEPTH | FTW_PHYS);
```

```
}
```

`FTW_PHYS` drapeau `FTW_PHYS` ne suit pas les liens symboliques

`FTW_DEPTH` indicateur `FTW_DEPTH` effectue une traversée de post-ordre, c'est-à-dire appelez `unlink_cb()` pour le répertoire lui-même après avoir manipulé le contenu du répertoire et de ses sous-répertoires.

`nftw` est interrompu si la fonction de rappel renvoie une valeur différente de zéro.

Note: cette méthode n'est pas thread-safe, car `nftw` utilise `chdir` .

Supprimer les fichiers de manière récursive (openat et unlinkat, thread-safe)

```
#include <stddef.h>   /* for offsetof() */
#include <stdlib.h>   /* for exit() */
#include <stdio.h>    /* for perror() */
#include <string.h>   /* for strcmp() */
#include <unistd.h>   /* for close(), unlink() */
#include <fcntl.h>    /* for open() */
#include <dirent.h>   /* for DIR */
#include <sys/stat.h> /* for stat */
#include <limits.h>   /* for NAME_MAX */

int rm_rf(const char* path)
{
    if (unlink(path) == 0) {
        return 0;
    } else {
        int dirfd = open(path, O_RDONLY | O_DIRECTORY);
        if (dirfd == -1) {
            perror("open");
            return -1;
        }
        if (rm_children(dirfd) == -1) {
            return -1;
        }
        if (rmdir(path) == -1) {
            perror("rmdir");
            return -1;
        }
        return 0;
    }
}

int rm_children(int dirfd)
{
    DIR* dir = fdopendir(dirfd);
    if (dir == NULL) {
        perror("fdopendir");
        if (close(dirfd) == -1) {
            perror("close");
        }
        return -1;
    }

    char buf[offsetof(struct dirent, d_name) + NAME_MAX + 1];
```

```
    struct dirent* dbuf = (struct dirent*)buf;
    struct dirent* dent = NULL;

    int ret = 0;

    for (;;) {
        if ((ret = readdir_r(dir, dbuf, &dent)) == -1) {
            perror("readdir_r");
            break;
        }
        if (dent == NULL) {
            break;
        }
        if (strcmp(dent->d_name, ".") == 0 || strcmp(dent->d_name, "..") == 0) {
            continue;
        }
        if ((ret = rm_at(dirfd, dent->d_name)) == -1) {
            break;
        }
    }

    if (closedir(dir) == -1) {
        perror("closedir");
        ret = -1;
    }

    return ret;
}

int rm_at(int dirfd, const char* name)
{
    int fd = openat(dirfd, name, O_RDONLY);
    if (fd == -1) {
        perror("openat");
        return -1;
    }

    int ret = 0;

    struct stat st;
    if ((ret = fstat(fd, &st)) == -1) {
        perror("fstat");
        goto out;
    }

    if (S_ISDIR(st.st_mode)) {
        ret = rm_children(fd);
        fd = -1;
        if (ret == -1) {
            goto out;
        }
    }

    ret = unlinkat(dirfd, name, S_ISDIR(st.st_mode) ? AT_REMOVEDIR : 0);
    if (ret == -1) {
        perror("unlinkat");
        goto out;
    }

out:
```

```
    if (fd != -1) {
        if (close(fd) == -1) {
            perror("close");
            ret = -1;
        }
    }

    return ret;
}
```

Remarque: cette méthode est adaptée aux threads, mais utilise la pile pour la récursion incontrôlée. Chaque niveau d'arborescence ajoute au moins les octets NAME_MAX + une taille d'image.

Lire Système de fichiers en ligne: https://riptutorial.com/fr/posix/topic/5586/systeme-de-fichiers

Chapitre 10: Verrous de fichiers

Syntaxe

- int fcntl (int fd, int cmd, struct flock *);
- int lockf (int fd, int cmd, off_t len);

Examples

Verrous d'enregistrement POSIX (fcntl)

Cet exemple illustre l'utilisation des verrous d'enregistrement POSIX (également appelés verrous associés aux processus), fournis par la fonction `fcntl` (norme de base POSIX).

Remarques:

- Les verrous exclusifs et partagés sont pris en charge.
- Peut être appliqué à une plage d'octets, éventuellement en expansion automatique lorsque des données sont ajoutées ultérieurement (contrôlées par `struct flock`).
- Les verrous sont libérés à la première fermeture par le processus de verrouillage de *tout* descripteur de fichier pour le fichier ou lorsque le processus se termine.

```
#include <stdlib.h>  /* for exit() */
#include <stdio.h>   /* for perror() */
#include <string.h>  /* for memset() */
#include <unistd.h>  /* for close() */
#include <fcntl.h>   /* for open(), fcntl() */

int main(int argc, char **argv) {
    /* open file
     * we need O_RDWR for F_SETLK */
    int fd = open(argv[1], O_RDWR);
    if (fd == -1) {
        perror("open");
        exit(EXIT_FAILURE);
    }

    struct flock fl;
    memset(&fl, 0, sizeof(fl));

    /* lock entire file */
    fl.l_type = F_RDLCK;    /* F_RDLCK is shared lock */
    fl.l_whence = SEEK_SET; /* offset base is start of the file */
    fl.l_start = 0;         /* starting offset is zero */
    fl.l_len = 0;           /* len is zero, which is a special value
                               representing end of file (no matter
                               how large the file grows in future) */

    /* F_SETLKW specifies blocking mode */
    if (fcntl(fd, F_SETLKW, &fl) == -1) {
        perror("fcntl(F_SETLKW)");
        exit(EXIT_FAILURE);
```

```
    }

    /* atomically upgrade shared lock to exclusive lock, but only
     * for bytes in range [10; 15)
     *
     * after this call, the process will hold three lock regions:
     *   [0; 10)        - shared lock
     *   [10; 15)       - exclusive lock
     *   [15; SEEK_END) - shared lock
     */
    fl.l_type = F_WRLCK;    /* F_WRLCK is exclusive lock */
    fl.l_whence = SEEK_SET;
    fl.l_start = 10;
    fl.l_len= 5;

    /* F_SETLKW specifies non-blocking mode */
    if (fcntl(fd, F_SETLK, &fl) == -1) {
        perror("fcntl(F_SETLK)");
        exit(EXIT_FAILURE);
    }

    /* release lock for bytes in range [10; 15) */
    fl.l_type = F_UNLCK;

    if (fcntl(fd, F_SETLK, &fl) == -1) {
        perror("fcntl(F_SETLK)");
        exit(EXIT_FAILURE);
    }

    /* close file and release locks for all regions
     * note that locks are released when process calls close() on any
     * descriptor for a lock file */
    close(fd);

    return EXIT_SUCCESS;
}
```

fonction de verrouillage

Cet exemple montre l'utilisation de la fonction `lockf` (POSIX XSI).

Remarques:

- Seuls les verrous exclusifs sont pris en charge.
- Peut être appliqué à une plage d'octets, éventuellement en expansion automatique lorsque des données sont ajoutées ultérieurement (contrôlées par l'argument `len` et la position définie avec la fonction `lseek`).
- Les verrous sont libérés à la première fermeture par le processus de verrouillage de *tout* descripteur de fichier pour le fichier ou lorsque le processus se termine.
- L'interaction entre les verrous `fcntl` et `lockf` n'est pas spécifiée. Sous Linux, `lockf` est un wrapper pour les verrous d’enregistrement POSIX.

```
#include <stdlib.h>  /* for exit() */
#include <stdio.h>   /* for perror() */
#include <unistd.h>  /* for lockf(), lseek() */
#include <fcntl.h>   /* for open() */
```

```
int main(int argc, char **argv) {
    /* open file
     * we need O_RDWR for lockf */
    int fd = open(argv[1], O_RDWR);
    if (fd == -1) {
        perror("open");
        exit(EXIT_FAILURE);
    }

    /* set current position to byte 10 */
    if (lseek(fd, 10, SEEK_SET) == -1) {
        perror("lseek");
        exit(EXIT_FAILURE);
    }

    /* acquire exclusive lock for bytes in range [10; 15)
     * F_LOCK specifies blocking mode */
    if (lockf(fd, F_LOCK, 5) == -1) {
        perror("lockf(LOCK)");
        exit(EXIT_FAILURE);
    }

    /* release lock for bytes in range [10; 15) */
    if (lockf(fd, F_ULOCK, 5) == -1) {
        perror("lockf(ULOCK)");
        exit(EXIT_FAILURE);
    }

    return EXIT_SUCCESS;
}
```

Lire Verrous de fichiers en ligne: https://riptutorial.com/fr/posix/topic/5581/verrous-de-fichiers

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec POSIX	alk, Community, gavv, Jonathan Leffler, kdhp, P.P., pah
2	Des filets	alk, Nemanja Boric, P.P., Toby
3	Des minuteries	gavv, pah
4	Douilles	alk, gavv, Giorgio Gambino, kdhp, Nemanja Boric, RamenChef, Toby, yanpas
5	Les signaux	alk, gavv, Nemanja Boric, P.P., Toby
6	Multiplexage d'entrée / sortie	yanpas
7	Pipes	John Bollinger
8	Processus	alk, gavv, P.P., pah
9	Système de fichiers	gavv, yanpas
10	Verrous de fichiers	gavv