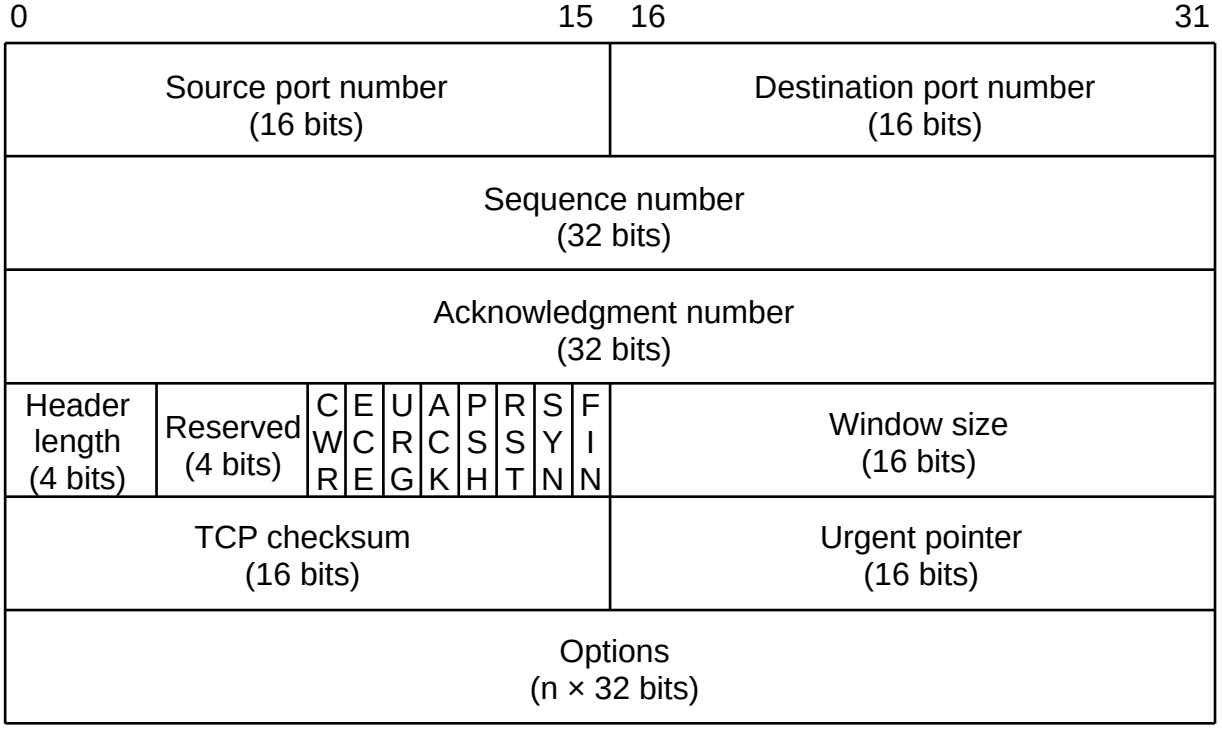
# Лекція 5
# TCP-сокети, частина 2

# Заголовок TCP

TCP-сегмент інкапсулюється в дані IP-пакета, відразу після заголовка IPv4 або IPv6. TCP-сегмент складається із заголовка TCP (20 байт базовий заголовок, до 60 байтів базовий заголовок з опціями), він однаковий для IPv4 та IPv6, та даних (payload), які відправляє програма (дані можуть не відправлятися, залежить від прапорців у заголовку).



Заголовок TCP має прапорці, які визначають використання інших полів заголовка та стан TCP-з’єднання:

- CWR (congestion window reduced)
- ECE (ENC-echo)
- URG (urgent) повідомляє іншій стороні TCP-з’єднання, що в цьому з’єднанні є важливі відправлені дані. Цей сегмент відправляється минаючи необхідність відправляти інші сегменти з даними (дані, які містяться в буфері даних на відправлення, а інша сторона повернула нульовий розмір вікна). Сегмент з прапорцем URG не обов’язково має відправлені важливі дані, але інша сторона з’єднання отримає інформацію про наявність таких даних якнайшвидше. Використання цього прапорця має відношення до поля *покажчик важливості* (*urgent pointer*).
- ACK (acknowledge) повідомляє іншій стороні TCP-з’єднання, що цей сегмент також підтверджує отримання сегментів, відправлених іншою стороною з’єднання. Сегмент з прапорцем ACK має в полі *номер підтвердження* (*acknowledgment number*) номер послідовності, до якого були отримані всі

байти від іншої сторони в цьому з’єднанні. Сегмент з підтвердженням також може мати дані, які відправляються іншій стороні цього з’єднання.

- PSH (push) повідомляє іншій стороні TCP-з’єднання, що були відправлені всі дані, які були в буфері даних на відправлення цього з’єднання. Відповідно, інша сторона цього з’єднання має повернути програмі всі дані, які бути збережені в буфері отриманих даних цього з’єднання, оскільки більше даних, відповідно до інформації від іншої сторони з’єднання, не буде в найближчому майбутньому. Цей прапорець є домовленістю між сторонами TCP-з’єднання і не має безпосереднього значення для TCP.
- RST (reset) повідомляє іншій стороні TCP-з’єднання про завершення цього з’єднання минаючи надсилання сегментів з іншими прапорцями для типового завершення з’єднання. Може відправлятися іншій стороні з’єднання навіть на перший отриманий сегмент для встановлення з’єднання (сегмент з прапорцем SYN), тобто як такого з’єднання ще може не бути. Зазвичай відправляється реалізацією, але може відправлятися програмою, яка використовує вже встановлене TCP-з’єднання.

- SYN (synchronize) повідомляє іншій стороні про встановлення TCP-з’єднання, відбувається синхронізація номерів послідовностей. Інша сторона цього з’єднання має відповісти на цей сегмент сегментом з прапорцями SYN та ACK.
- FIN (finish) повідомляє іншій стороні TCP-з’єднання про завершення цього з’єднання. З’єднання стає *напівзакритим* (*half-closed*), інша сторона може продовжувати відправляти дані по цьому з’єднанню. Інша сторона може закрити це з’єднання якщо також відправить сегмент з прапорцем FIN.

Зарезервовані біти в заголовку TCP повинні мати значення нуль, що відповідає невстановленому прапорцю. Можуть бути RFC (часто вони мають статус експериментальних), які визначають додаткові прапорці в цих зарезервованих бітах.

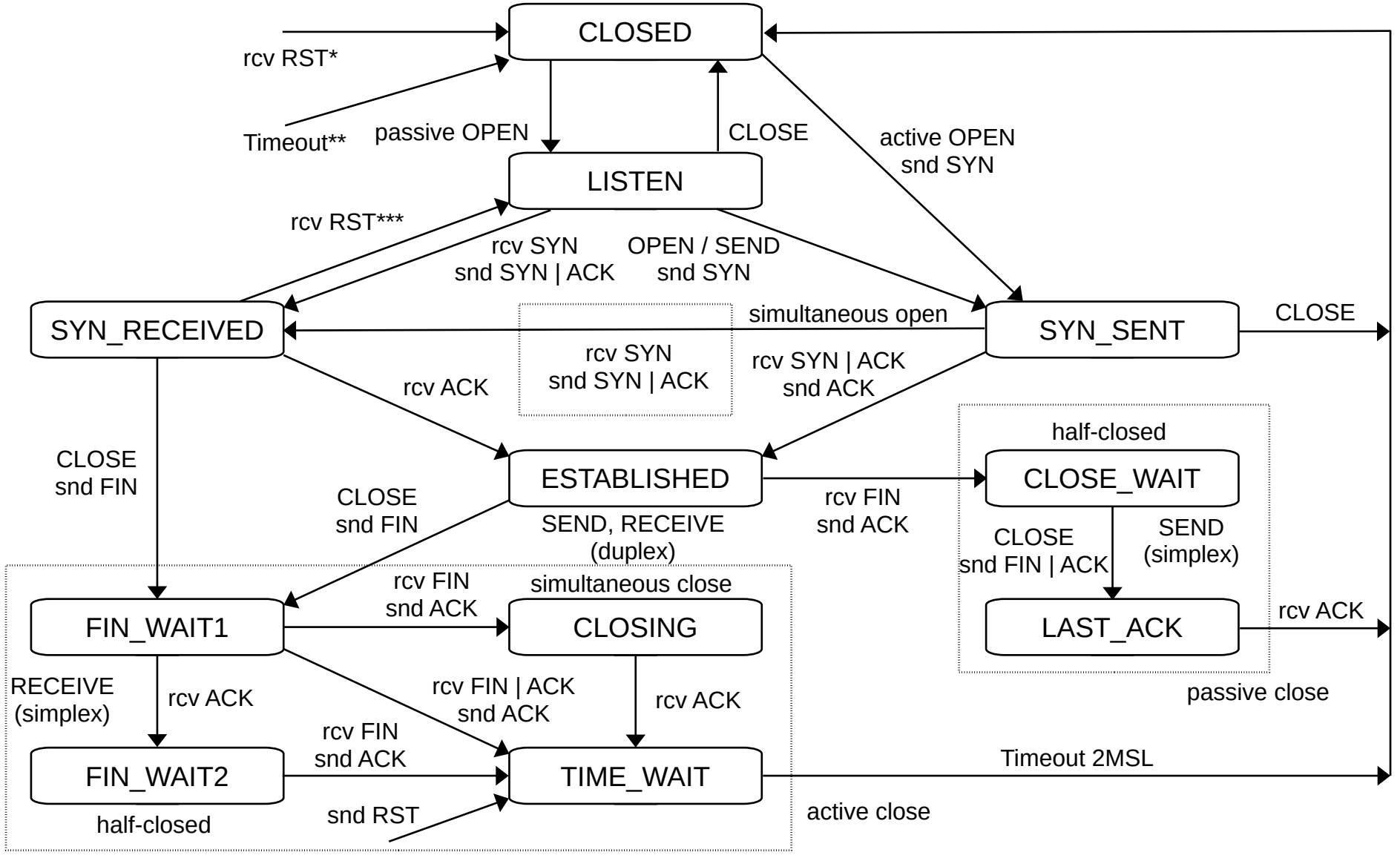
У POSIX не визначено функцію для встановлення прапорця PSH. У POSIX не визначено функцію для відправлення сегмента з встановленим прапорцем RST, але реалізації надають таку можливість, що призводить до

безпосереднього втручання в TCP. У POSIX є функція для відправлення out-of-band даних у мережеве з’єднання, що має відношення до прапорця URG.

# Діаграма станів TCP

Логіку змін стану TCP-з’єднання або TCP можна зобразити у вигляді діаграми станів. Зазвичай не все, що визначено в цій діаграмі станів, реалізується, а якщо реалізується, тоді не все може бути використано за допомогою API, визначеного в POSIX.

У прямокутниках на діаграмі станів TCP вказані назви станів. Такі самі назви станів TCP зазвичай виводить програма **netstat** для TCP-з’єднань. Слова, написані великими літерами, які не є назвами прапорців заголовка TCP, позначають запит в API. Різні ОС можуть мати різний API для використання TCP, але реалізації TCP повинні відповідати цій діаграмі станів, тому будь-якій набір функцій API для TCP надає можливість застосувати вказані запити. Слова «snd» та «rcv» в умовах переходів між станами позначають відправлення та отримання сегментів з вказаним прапорцями відповідно. Прямокутники з пунктирними лініями позначають логічну дію, яку виконує сторона TCP-з’єднання (позначають логічну дію, яка запрограмована в програмі для TCP-з’єднання).


CLOSED
rcv RST*
Timeout**
passive OPEN
CLOSE
active OPEN
snd SYN
LISTEN
rcv RST***
rcv SYN
snd SYN | ACK
OPEN / SEND
snd SYN
SYN_RECEIVED
simultaneous open
SYN_SENT
CLOSE
rcv SYN
snd SYN | ACK
rcv SYN | ACK
snd ACK
rcv ACK
half-closed
CLOSE
snd FIN
ESTABLISHED
CLOSE_WAIT
CLOSE
snd FIN
SEND, RECEIVE
(duplex)
rcv FIN
snd ACK
CLOSE
snd FIN | ACK
SEND
(simplex)
rcv FIN
snd ACK
simultaneous close
FIN_WAIT1
CLOSING
LAST_ACK
rcv ACK
RECEIVE
(simplex)
rcv ACK
rcv FIN | ACK
snd ACK
rcv ACK
passive close
rcv FIN
snd ACK
FIN_WAIT2
TIME_WAIT
Timeout 2MSL
half-closed
snd RST
active close

Стан CLOSED не є справжнім станом TCP. Цей стан позначає відсутність TCP-з'єднання. У реалізації не треба мати жодної інформації про з'єднання, яке може бути асоційоване з TCP-сокетом, якщо цього з'єднання нема. У цей стан відбувається перехід з будь-якого іншого стану в разі отримання сегмента з прапорцем RST (позначено *). Винятком є стан SYN_RECEIVED, якщо в нього відбувся перехід зі стану LISTEN (позначено ***). Сегмент з прапорцем RST може відправити реалізація в разі отримання сегменту, який має некоректний вміст заголовка TCP або якщо вміст заголовка TCP не відповідає наявним TCP-з'єднанням, або відправлення сегменту з прапорцем RST може ініціювати програма. Також у цей стан відбувається перехід з будь-якого іншого стану в разі тайм-ауту у відправленні даних, не було отримано сегмент з прапорцем ACK після кількох спроб відправлення сегмента іншої стороні з'єднання (позначено **).

Стан LISTEN позначає очікування запиту на встановлення TCP-з'єднання з будь-якої мережевої адреси та номера порту. Перехід у цій стан відбувається через пасивне відкриття (passive OPEN) TCP-з'єднання. Перехід у цій стан не потребує відправленням будь-яких сегментів (TCP-з'єднання ще нема,

відповідно нема іншої сторони з’єднання, якій можна щось відправляти), відповідно в разі закриття (CLOSE) TCP-сокета ніякий сегмент не відправляється. Діаграма станів TCP визначає можливість змінити пасивне відкриття на активне відкриття (відправити сегмента з прапорцем SYN), але зазвичай реалізація або API це не підтримують.

Для активного відкриття (active OPEN) TCP-з’єднання необхідно відправити іншій стороні з’єднання сегмент з прапорцем SYN, стан TCP-з’єднання буде SYN_SENT. Після отримання сегмента з прапорцями SYN та ACK, необхідно підтвердити отримання сегмента з прапорцем SYN від іншої сторони та змінити стан TCP-з’єднання на ESTABLISHED. Відбувається так зване *триразове рукостискання* (*three-way handshake*). Це типовий сценарій коли є пасивне відкриття TCP-з’єднання на одній стороні та активне відкриття TCP-з’єднання на іншій стороні. Діаграма станів TCP визначає можливість одночасного відкриття TCP-з’єднання (simultaneous open), але зазвичай реалізація або API це не підтримують. «Одночасно» значить, що сторони відправляють одна одній сегменти з прапорцями SYN.

Стан SYN_RECEIVED позначає очікування підтвердження на відправлений сегмент з прапорцями SYN та ACK після отримання сегмента з прапорцем SYN. Тобто інша сторона активно відкрила TCP-з'єднання, яке ця сторона погодилась прийняти (була в пасивному відкритті).

Стан ESTABLISHED позначає створене TCP-з'єднання, в якому дозволений дуплексний режим передачі даних (SEND, RECEIVE). Дані можу відправлятися разом з підтвердженнями.

Для активного закриття (active close) TCP-з'єднання необхідно відправити (CLOSE) іншій стороні з'єднання сегмент з прапорцем FIN. У станах FIN_WAIT1 та FIN_WAIT2 режим передачі даних у TCP-з'єднанні симплексний, можна тільки отримувати (RECEIVE) дані від іншої сторони з'єднання. TCP-з'єднання стає напівзакритим (half-closed). Як тільки буде отримано підтвердження від іншої сторони TCP-з'єднання про отримання сегмента з прапорцем FIN, сторона TCP-з'єднання чекає на сегмент з прапорцем FIN від іншої сторони та підтверджує його. Стан TCP-з'єднання буде TIME_WAIT для сторони з'єднання, яка активно закрила з'єднання. Може бути так, що обидві сторони TCP-з'єднання одночасно активно закрили з'єднання (simultaneous close), тобто

відправили сегмент з прапорцем FIN одна одній. У цьому випадку стан TCP-з'єднання буде TIME_WAIT для обох сторін з'єднання, оскільки вони обидві активно закрили з'єднання.

Стан CLOSE_WAIT позначає напівзакрите TCP-з'єднання для сторони з'єднання, яке отримало сегмент з прапорцем FIN. У цьому стані режим передачі даних у TCP-з'єднанні симплексний, можна тільки відправляти (SEND) дані іншій стороні з'єднання. Для пасивного закриття (pasive close) TCP-з'єднання необхідно відправити (CLOSE) іншій стороні з'єднання сегмент з прапорцем FIN. Як тільки буде отримано підтвердження від іншої сторони TCP-з'єднання отримання відправленого сегмента з прапорцем FIN, інформація про з'єднання забирається з системи.

Стан TIME_WAIT позначає закрите TCP-з'єднання для сторони з'єднання, яке активно закрила з'єднання. Закрите TCP-з'єднання буде в цьому стані два максимальних проміжки часу існування TCP-сегменту в мережі (*maximum segment lifetime*, MSL) після кожного відправленого сегменту з підтвердженням. RFC 9293 визначає MSL у 2 мінути, відповідно 2MSL буде 4 мінути, але реалізації зазвичай використовують менший проміжок часу. Значення 2MSL

значить один MSL для отримання сегмента з прапорцем FIN від іншої сторони TCP-з'єднання і один MSL для отримання іншою стороною відправленого підтвердження. Наявність цього стану запобігає створенню нового TCP-з'єднання з тими самими мережевими адресами та номерами портів, це називається *інкарнація* (*incarnation*) з'єднання. Впродовж проміжку часу поки закрите TCP-з'єднання знаходиться в стані TIME_WAIT відбувається підтвердження отриманих сегментів з прапорцем FIN від іншої сторони з'єднання та ігнорування дубльованих сегментів від іншої сторони з'єднання. Якщо не використовувати цей стан, тоді інша сторона TCP-з'єднання може не отримати підтвердження на відправлений сегмент з прапорцем FIN, а інкарнація цього з'єднання отримає сегменти, які були відправлені іншою стороною з'єднання для попереднього з'єднання. Після завершення цього проміжку часу інформація про TCP-з'єднання забирається з системи.

Сторона TCP-з'єднання, яка пасивно закриває з'єднання, відправляє сегмент з прапорцями FIN та ACK (підтвердження повторюється), тому, якщо вона отримає підтвердження для свого відправленого сегмента з прапорцем FIN,

тоді інша сторона з’єднання також отримала підтвердження для свого відправленого сегмента з прапорцем FIN.

Стан TIME_WAIT не потрібен для сторони TCP-з’єднання, яка пасивно закриває з’єднання, оскільки неможливість отримання дубльованих сегментів від іншої сторони гарантується неможливістю інкарнації TCP-з’єднання з іншою стороною, яке має попереднє з’єднання в стані TIME_WAIT.

У разі настання тайм-ауту для неотримання підтвердження на відправлений сегмент (сегмент з прапорцем SYN, FIN, або сегмент з даними) стан TCP-з’єднання змінюється на CLOSED (позначено **).

Застосування системного виклику `listen()` для TCP-сокета призводить до пасивного відкриття з’єднання, стан LISTEN. Застосування системного виклику `connect()` для TCP-сокета призводить до активного відкриття з’єднання, стан SYN_SENT. Відправлення даних у TCP-з’єднанні виконується системним викликом `write()` або відповідними. Отримання даних у TCP-з’єднанні виконується системним викликом `read()` або відповідними.

Завершення TCP-з’єднання значить, що сторона з’єднання закрила відповідний сокет та відправила іншій стороні з’єднання сегмент з прапорцем FIN. Змінення режиму передачі даних у TCP-з’єднанні на симплексний у напрямку отримання даних значить, що іншій стороні з’єднання відправлено сегмент з прапорцем FIN. Тобто в TCP не визначено чи з’єднання закрито іншою стороною чи інша сторона змінила режим передачі даних на симплексний у напрямку отримання даних (в обох випадках відправляється сегмент з прапорцем FIN). Якщо TCP-з’єднання завершено, тоді стан з’єднання змінюється на відповідний і вмикається тайм-аут. Після завершення тайм-ауту, інформація про це TCP-з’єднання забирається з системи, навіть якщо не було отримано жодного сегменту від іншої сторони з’єднання. Якщо цього не зробити, тоді сторона TCP-з’єднання, яка завершує з’єднання, буде знаходитися в стані FIN_WAIT1 обмежений час (поки не буде отримано підтвердження на відправлений сегмент з прапорцем FIN або поки не завершаться тайм-аути не отриманих підтверджень на повторно відправлених сегментів з прапорцями FIN) або в стані FIN_WAIT2 необмежений час, якщо інша сторона з’єднання не відправить сегмент з прапорцем FIN або RST. У результаті це призведе, що в системі

можуть бути використані всі доступні номерів портів і не буде можливо призначити новому TCP-сокету жодного номера порту.

# Опції сокета

Кожний сокет має опції. Деякі з цих опцій визначають характеристики сокета, які встановлює програма користувача, тому не мають типових значень. Деякі з цих опцій визначають параметри сокета, тому мають типові значення, визначені POSIX, або типові значення, визначені реалізацією. Програма користувача може отримати поточні значення всіх опцій сокета. Програма користувача може змінити значення деяких опцій сокета, якщо типові значення цих опцій неефективні для конкретного сценарію використання сокета.

Системний виклик `getsockopt()` повертає значення вказаної опції для сокета, асоційованого з вказаним дескриптором файлу. Системний виклик `setsockopt()` встановлює значення вказаної опції для сокета, асоційованого з вказаним дескриптором файлу.

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

int getsockopt(int sockfd, int level, int optname, void *restrict optval,
        socklen_t *restrict optlen);
int setsockopt(int sockfd, int level, int optname, const void *optval,
        socklen_t optlen);
```

Аргумент `sockfd` – це дескриптор файлу, з яким має бути асоційований сокет, для якого застосовується системний виклик. Аргумент `level` визначає рівень на якому опція має застосовуватися. Аргумент `optname` визначає опцію.

Системний виклик `getsockopt()` копіює поточне значення вказаної опції в об’єкт, на який вказує аргумент `optval`. Об’єкт, на який вказує аргумент `optlen`, має містити розмір об’єкта, на який вказує аргумент `optval`. Якщо розмір значення опції більший, ніж вказаний розмір, тоді копіюється частина значення опції. Кількість скопійованих байт значення опції записується в об’єкт, на який вказує аргумент `optlen`. Тобто недостатній розмір об’єкта для отримання значення опції не призводить до помилки.

Системний виклик `setsockopt()` змінює значення вказаної опції. Сенс аргументу `optval` такий самий, як у `getsockopt()`. Аргумент `optlen` має дорівнювати розміру об’єкта, на який вказує аргумент `optval`.

Далі, якщо не вказано інше, тоді об’єкт зі значенням опції має мати тип `int`. Якщо опція булева, тоді ненульове значення опції позначає увімкнену опцію, нульове значення опції позначає вимкнену опцію.

POSIX визначає такі опції для сокета на рівні SOL_SOCKET. Цей рівень та опції визначені в <sys/socket.h>. Реалізація може мати додаткові опції на цьому рівні також.

- Опція SO_ACCEPTCONN встановлена для сокета, якщо цей сокет може приймати з'єднання. Ця опція має використовуватися в getsockopt(). Ця опція не має типового значення.
- Опція SO_BROADCAST вмикає дозвіл на відправлення широкомовних датаграм для сокета. Ця опція може бути застосована тільки для датаграмних сокетів (тип сокета має бути SOCK_DGRAM). Ця опція вимкнена, як усталено.
- Опція SO_DEBUG вмикає режим налагодження в реалізації для сокета. Семантика режиму налагодження визначає реалізація. Ця опція вимкнена, як усталено.
- Опція SO_DONTROUTE вимикає стандартну маршрутизацію для відправлених повідомлень для сокета. Відповідно адреса отримувача відправленого повідомлення має бути в мережі, яка напряму досяжна з системи. Чи буде підтримуватися ця опція і як буде вибиратися мережевий інтерфейс для

спрямування повідомлення залежить від мережевого протоколу. Підтримування цієї опції залежить від реалізації. Ця опція вимкнена, як усталено.

- Опція `SO_ERROR` повертає та скидає інформацію про поточну помилку в сокеті. Ця опція має використовуватися в `getsockopt()`. Якщо поточне значення опції нуль, тоді сокет не має помилки. Отримання значення цієї опції може бути застосовано для перевірки асинхронних помилок в сокеті. Ця опція не має типового значення.
- Опція `SO_KEEPALIVE` вмикає періодичне відправлення повідомлень (що це за повідомлення визначає мережевий протокол) для перевірки мережевого з'єднання, асоційованого з сокетом. Якщо відправлення повідомлення завершиться помилкою, тоді для сокета буде встановлена відповідна помилка. Ця опція вимкнена, як усталено.
- Значення опції `SO_LINGER` (перекладається, як «затриматись») визначає семантику виконання системного виклику `close()` застосованого для сокета, якщо сокет не асоційований з жодним іншим дескриптором файлу.

Семантика впливає на дані, які містяться в буфері даних на відправлення, та на умову завершення виконання `close()`. Ця семантика залежить від мережевого протоколу. Значення цієї опції вказується в об’єкті типу `struct linger`. У `struct linger` є принаймні такі поля: `int l_onoff` (якщо нуль, тоді ця опція вимкнена, якщо не нуль, тоді ця опція увімкнена), `int l_linger` (проміжок часу в секундах). Типове значення цієї опції дорівнює нулям у полях `l_onoff` та `l_linger`, тобто ця опція вимкнена.

- Опція `SO_OOBINLINE` вказує додавати отримані out-of-band дані до звичайних даних. Ця опція має сенс тільки для протоколів, які підтримуються out-of-band дані. Ця опція вимкнена, як усталено.
- Значення опцій `SO_RCVBUF` та `SO_SNDBUF` визначають розмір буфера отриманих даних та розмір буфера даних на відправлення сокета відповідно. Ці опції мають типові значення, які залежать від реалізації.
- Значення опції `SO_RCVLOWAT` («receive low water mark») визначає мінімальну кількість байтів, яка опрацьовується в разі читання даних із сокета. Системний виклик `read()`, застосований для сокета, блокується, поки не буде

отримано меншу кількість байтів із кількість байтів, вказаної в його аргументі, або значення цієї опції. Значення цієї опції визначає ознаку готовності сокета для читання в мультиплексуванні введення-виведення. Менша кількість байтів може повернутися в разі помилки, отримання сигналу, отримання іншого типу даних у мережевому з’єднанні. Типове значення цієї опції `1` байт. Можливість встановлення значення цієї опції залежить від реалізації.

- Значення опцій `SO_RCVTIMEO` та `SO_SNDTIMEO` визначають тайм-аути для отримання та відправлення даних для сокета відповідно. Значення цієї опції вказується в об’єкті типу `struct timeval`. У `struct timeval` є принаймні такі поля: `time_t tv_sec` (секунди), `suseconds_t tv_usec` (знаковий цілочисельний тип достатній для збереження значень у діапазоні [`-1`, `1000000`], мікросекунди). Якщо системний виклик `read()` або `write()` застосований для сокета блокується довше, ніж вказаний тайм-аут, тоді цей системний виклик виконується частково (прочитає або запише менше байтів, ніж вказано в аргументі) або завершується з помилкою `EAGAIN` або `EWOULDBLOCK` (нічого не було прочитано або записано). Настання тайм-ауту через встановлену опцію `SO_RCVTIMEO` або

SO_SNDTIMEO під час виконання read() або write() ніяк не впливає на з’єднання. Типові значення цих опцій нуль секунд та нуль мікросекунд, що позначає відсутність тайм-ауту. Можливість встановлення значень цих опцій залежить від реалізації.

- Опція SO_REUSEADDR вказує дозволити повторне призначення адреси сокету. Семантика застосування цієї опції залежить від мережевого протоколу. Ця опція вимкнена, як усталено.
- Значення опції SO_SNDLOWAT («send low water mark») визначає мінімальну кількість байтів, яка опрацьовується в разі записи даних у сокет. Системний виклик write(), застосований для сокета, блокується, поки не буде можливості відправити меншу кількість байтів із кількість байтів, вказаної в його аргументі, або значення цієї опції. Значення цієї опції визначає ознаку готовності сокета для запису в мультиплексуванні введення-виведення. Типове значення цієї опції залежить від реалізації. Можливість встановлення значення цієї опції залежить від реалізації.

- Значення опції SO_TYPE дорівнює типу сокета (тип сокета, який вказується в socket()). Ця опція має використовуватися в getsockopt(). Програма користувача використовуючи fstat() та getsockopt(), може отримати інформацію про сокет, асоційований з дескриптором файлу. Ця опція не має типового значення.

POSIX нічого не визначає щодо значень опцій сокета, асоційованого з дескриптором файлу, який повертає accept(), але зазвичай цей сокет успадковує значення опцій сокета, який був вказаний у його аргументі (сокет, для якого був застосований listen()).

Опція SO_KEEPALIVE застосована для TCP-сокета вмикає періодичне відправлення повідомлень для перевірки мережевого з'єднання, асоційованого з сокетом. Проміжок часу відправлення періодичних відправлень повідомлень визначає реалізація, але перший keep-alive сегмент рекомендовано відправити не раніше, ніж через дві години неактивності TCP-з'єднання.

Опція SO_RCVBUF має застосовуватися для TCP-сокета до застосування для цього сокета системного виклику connect() або listen(). Це пов'язано з тим, що опція

масштабування вікна в TCP вказується в сегменті з прапорцем SYN, тобто під час встановлення з’єднання.

Результат застосування `setsockopt()` з опціями `SO_ACCEPTCONN`, `SO_ERROR` та `SO_TYPE` для сокета на рівні `SOL_SOCKET` не специфіковано.

Семантика застосування опцій `SO_LINGER`, `SO_OOBINLINE` та `SO_REUSEADDR` для TCP-сокета пояснюється далі. Семантика застосування опцій `SO_RCVLOWAT` та `SO_SNDLOWAT` для TCP-сокета пояснюється в наступній лекції.

Семантика застосування деяких опцій для UDP-сокетів та SCTP-сокетів пояснюється у відповідних лекціях про ці транспортні протоколи.

POSIX визначає такі опції для сокета на рівні `IPPROTO_TCP`. Цей рівень та опції визначені в `<netinet/in.h>`. Реалізація може мати додаткові опції на цьому рівні також.

- Опція `TCP_NODELAY` запобігає об’єднанню даних, які містяться в буфері даних TCP-сокета на відправлення, для відправлення їх в одному TCP-сегменті.

Реалізація не має підтримувати застосування `getsockopt()` та `setsockopt()` з цією опцією. Ця опція вимкнена, як усталено.

Сторона TCP-з'єднання може не відправляти нові дані, якщо не були підтверджені раніше відправлені дані. У такий спосіб зменшується кількість відправлених сегментів. Інша сторона TCP-з'єднання може затримувати відправлення сегмента з прапорцем ACK (підтвердження), для відправлення його з іншими даними, які ще можуть бути відправлені невдовзі в це з'єднання. Якщо інша сторона TCP-з'єднання не відправляє дані, тоді відправлення підтвердження затримується, відповідно відправлення нового сегменту іншій стороні TCP-з'єднання також затримується.

Опцію `TCP_NODELAY` є сенс застосовувати для TCP-сокета, який використовується для відправлення даних, які мають відношення до чогось інтерактивного. Якщо TCP-сокет використовується для відправлення даних, які мають формат деякого протоколу рівня застосунку (заголовки та дані), тоді програмі користувача є сенс відправляти дані в одному виклику `write()` (програма користувача має об'єднати вміст заголовка та даних в одному буфері) або `writev()` (вміст заголовка та даних можуть бути в різних буферах, ядро об'єднає

їхній вміст), щоб сторона TCP-з’єднання відразу розуміла скільки даних треба відправити.

POSIX визначає декілька опцій для сокета на рівні `IPPROTO_IPV6`. Цей рівень та опції визначені в `<netinet/in.h>`. Використання цих опцій поза тем лекцій.

# Призначення адреси сокету, частина 2

Призначення адреси TCP-сокету, коли програма користувача запитує ядро призначити сокету вільний номер порту (програма вказує нульовий номер порту в структурі адреси сокета в `bind()`), працює завжди (якщо буде вільний номер порту з відповідного діапазону номерів портів і якщо в ядра буде достатньо ресурсів для цього призначення). Якщо програма користувача вказує конкретний номер порту в структурі адреси сокета для призначення TCP-сокету (програма вказує не нульовий номер порту в структурі адреси сокета в `bind()`), тоді є особливості, якщо є інший TCP-сокет в системі з таким самим номером порту. Ці особливості обумовлені питаннями коректності реалізації TCP та питаннями безпеки.

Вплинути на систему під час призначення адреси TCP-сокету можна опцією `SO_REUSEADDR` у `setsockopt()`. POSIX не визначає семантику застосування цієї опції для TCP-сокетів. Далі наводиться семантика застосування опції `SO_REUSEADDR` для TCP-сокетів у реалізаціях.

1. Опція SO_REUSEADDR, застосована для TCP-сокета, дозволяє призначити сокету IP адресу та номер порту, якщо вже є TCP-з'єднання, в якому використовуються ці IP адреса та номер порту.

   Кожне TCP-з'єднання ідентифікується IP адресою та номером порту одної сторони з'єднання та IP адресою та номером порту іншої сторони з'єднання. Не можна створити TCP-з'єднання між сторонами з тими самими IP адресами та тими самими номерами портів, оскільки реалізація не зможе визначити до якого з'єднання належить отриманий сегмент. Наявне TCP-з'єднання може мати будь-який стан, може мати стан TIME_WAIT також, тобто з'єднання може бути завершено (реалізація чекає на повторно відправлений сегмент з прапорцем FIN та на дубльовані сегменти, які вже були підтверджені раніше).

   Системи зазвичай мають більш строгі вимоги вимоги та не дозволяють призначити TCP-сокету IP адресу та номер порту, якщо вже є TCP-з'єднання, в якому використовуються ці IP адреса та номер порту, незалежно від того як буде використовуватися цей сокет (для активного відкриття або для пасивного відкриття TCP-з'єднання).

Типові сценарії такі. Сервер приймає TCP-з’єднання від клієнта, це з’єднання опрацьовується в дочірньому процесі (наприклад, його стан буде ESTABLISHED), оригінальний процес сервера завершує своє виконання, сервер запускається знову. Сервер приймає TCP-з’єднання, працює з цим з’єднанням, активно завершує це з’єднання (наприклад, його стан буде TIME_WAIT), завершує своє виконання, сервер запускається знову. У цих двох сценаріях є TCP-з’єднання, в якому використовується певні IP адреса та номер порту. Якщо опція `SO_REUSEADDR` не застосована, тоді в цих сценаріях сервер не зможе призначити своєму TCP-сокету IP адресу та номер порту, які ще використовується в наявному TCP-з’єднанні (стан цього з’єднання може бути будь-яким). Системний виклик `bind()` буде завершено з помилкою `EADDRINUSE`.

У цих сценаріях замість сервера, який не можна запустити повторно, може бути новий клієнт, який не може призначити своєму TCP-сокету конкретну IP адресу та номер порту, якщо вже є завершене TCP-з’єднання з тими самими IP адресою та номером порту. Якщо клієнт спробує активно відкрити TCP-з’єднання і якщо це з’єднання буде мати такі самі IP адреси та номери

портів як наявне TCP-з’єднання, тоді системний виклик `connect()` буде завершено з помилкою `EADDRINUSE`. Зазвичай клієнт запитує ядро призначити вільний номер порту для свого TCP-сокета, тому на стороні клієнта `bind()` зазвичай не завершується з помилкою `EADDRINUSE`. Навіть якщо для цього сокета не застосована опція `SO_REUSEADDR`.

Деякі реалізації можуть прийняти запит на створення нового TCP-з’єднання, якщо в системі ще є інформація про попереднє TCP-з’єднання з тими самими IP адресами та номерами портів, стан якого TIME_WAIT, якщо значення послідовності байтів та значення опції мітки часу в заголовку TCP дають змогу однозначно визначити сегменти, відправлені для нового з’єднання.

Опцію `SO_ADDRINUSE` є сенс застосовувати в сервері для TCP-сокета, який буде приймати нові з’єднання. Цю опцію можна застосовувати в клієнті для TCP-сокета, який буде використовуватися для активного відкриття нового з’єднання. Застосування цієї опції для TCP-сокета не може якось вплинути на TCP.

2. Опція SO_REUSEADDR, застосована для TCP-сокета, дозволяє призначити TCP-сокету адресу одного з мережевих інтерфейсів системи, якщо є інший TCP-сокет з wildcard адресою і тим самим номером порту. Опція SO_REUSEADDR, застосована для TCP-сокета, дозволяє призначити TCP-сокету wildcard адресу, якщо є інші TCP-сокети з адресами мережевих інтерфейсів системи.

   Нові TCP-з'єднання призначаються TCP-сокету з найбільш специфічною адресою. Якщо є TCP-сокет з адресою, з якою було встановлено TCP-з'єднання, тоді система вибирає цей сокет, інакше система вибирає TCP-сокет з wildcard адресою (якщо TCP-з'єднання встановлено, тоді точно є якийсь TCP-сокет, адреса якого відповідає адресі, яка була застосована в сегменті з прапорцем SYN, або він має wildcard адресу). Проблема полягає в тому, що якщо програма, яка приймає TCP-з'єднання, не призначила своїм TCP-сокетам

   У деяких системах замість опції SO_REUSEADDR треба використовувати нестандартну опцію SO_REUSEPORT. У цьому є сенс, оскільки цей сценарій відрізняється від першого сценарію.

Типовий сценарій такий. Є «офіційний» сервер, який призначив своєму TCP-сокету wildcard адресу або адресу одного з мережевих інтерфейсів системи. Якщо звичайний користувач запустить свій сервер і призначить його TCP-сокету адресу одного з мережевих інтерфейсів системи або wildcard адресу і такий самий номер порту, тоді цей сервер звичайного користувача буде приймати запити від деяких клієнтів (система вибирає TCP-сокет з найбільш специфічною адресою), замість «офіційного» сервера. Якщо номер порту «офіційного» сервера належить діапазону широко відомих портів (1–1023), тоді процес користувача має мати відповідні привілеї для призначення цього номера порту своєму сокету, тобто процес звичайного користувача не зможе призначити такий номер порту своєму TCP-сокету.

Система може забороняти призначати TCP-сокету, який приймає нові з'єднання, номер порту, якщо вже є TCP-сокет, який приймає нові з'єднання, з таким самим номером порту навіть якщо була застосована опція `SO_REUSEADDR` або нестандартна опція `SO_REUSEPORT`, для процесів одного користувача або процесів різних користувачів. Це робиться з міркувань безпеки.

Реалізація може вимагати, щоб опція SO_REUSEADDR або нестандартна опція SO_REUSEPORT також була застосована для сокетів, які ще використовуються (якщо TCP-з’єднання відкриті) або які раніше використовувалися (якщо TCP-з’єднання закриті), інакше ці опції не будуть мати ефект у разі їхнього застосування для нового TCP-сокета. У цьому є сенс, усі програми, які призначають певний номер порту для своїх TCP-сокетів, мають погодитися на семантику опції SO_REUSEADDR або нестандартної опції SO_REUSEPORT. Оскільки опція SO_REUSEPORT нестандартна і якщо вона визначена в системі, тоді є сенс її застосовувати разом з опцією SO_REUSEADDR (перевірити чи вона визначена в системі можна за допомогою `#ifdef` у вихідному коді програми).

# Відправлення та отримання даних, частина 2

Системні виклики `read*()` та `write*()` можна застосовувати для будь-яких об’єктів ядра (файли, неіменовані канали, сокети). Але іноді треба вказати специфічні аргументи для введення-виведення для сокетів. POSIX визначає кілька функцій введення-виведення, які можна застосовувати тільки для сокетів.

Системний виклик `recv()` зчитує дані з сокета, асоційованого з вказаним дескриптором файлу.

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

ssize_t recv(int sockfd, void *buf, size_t count, int flags);
```

Аргументи `sockfd`, `buf` та `count` мають такий самий сенс як у `read()`, але з дескриптором файлу `sockfd` має бути асоційований сокет. Цей системний виклик повертає таке саме значення як `read()`. Прапорці в аргументі `flags` можуть бути такі: прапорець `MSG_KEEP` вказує залишити прочитані дані, їх можна прочитати знову; прапорець `MSG_OOB` вказує повернути out-of-band дані; прапорець `MSG_WAITALL` вказує блокуватися під час читання даних з потокового сокета (тип сокета має бути `SOCK_STREAM`) до тих пір, доки не будуть прочитані всі запитані дані. Реалізації

зазвичай мають додаткові нестандартні прапорці для цього системного виклику. Застосування прапорця `MSG_WAITALL` може призвести до того, що `recv()` може прочитати менше даних, ніж було запитано (у разі наявності повідомлень у з'єднанні, переривання виконання `recv()` сигналом, завершення з'єднання, використання також прапорця `MSG_KEEP` або помилки в з'єднанні).

Системний виклик `send()` записує дані в сокет, асоційований з вказаним дескриптором файлу.

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

ssize_t send(int sockfd, const void *buf, size_t count, int flags);
```

Аргументи `sockfd`, `buf` та `count` мають такий самий сенс як у `write()`, але з дескриптором файлу `sockfd` має бути асоційований сокет. Цей системний виклик повертає таке саме значення як `write()`. Прапорці в аргументі `flags` можуть бути такі: прапорець `MSG_EOR` вказує завершити запис; прапорець `MSG_OOB` вказує відправити out-of-band дані; прапорець `MSG_NOSIGNAL` вказує не відправляти синхронний сигнал `SIGPIPE` потоку в разі відправлення даних у сокет, асоційоване мережеве з'єднання з яким вже завершене (у цьому випадку виконання `send()`

завершиться з помилкою `EPIPE`). Реалізації зазвичай мають додаткові нестандартні прапорці для цього системного виклику.

Семантика застосування прапорця `MSG_OOB` у системних викликах `recv()` та `send()` залежить від протоколу сокета (може взагалі не мати сенсу). Семантика застосування прапорця `MSG_EOR` у системному виклику `send()` залежить від протоколу сокета (може взагалі не мати сенсу).

Системні виклики `recv()` та `send()` можна застосовувати для TCP-сокетів.

Прапорець `MSG_OOB` можна застосовувати в системних викликах `recv()` та `send()` для TCP-сокетів, хоча в TCP нема справжніх out-of-band даних. Прапорець `MSG_EOR` нема сенсу застосовувати в системному виклику `send()` для TCP-сокетів, оскільки в TCP нема повідомлень (записів).

Усе, що було зазначено для системних викликів `read()` та `write()`, застосованих для TCP-сокетів у попередній лекції, також є актуальним для системних викликів `recv()` та `send()`, застосованих для TCP-сокетів відповідно.

Ще є системні виклики `recvfrom()` та `sendto()`, які можна застосовувати для TCP-сокетів. Семантика їхнього застосовування для TCP-сокетів нічім не відрізняється від семантики системних викликів `recv()` та `send()` відповідно, тому тут не йдеться про ці системні виклики.

Ще є системні виклики `recvmsg()` та `sendmsg()`, які можна застосовувати для TCP-сокетів. Ці системні виклики дають можливість вказувати кілька буферів, як у системних викликів `readv()` та `writev()` відповідно. Крім цієї можливості, семантика їхнього застосування для TCP-сокетів нічім не відрізняється від семантики застосування системних викликів `recv()` та `send()`, тому тут не йдеться про ці системні виклики.

Іноді треба прочитати дані з одного дескриптора файлу та без змін записати ці дані в інший дескриптор файлу. Читання даних потребує копіювання даних із простору ядра в простір користувачу, а запис даних потребує копіювання даних із простору користувача в простір ядра. Тобто дані копіюються між простором ядра та простором користувача два рази. Для оптимізації цього сценарію деякі системи мають системний виклик `sendfile()`, який дає змогу прочитати дані з одного дескриптора файлу та записати їх в інший дескриптор файлу всередині

ядра, тобто дані не копіюються між простором ядра та простором користувача взагалі. Цей системний виклик зазвичай має різні аргументи в різних системах (два дескриптори файлу та розмір даних в аргументах присутні завжди, можуть бути ще зміщення та прапорці), тому тут не надається його прототип відповідної функції. Також різні системи накладають різні вимоги щодо об’єктів, з якими можуть бути асоційовані ці два дескриптори файлів.

# Out-of-band дані

Out-of-band дані, які треба відправити якнайшвидше, ігноруючи управління потоком та дані, які містяться в буфері даних на відправлення. Програма використовує out-of-band дані, якщо треба відправити відправити щось важливе іншій стороні мережевого з’єднання.

У TCP нема справжніх out-of-band даних, у TCP є важливі дані. Якщо в TCP-сегменті встановлено прапорець URG, тоді в заголовку цього сегмента встановлено значення покажчика важливості, який визначає зміщення наступного байту за важливими даними відносно початку даних у цьому сегменті. Тобто важливі дані можуть бути відправлені в одному з наступних сегментів. Щойно програма відправила важливі дані іншій стороні TCP-з’єднання, реалізація відправляє іншій стороні з’єднання сегмент з прапорцем URG та з встановленим значенням покажчика важливості навіть якщо інша сторона цього з’єднання повідомляє нульове вікно, тобто повідомляє про неможливість прийняти нові дані. Максимальне значення покажчика важливості обмежено (воно має 16 біт, але це не принципово), тому в цьому значенні може бути неможливо точно закодувати зміщення наступного байту за

важливими даними відносно початку даних у поточному сегменті. Якщо в значенні покажчика важливості неможливо точно закодувати потрібне значення, тоді одна сторона TCP-з’єднання в покажчику важливості вказує значення, яке представляє важливі дані не в поточному сегменті, а інша сторона з’єднання не контролює чи це значення має сенс для наступних отриманих сегментів.

Ця ідея важливих даних у TCP значить, що в TCP є можливість відправити один байт важливих даних. У TCP є можливість відправити важливі дані кілька разів, але якщо інша сторона TCP-з’єднання не отримала інформацію про вже отриманий байт важливих даних, тоді інформація про отриманий наступний байт важливих даних замініть інформацію про раніше отриманий байт важливих даних у цьому з’єднанні.

Опція `SO_OOBINLINE`

Системний виклик `sockatmark()`.

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

int sockatmark(int sockfd);
```

# Завершення з’єднання, частина 2

Мережеве з’єднання завершується, якщо сокет, з яким воно асоційоване, закривається або реалізація отримає помилку в з’єднанні. Є сценарії, коли програмі необхідно завершити тільки відправлення даних або тільки отримання даних у мережевому з’єднання, тобто змінити режим передачі даних у мережевому з’єднанні на симплексний в одному з напрямків передачі даних. Якщо комунікаційний домен мережевого з’єднання підтримує симплексний режим передачі даних для відповідного напрямку передачі даних, тоді одна сторона з’єднання може змінити режим передачі даних на потрібний симплексний режим і в такий спосіб повідомити іншу сторону про деяку подію. Тобто можна інформувати іншу сторону мережевого з’єднання не використовуючи протокол рівня застосунку, а засобами самого траспортного протоколу мережевого з’єднання.

Системний виклик `shutdown()` завершує відправлення та/або отримання даних для мережевого з’єднання, яке асоційоване з сокетом, з яким асоційований вказаний дескриптор файлу.

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

int shutdown(int sockfd, int how);
```

Аргумент `sockfd` – це дескриптор файлу, який має бути асоційований з сокетом, з яким має бути асоційоване з'єднання. Значення аргументу `how` мають бути такі: `SHUT_RD` (завершити отримання даних у з'єднанні), `SHUT_WR` (завершити відправлення даних у з'єднанні), `SHUT_RDWR` (завершити відправлення та отримання даних у з'єднанні).

POSIX не визначає чи будуть доступні раніше отримані дані з буфера отриманих даних сокета та чи будуть відправлені дані з буфера даних на відправлення сокета, після застосування для нього функції `shutdown()` з аргументом `SHUT_RD` або `SHUT_WR` відповідно. POSIX не визначає чи застосування `shutdown()` з аргументом `SHUT_RDWR` еквівалентно двом застосуванням `shutdown()` з аргументами `SHUT_RD` та `SHUT_WR`. POSIX також не визначає семантику функції `shutdown()` для мережевих з'єднань у конкретних транспортних протоколах конкретних комунікаційних доменів. Інформація, яка надається далі, відповідає семантиці системного виклику `shutdown()`, застосованого для TCP-сокета, яка реалізована в наявних системах.

Використання `SHUT_WR` у виклику `shutdown()`, застосованого для TCP-сокета, призводить до відправлення всіх даних з буфера даних на відправлення та до відправлення сегменту з прапорцем FIN. TCP-з'єднання стає напівзакритим або режим передачі даних у TCP-з'єднанні стає симплексним у напрямку отримання даних для сторони, яка застосувала `SHUT_WR` у виклику `shutdown()` для відповідного сокета. Сторона TCP-з'єднання, яка завершила відправлення даних (стан CLOSE_WAIT), може продовжувати отримувати дані від іншої сторони з'єднання (стани FIN_WAIT1 та FIN_WAIT2).

Використання `SHUT_RD` у виклику `shutdown()`, застосованого для TCP-сокета, не призводить до відправлення сегменту з яким-небудь прапорцем, оскільки такий прапорець не визначений у TCP. Тому завершити отримання даних у TCP-з'єднанні на рівні TCP неможливо, але можливо логічно на рівні реалізації. Реалізації можуть повертати процесу раніше отримані дані з буферу отриманих даних або можуть не повертати. Реалізації можуть мати таку семантику для нових отриманих даних: усі отримані сегменти з даними від іншої сторони TCP-з'єднання підтверджуються, але їхній вміст ігнорується; усі отримані сегменти з даними від іншої сторони TCP-з'єднання підтверджуються та зберігаються в

буфері отриманих даних сокета, але процес їх не отримує (у результаті буфер отриманих даних може бути заповнений повністю); усі отримані сегменти з даними від іншої сторони TCP-з'єднання підтверджуються та зберігаються в буфері отриманих даних сокета, перше застосування системного виклику `read()` для TCP-сокета повертає нуль байтів, але наступне застосування цього системного виклику для цього сокета повертає отримані дані (у результаті буфер отриманих даних може бути заповнений повністю).

Використання `SHUT_RDWR` у виклику `shutdown()`, застосованого для TCP-сокета, має бути еквівалентно використанню `SHUT_RD` у першому виклику `shutdown()` і `SHUT_RW` у другому виклику `shutdown()`, застосованого для TCP-сокета. Семантика використання `SHUT_RDWR` або `SHUT_RD` та `SHUT_WR` для одного TCP-сокета може бути або може не бути об'єднанням семантик `SHUT_RD` та `SHUT_WR`. У другому випадку реалізація може відправити іншій стороні TCP-з'єднання сегмент з прапорцем RST (розірвання TCP-з'єднання) у разі отримання нових даних. Тобто зовсім не так, якби був використаний `SHUT_RD` в одному виклику `shutdown()`, застосованого для TCP-сокета.

Можна зробити висновок, що використовувати SHUT_RD або SHUT_RDWR у shutdown(), застосованого для TCP-сокета, нема сенсу в переносному мережевому програмуванні.

Системний виклик shutdown() застосовується для TCP-сокета, асоційованого з вказаним дескриптором файлу, незважаючи на кількість дескрипторів файлів, з якими ще асоційований цей сокет. Системний виклик close() застосовується для дескриптора файлу, а TCP-сокет, асоційований з цим дескриптором файлу, закривається тільки тоді, коли цей сокет не асоційований ні з якими з іншими дескрипторами файлів. Семантика системного виклику shutdown(), застосованого для TCP-сокета, не еквівалентна семантиці закриттю TCP-сокета, через неможливість реалізувати семантику SHUT_RD у shutdown() для TCP-сокета на рівні TCP.

Опцію сокета SO_LINGER можна застосовувати для TCP-сокета тим самим вплинути на усталену семантику виконання close() (асинхронне закриття TCP-з’єднання відносно потоку, який закрив TCP-сокет) але POSIX не визначає семантику цієї опції, застосованої для TCP-сокета. Інформація, яка надається далі, відповідає семантиці опції сокета SO_LINGER, застосованої для TCP-сокета,

яка реалізована в наявних системах. Далі об’єкт типу `struct linger` – це об’єкт, на який вказує аргумент системних викликів `setsockopt()` та `getsockopt()`, застосованих для TCP-сокета.

Якщо значення поля `l_onoff` в об’єкті типу `struct linger` дорівнює нулю, тоді опція `SO_LINGER` не застосовується для TCP-сокета і `close()` завершується не чекаючи на результат завершення TCP-з’єднання, це типове налаштування. Значення поля `l_linger` ігнорується.

Якщо значення поля `l_onoff` в об’єкті типу `struct linger` не дорівнює нулю, тоді опція `SO_LINGER` застосовується для TCP-сокета. Подальша семантика визначається значенням поля `l_linger`.

Якщо значення поля `l_linger` дорівнює нулю, тоді всі дані в буфері даних для відправлення ігноруються, а іншій стороні TCP-з’єднання відправляється сегмент з прапорцем RST. Інформація про це TCP-з’єднання забирається з системи, тобто TCP-з’єднання не переходить у стан TIME_WAIT. Відправлення програмою сегмента з прапорцем RST впливає безпосередньо на TCP.

Якщо значення поля `l_linger` не дорівнює нулю, тоді іншій стороні TCP-з'єднання відправляються всі дані в буфері даних для відправлення і відправляється сегмент з прапорцем FIN, `close()` чекає на отримання підтвердження на відправлений сегмент з прапорцем FIN або на завершення вказаного тайм-ауту в секундах у полі `l_linger`. У разі завершення вказаного тайм-ауту `close()` завершується з помилкою `EWOULDBLOCK`, але вказаний дескриптор файлу буде закрито, а асоційований з цим дескриптором файлу TCP-сокет буде закрито асинхронно відносно процесу.

Також треба зазначити, що якщо в буфері отриманих даних TCP-сокета є дані, тоді в разі закриття сокета іншій стороні з'єднання відправляється сегмент з прапорцем RST. Інформація про це TCP-з'єднання забирається з системи, тобто TCP-з'єднання не переходить у стан TIME_WAIT. Закриття сокета, у буфері отриманих даних якого є дані, впливає безпосередньо на TCP через відправлення сегмента з прапорцем RST.

# Багатопотоковий ітеративний сервер

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

struct thread_info {
    pthread_t tid;
    unsigned int thread_idx;
    int listenfd;
};

static void *
thread_main(void *arg)
{
    struct thread_info *thread_info;
    unsigned int thread_idx;
    int listenfd, sockfd;

    thread_info = arg;
    listenfd = thread_info->listenfd;
    thread_idx = thread_info->thread_idx;
    for (;;) {
        sockfd = accept(listenfd, NULL, 0);
```

```
        if (sockfd < 0)
            return NULL;
        printf("Thread %u accepted connection\n", thread_idx);
        // Data transfer.
        if (close(sockfd) < 0)
            exit_err("close()");
    }
    return NULL;
}

int
main(void)
{
    const unsigned int thread_num = 5;
    struct thread_info thread_info[thread_num];
    struct sockaddr_in srv_sin4 = { 0 };
    in_port_t srv_port = htons(1234);
    unsigned int i;
    int listenfd, optflag, rv;

    listenfd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (listenfd < 0)
        exit_err("socket()");

    optflag = 1;
    if (setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &optflag,
        sizeof(optflag)) < 0
    ) {
```

```
        exit_err("setsockopt()");
    }

    srv_sin4.sin_family = AF_INET;
    srv_sin4.sin_port = srv_port;
    srv_sin4.sin_addr.s_addr = INADDR_ANY;
    if (bind(listenfd, (struct sockaddr *)&srv_sin4, sizeof(srv_sin4)) < 0)
        exit_err("bind()");

    if (listen(listenfd, 10) < 0)
        exit_err("listen()");

    for (i = 0; i < thread_num; ++i) {
        thread_info[i].listenfd = listenfd;
        thread_info[i].thread_idx = i;
        rv = pthread_create(&thread_info[i].tid, NULL, thread_main,
            &thread_info[i]);
        if (rv != 0)
            exit_errn(rv, "pthread_create()");
    }

    pause();
}
```

# Багатопотоковий паралельний сервер

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

struct thread_info {
    int sockfd;
};

static void *
thread_main(void *arg)
{
    struct thread_info *thread_info;
    int sockfd;

    pthread_detach(pthread_self());
    thread_info = arg;
    sockfd = thread_info->sockfd;
    free(thread_info);
    printf("Handle connection in new thread\n");
    // Data transfer.
    if (close(sockfd) < 0)
```

```
        exit_err("close()");
    return NULL;
}

int
main(void)
{
    const unsigned int thread_num = 5;
    struct thread_info *thread_info;
    struct sockaddr_in srv_sin4 = { 0 };
    pthread_t tid;
    in_port_t srv_port = htons(1234);
    unsigned int i;
    int listenfd, optflag, rv;

    listenfd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (listenfd < 0)
        exit_err("socket()");

    optflag = 1;
    if (setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &optflag,
        sizeof(optflag)) < 0
    ) {
        exit_err("setsockopt()");
    }

    srv_sin4.sin_family = AF_INET;
    srv_sin4.sin_port = srv_port;
```

```
    srv_sin4.sin_addr.s_addr = INADDR_ANY;
    if (bind(listenfd, (struct sockaddr *)&srv_sin4, sizeof(srv_sin4)) < 0)
        exit_err("bind()");

    if (listen(listenfd, 10) < 0)
        exit_err("listen()");

    for (i = 0; i < thread_num; ++i) {
        thread_info = malloc(sizeof(*thread_info));
        if (thread_info == NULL)
            exit_err("malloc()");
        thread_info->sockfd = accept(listenfd, NULL, NULL);
        rv = pthread_create(&tid, NULL, thread_main, thread_info);
        if (rv != 0)
            exit_errn(rv, "pthread_create()");
    }
}
```