

# ネットワークコンピューティング 第4回

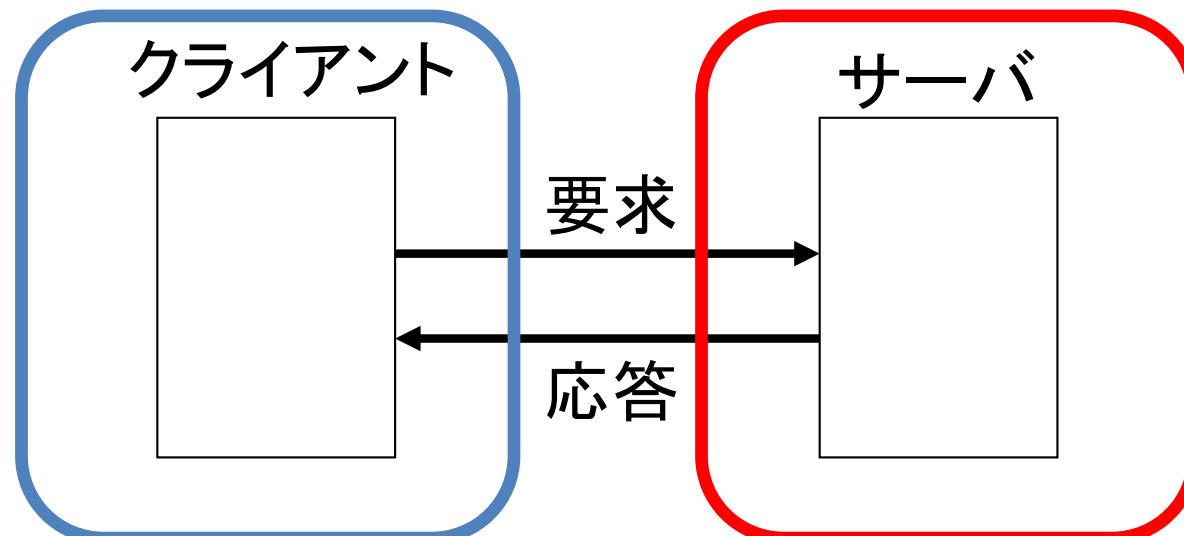
中山 雅哉 (m.nakayama@m.cnl.t.u-tokyo.ac.jp)  
関谷 勇司 (sekiya@nc.u-tokyo.ac.jp)

# 授業に関する情報

- 授業スライド、連絡事項、課題等に関する連絡
  - Web
    - <http://lecture.sekiya-lab.info/>
  - Mail
    - lecture@sekiya-lab.info
- 実験用ホスト
  - Resources
    - login1.sekiya-lab.info
    - login2.sekiya-lab.info
      - ともに外部から ssh で login できる Linux マシン

# サーバ / クライアントモデル (基本)

- クライアント
  - 処理を要求するプログラム
- サーバ
  - (クライアントからの) 要求を受けて処理をするプログラム

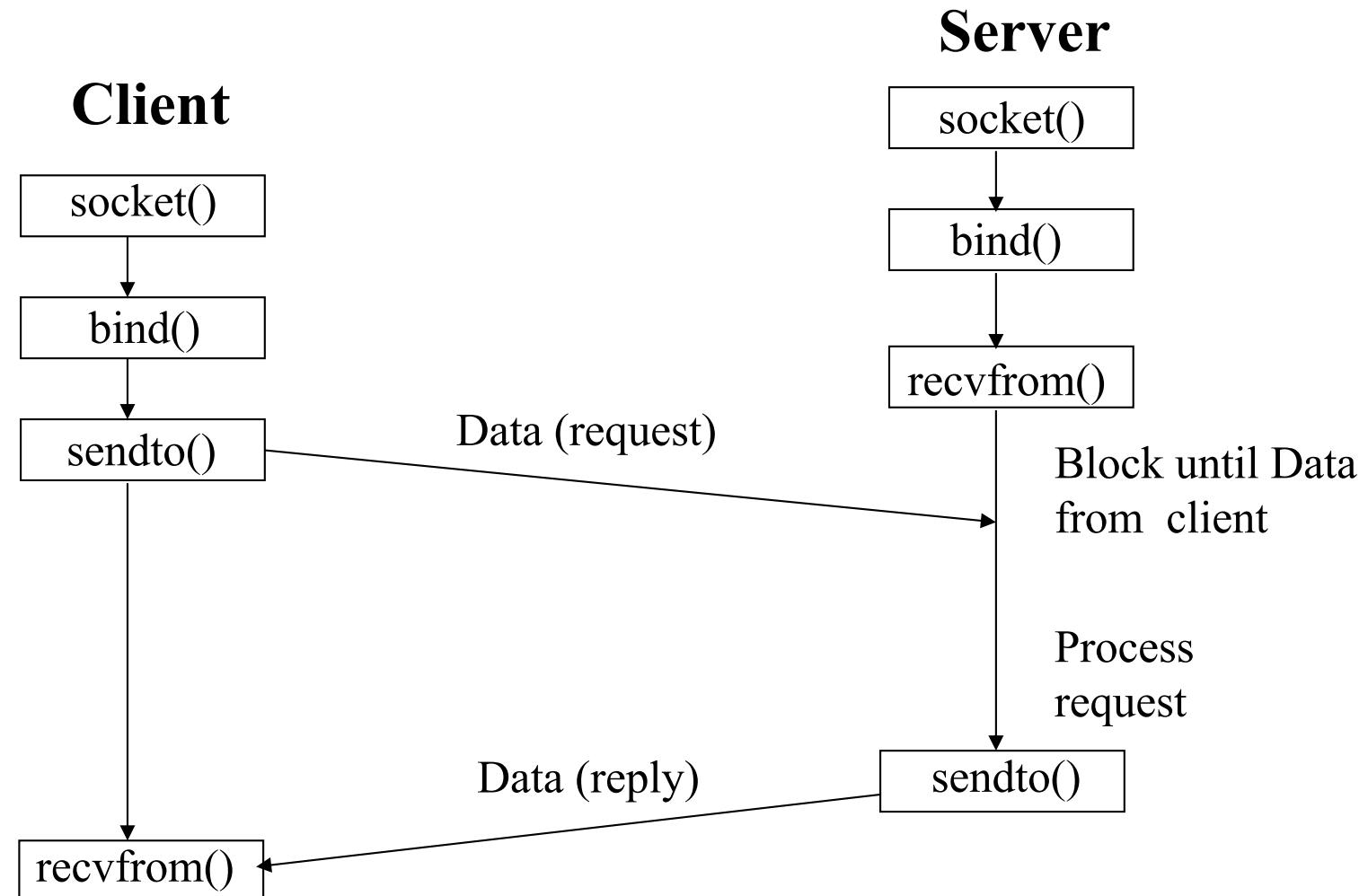


これまでの内容と  
課題1

今後の内容の中心

# UDP (サーバ) の (待受け) 状態

# Datagram example (UDP)

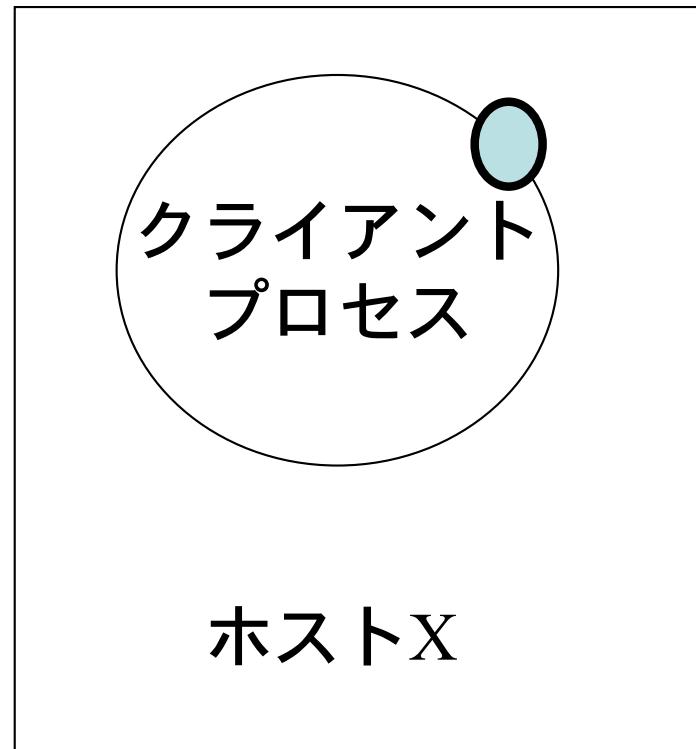


# サーバ側での待受け状態 (UDP)

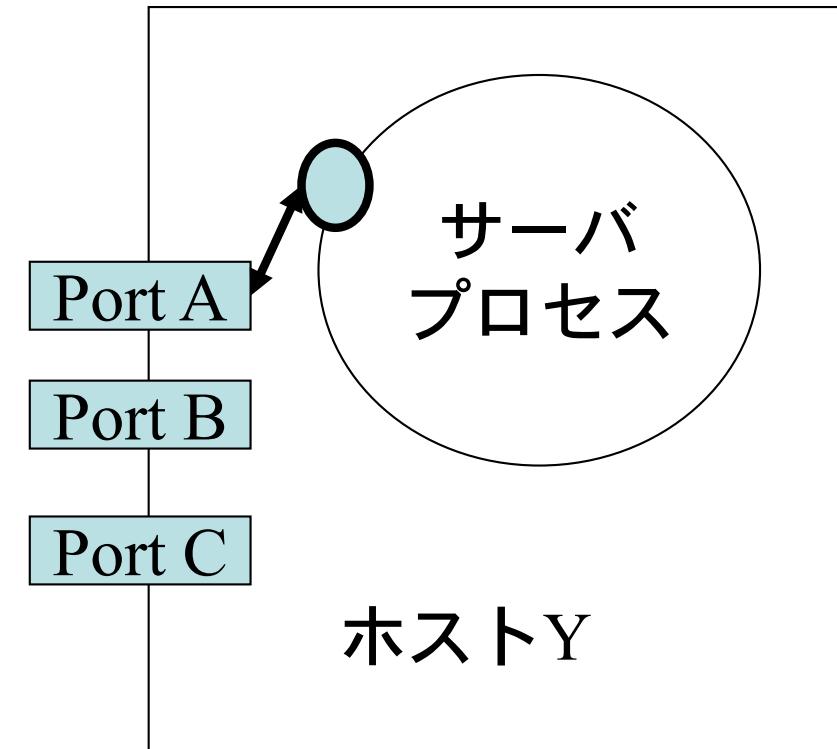
Proto LocalAddress  
**UDP** \*.\*A

ForeignAddress  
\*.\*

State



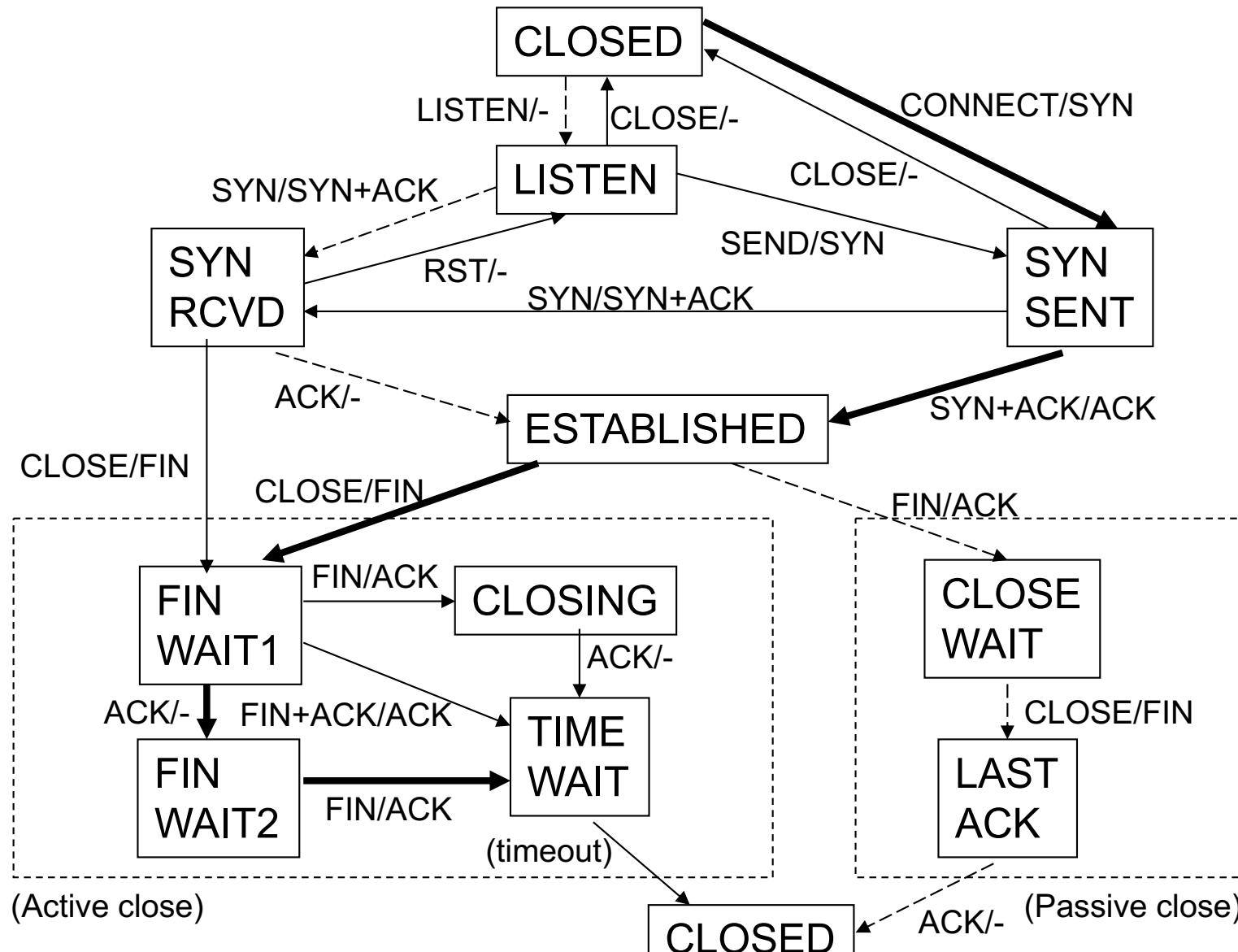
IP Address: 10.0.1.1



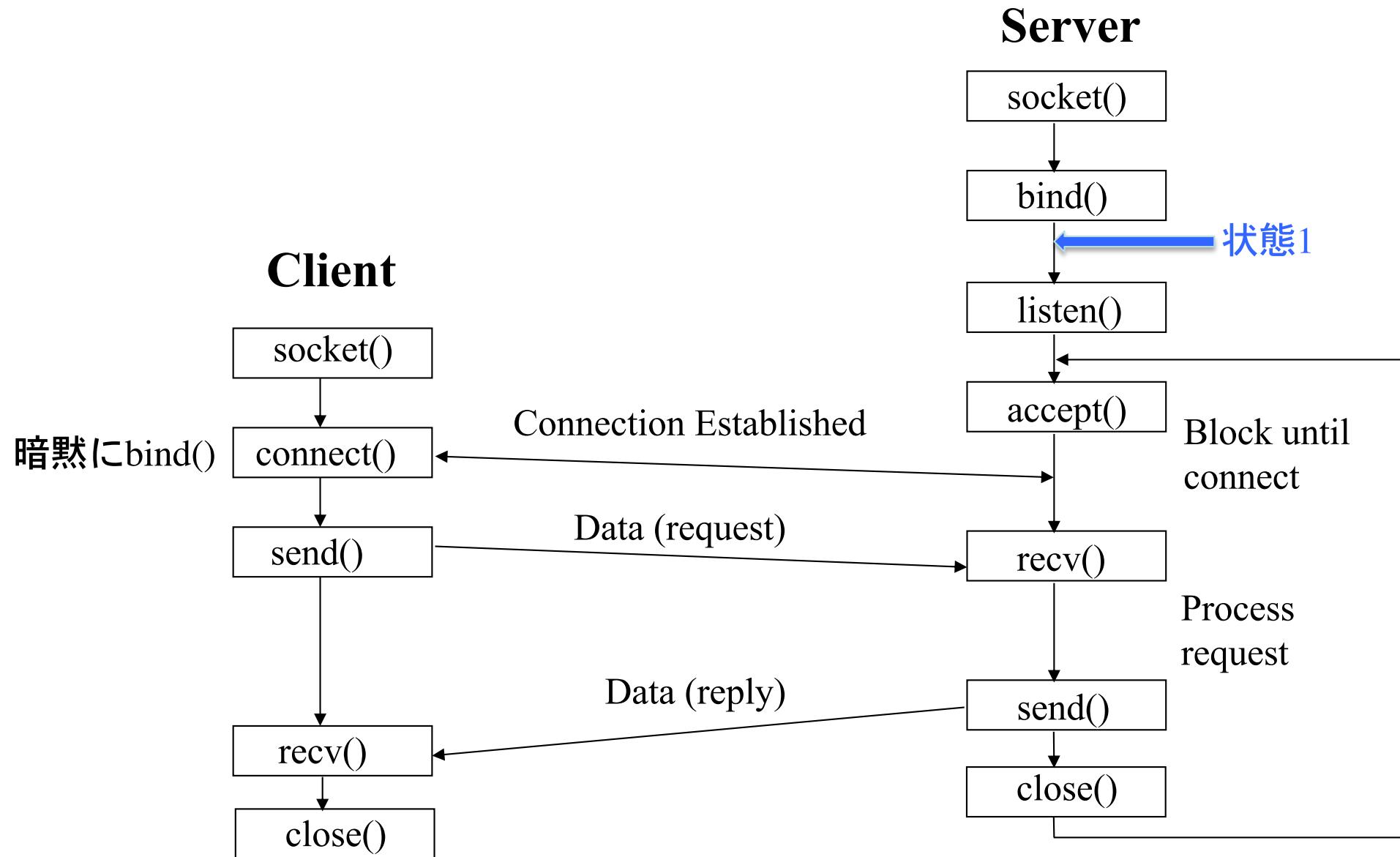
IP Address: 10.0.2.1

# TCP の状態遷移

# TCP State Transition Diagram

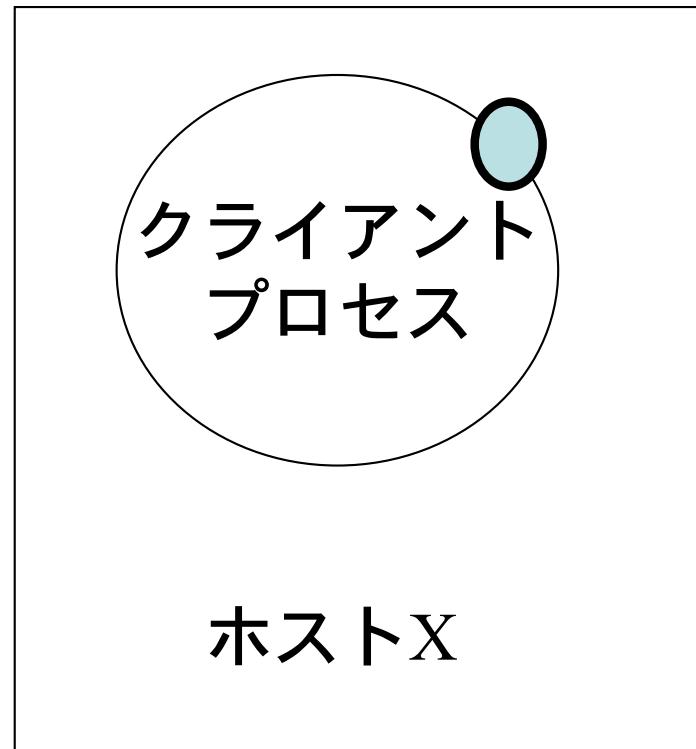


# Stream example (TCP)

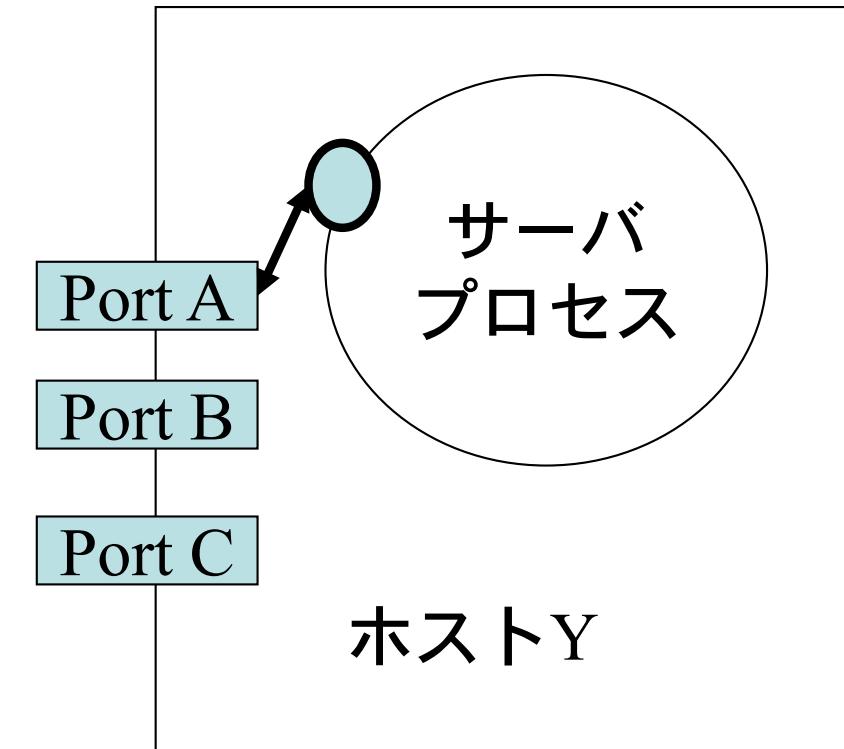


# サーバがbindした状態 (TCP) : 状態1

Proto	LocalAddress	ForeignAddress	State
TCP	*.A	*.*	CLOSED

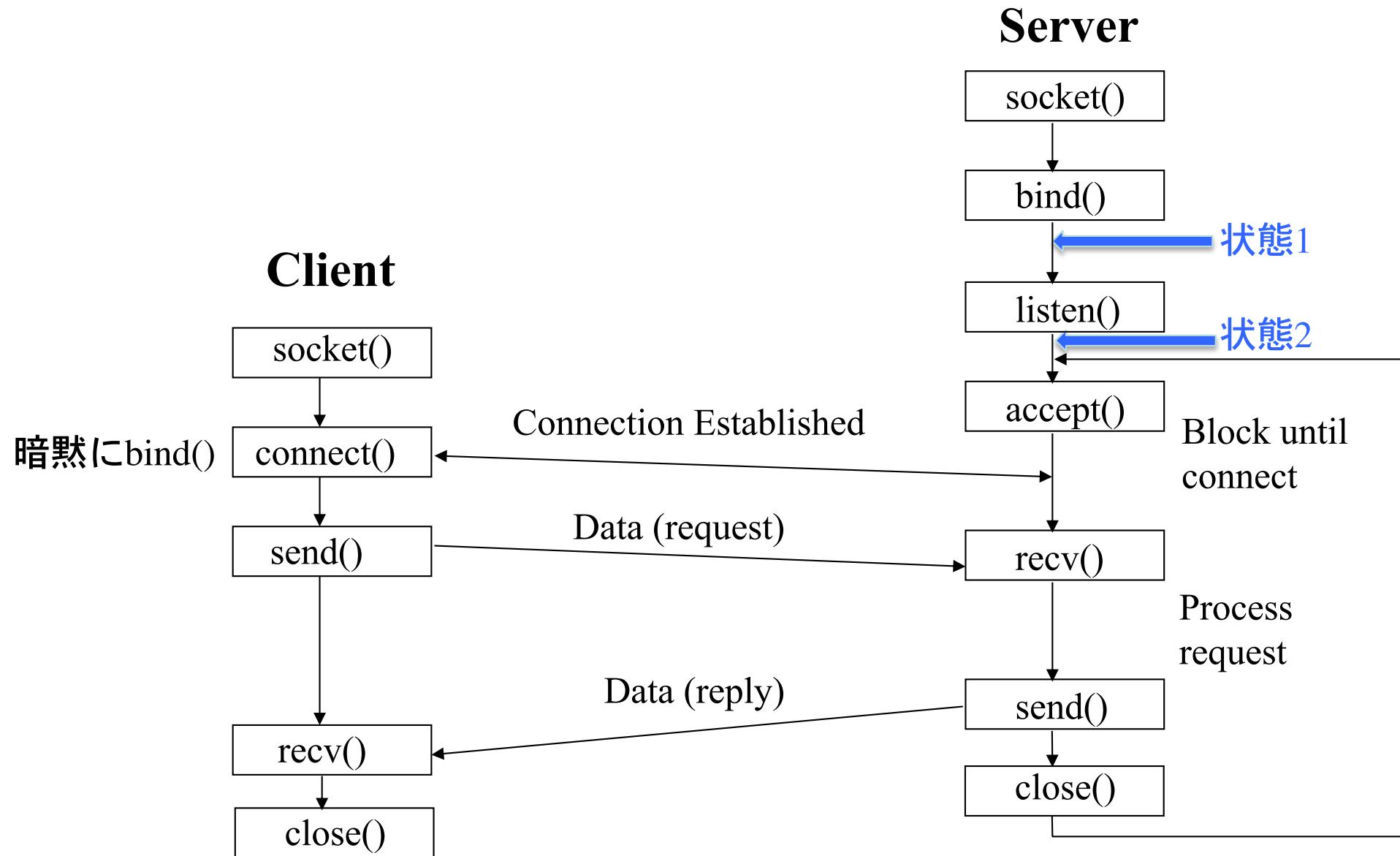


IP Address: 10.0.1.1



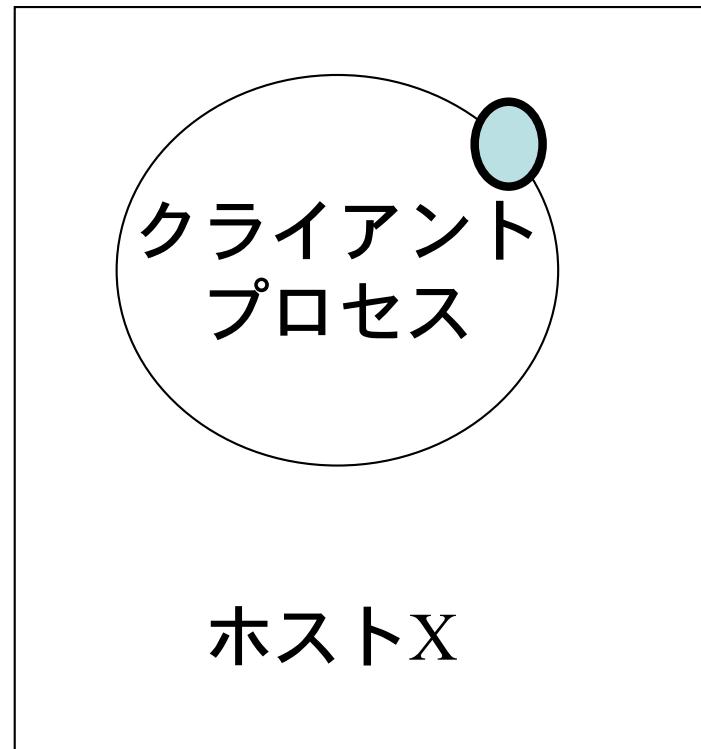
IP Address: 10.0.2.1

# Stream example (TCP)



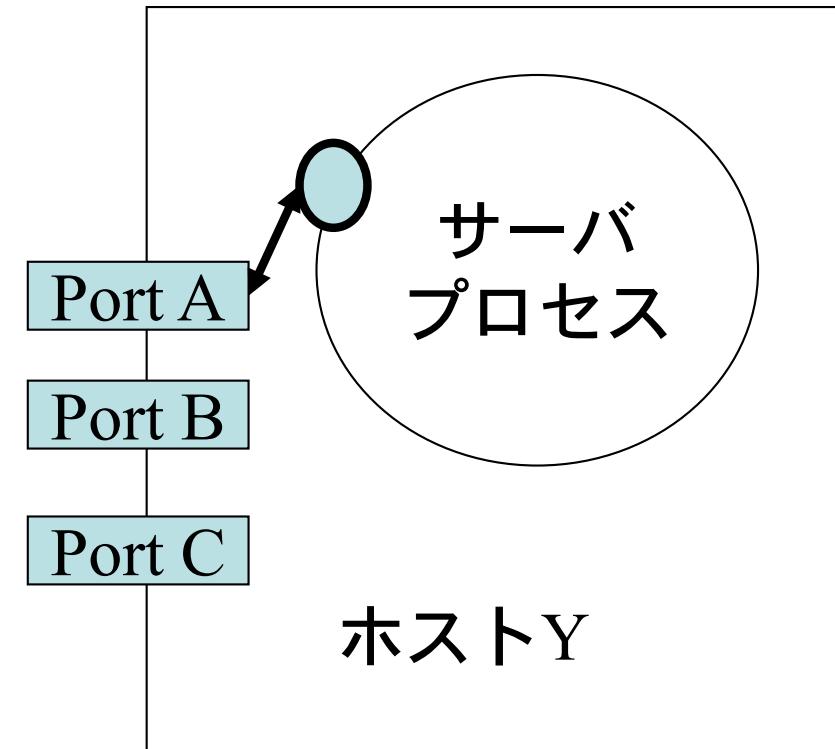
# サーバがlistenした状態 (TCP) : 状態2

Proto	LocalAddress	ForeignAddress	State
TCP	*.A	*.*	LISTEN



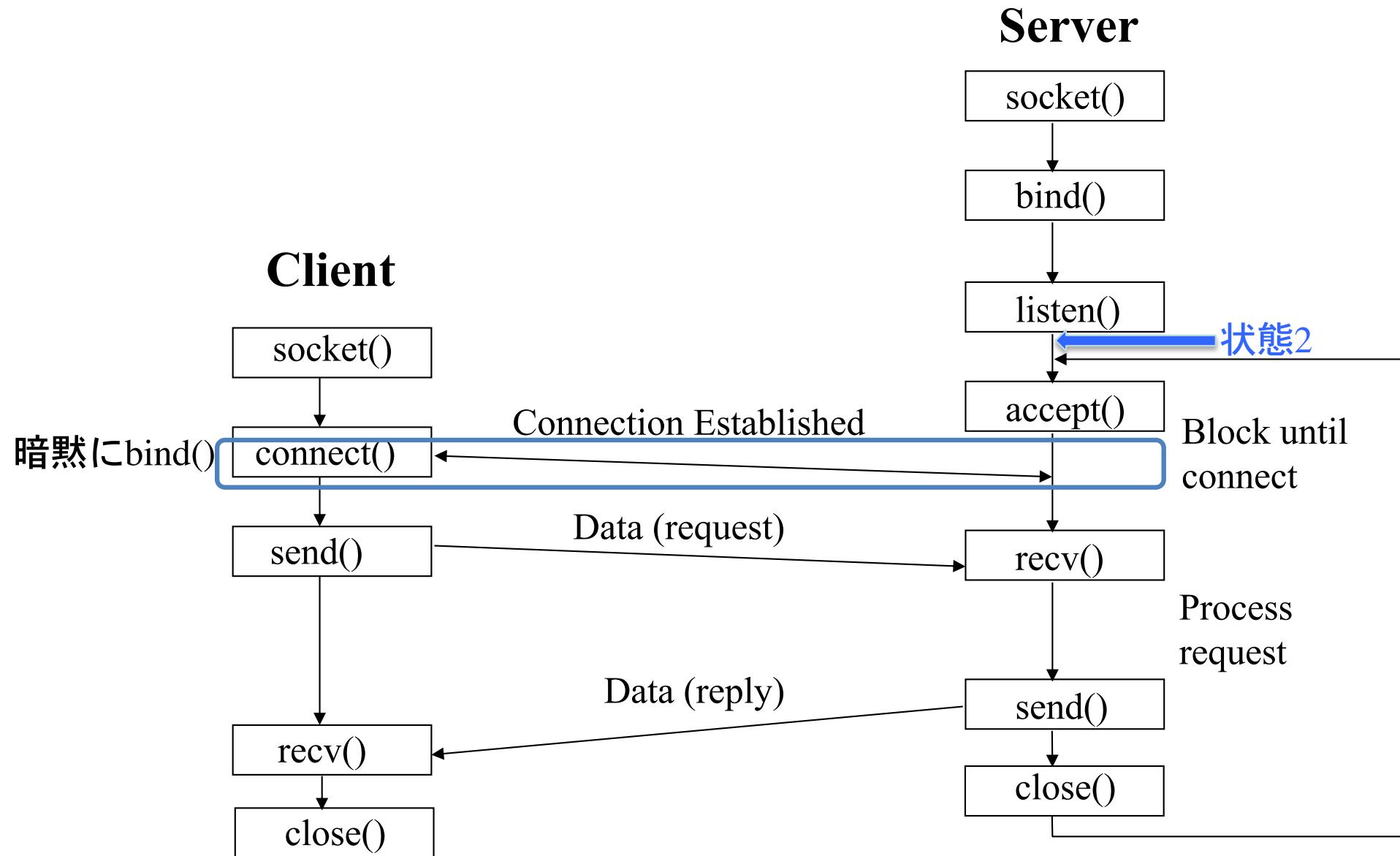
IP Address: 10.0.1.1

Proto	LocalAddress	ForeignAddress	State
TCP	*.A	*.*	LISTEN

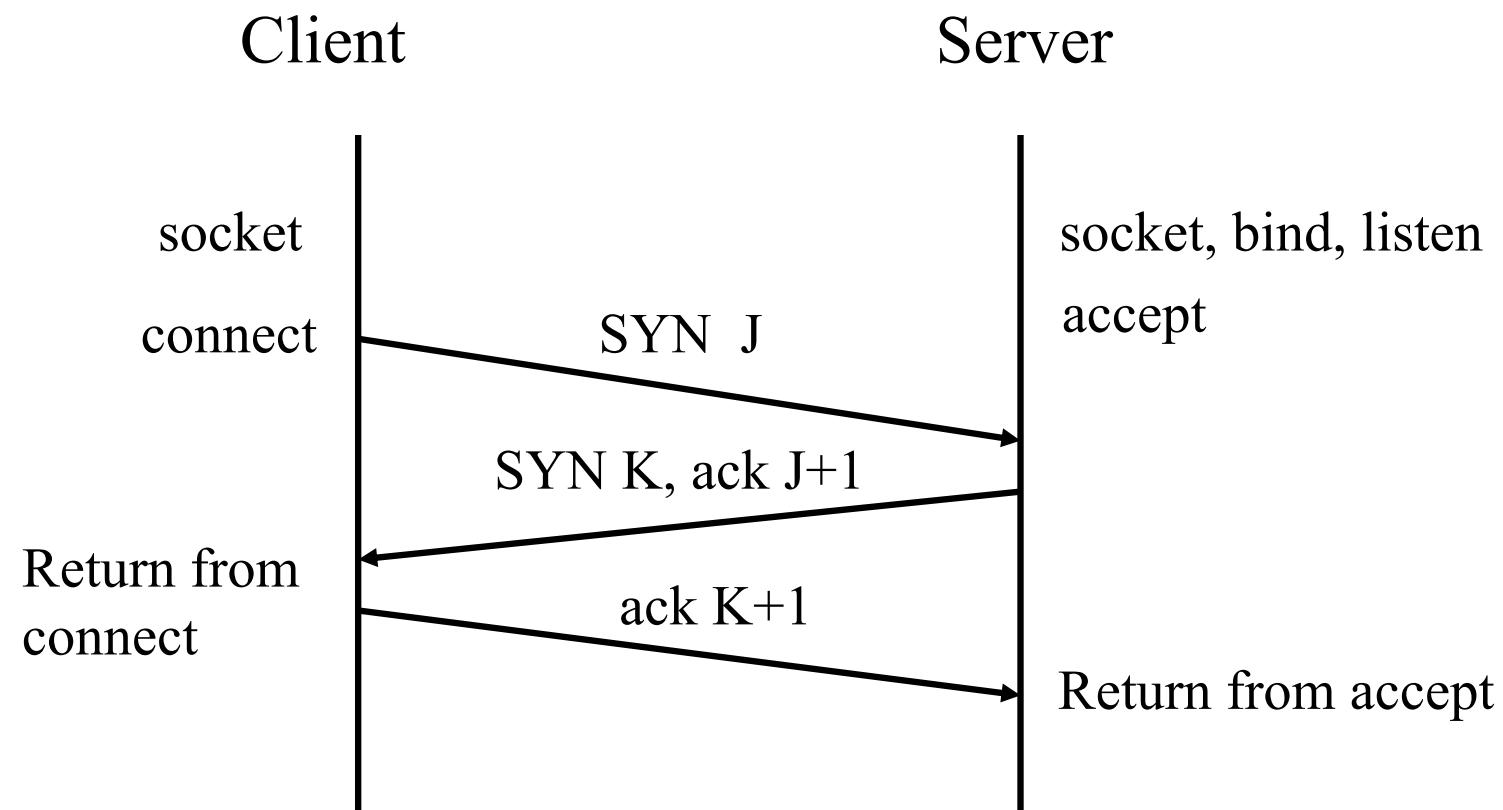


IP Address: 10.0.2.1

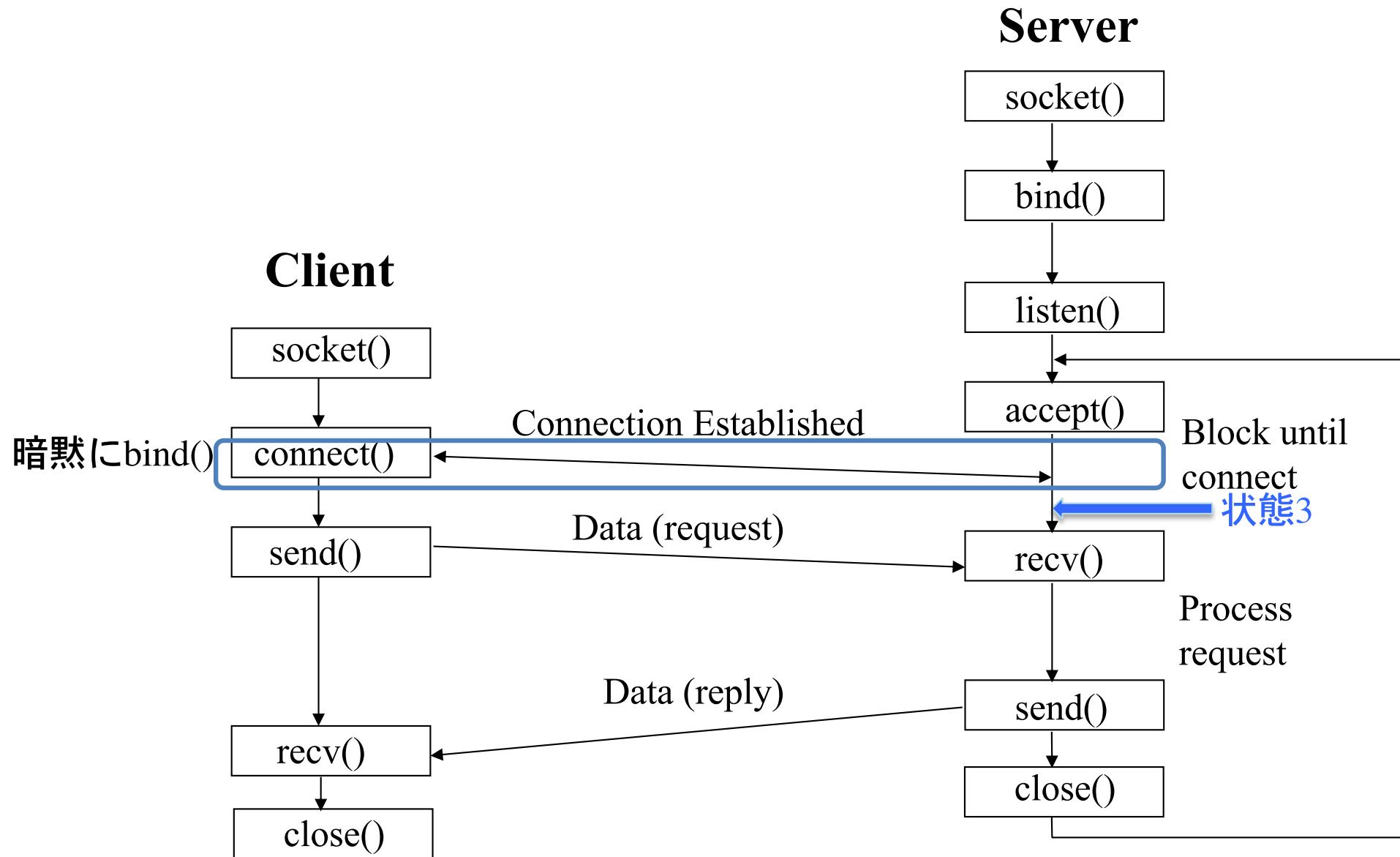
# Stream example (TCP)



# Establish TCP (3 way handshake)

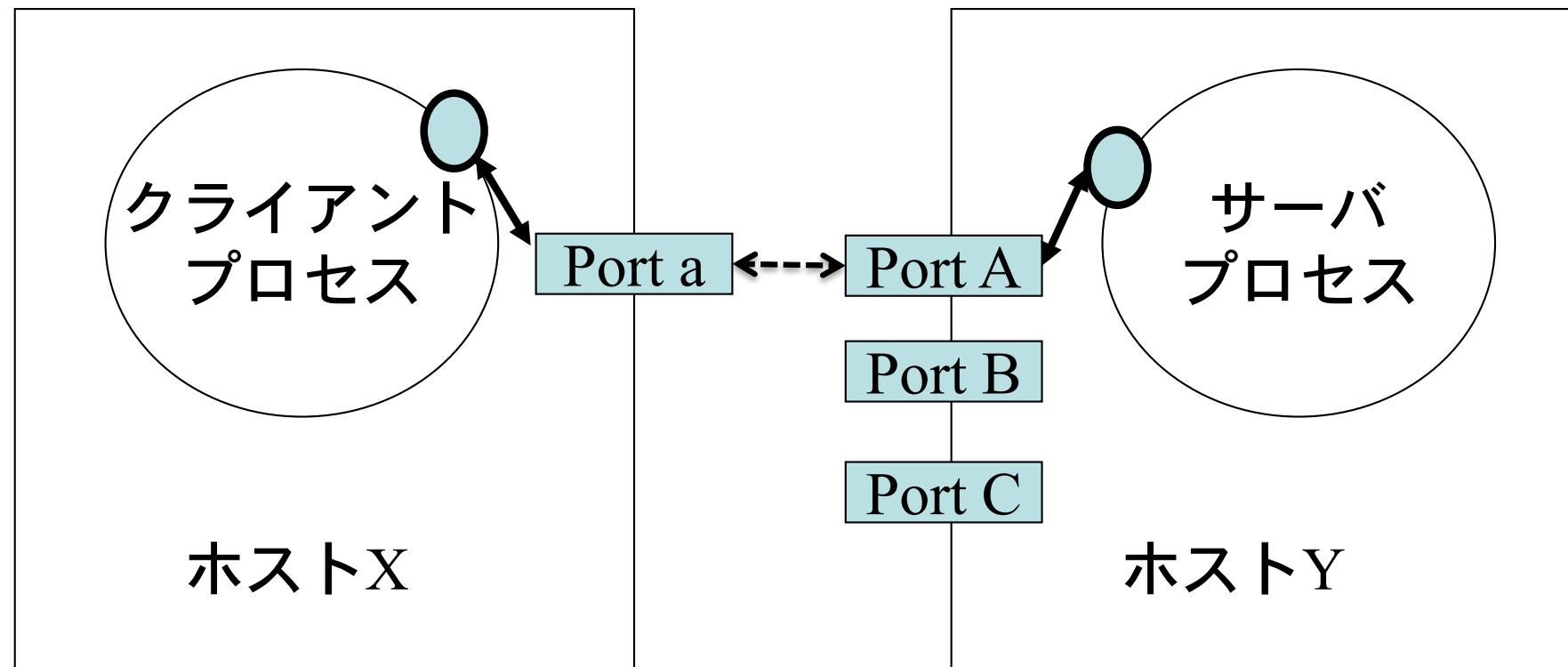


# Stream example (TCP)

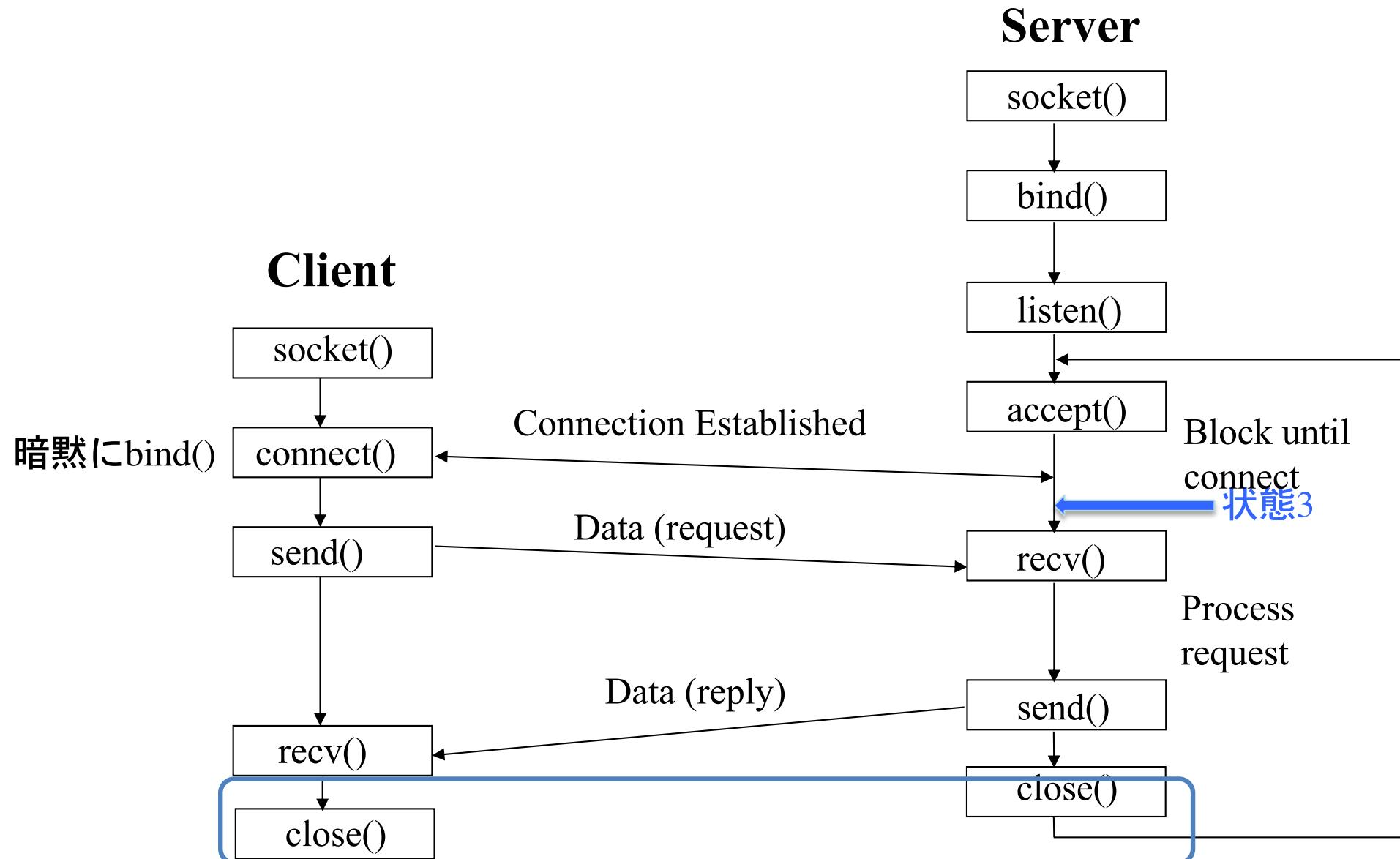


# connect() -> accept() した状態 (TCP) : 状態3

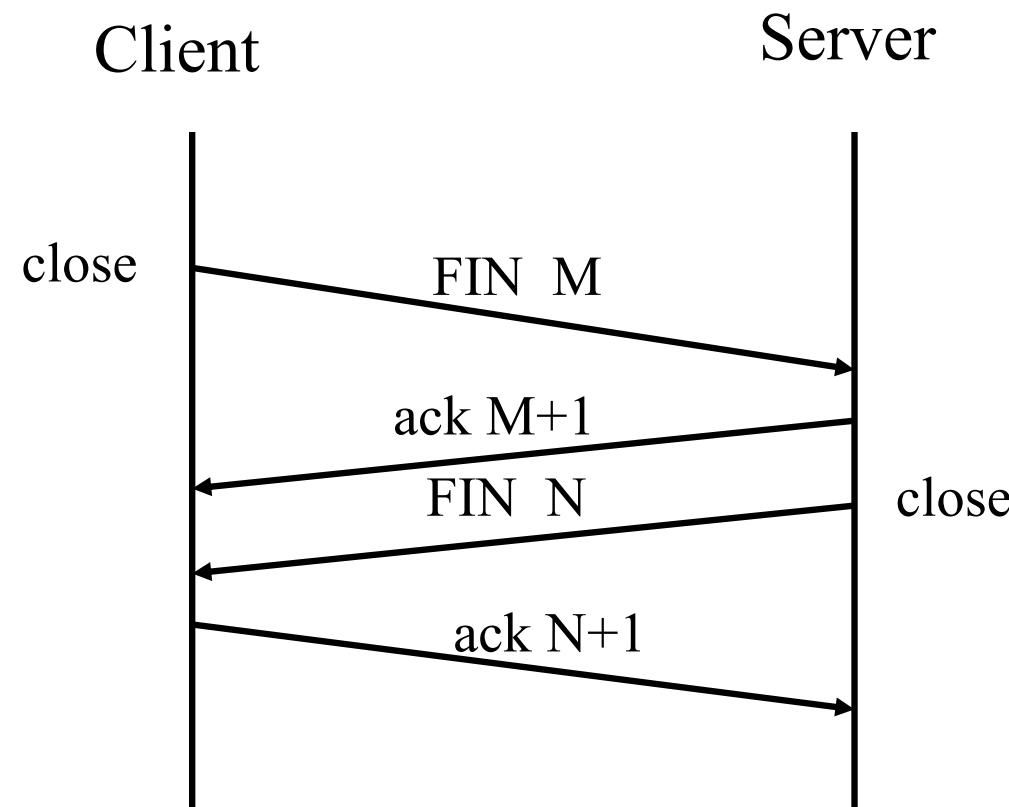
Proto	LocalAddress	ForeignAddress	State
TCP	*.A	*.*	LISTEN
TCP	10.0.2.1.A	10.0.1.1.a	ESTABLISHED



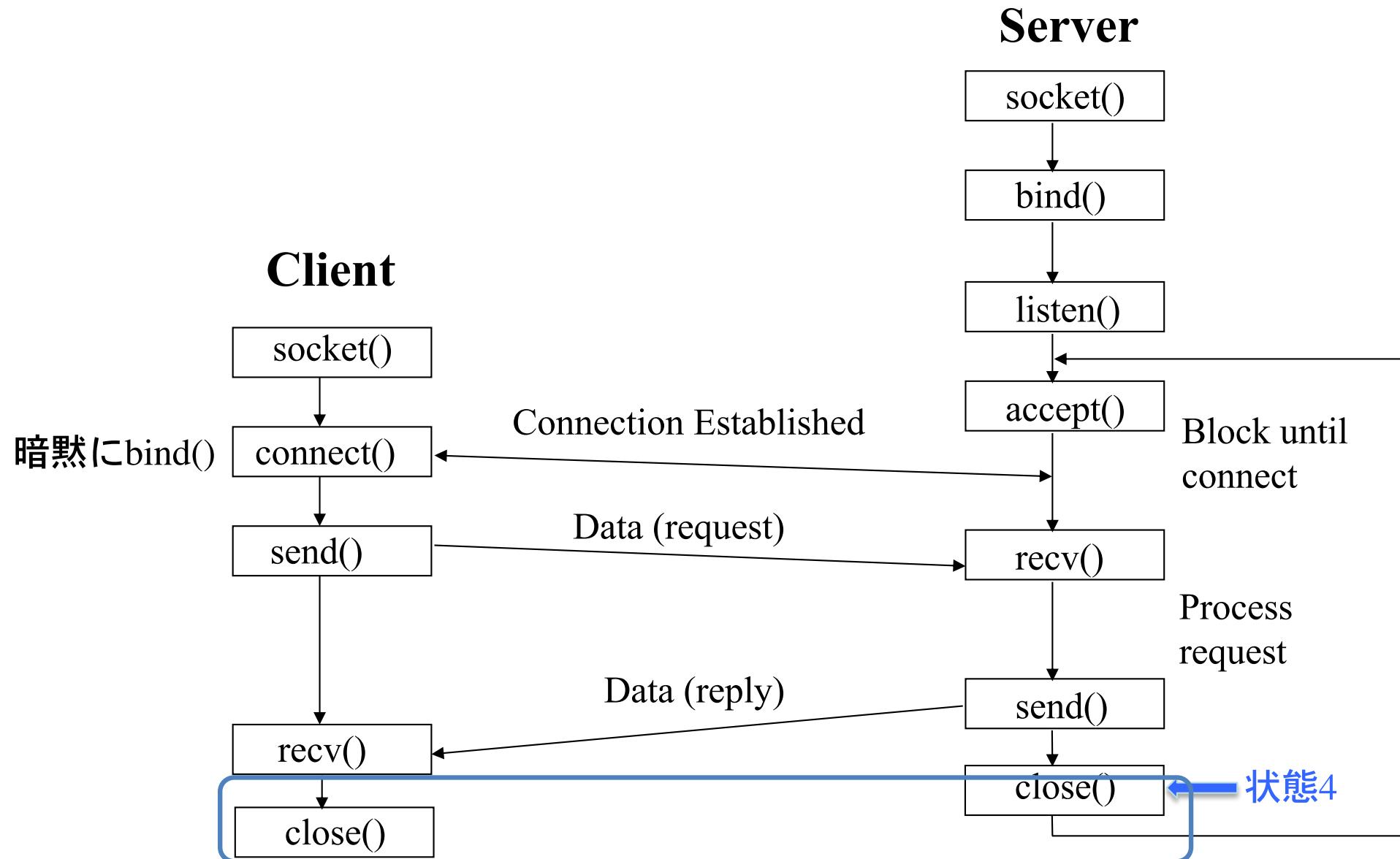
# Stream example (TCP)



# Terminate TCP connection



# Stream example (TCP)



# close() 中の状態 (TCP) : 状態4

**Proto LocalAddress**

TCP \*.\*A

TCP 10.0.2.1.A

**ForeignAddress**

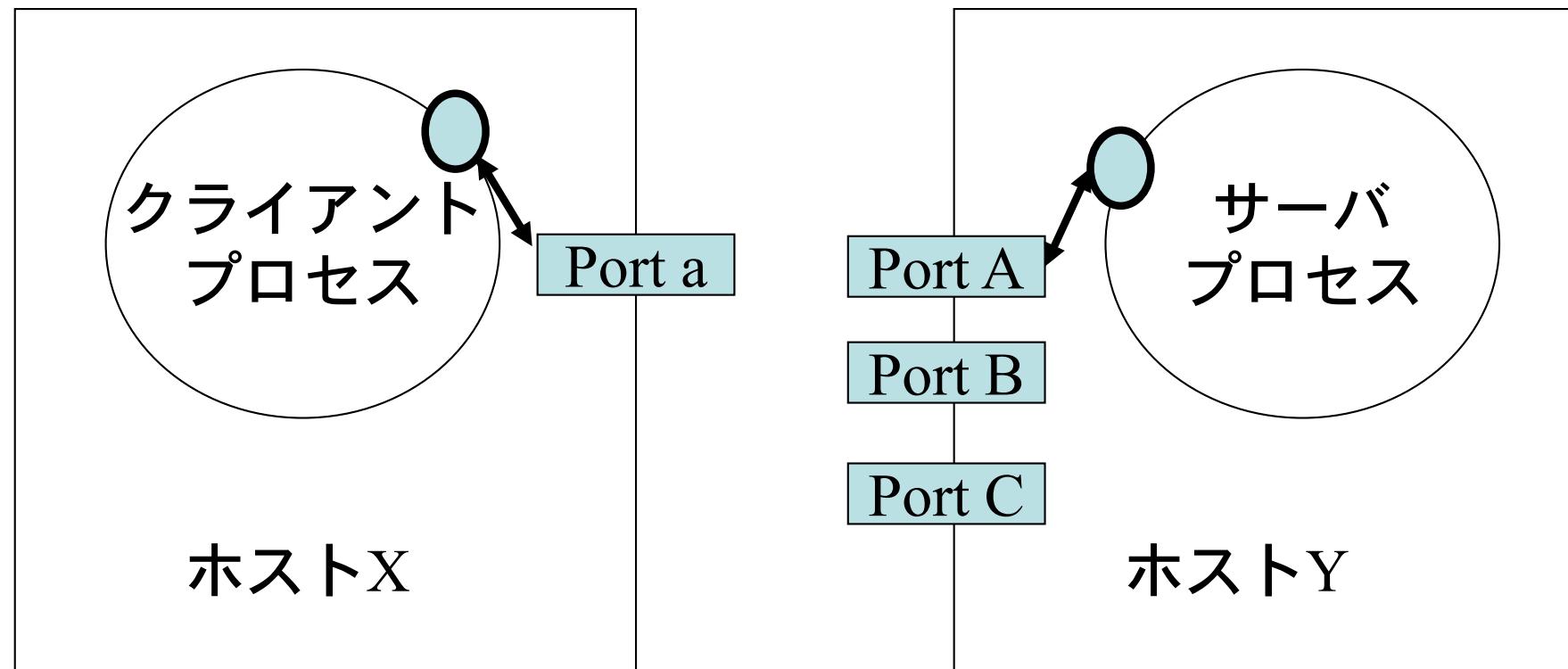
\*.\*

10.0.1.1.a

**State**

LISTEN

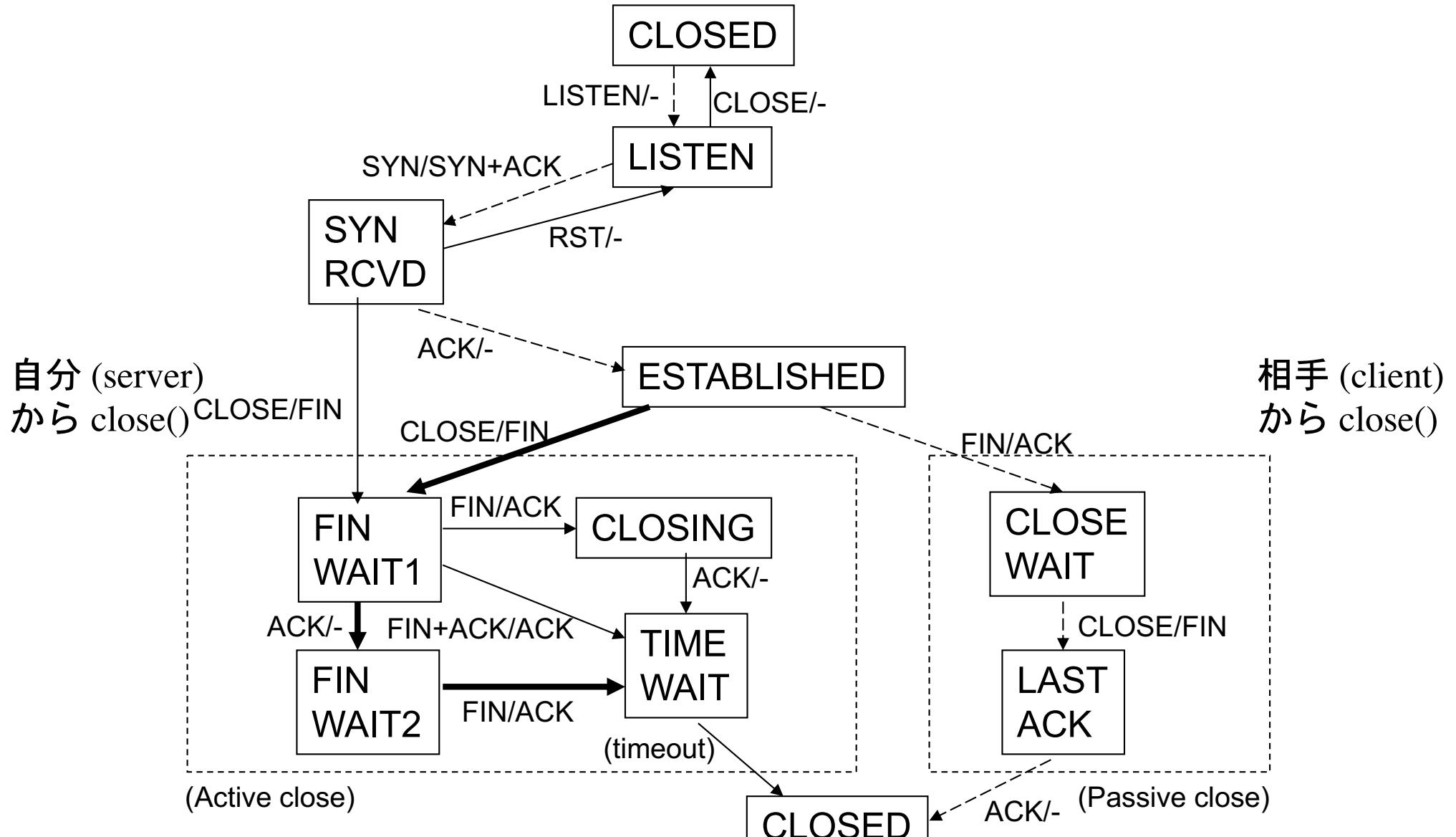
TIME\_WAIT



IP Address: 10.0.1.1

IP Address: 10.0.2.1

# TCP Server State Transition Diagram



# Results of netstat

```
# netstat -an
```

Active Internet connections (including servers)

Proto	Recv-Q	Send-Q	Local Address	Foreign Address	(state)
tcp4	0	0	192.168.1.20.52096	133.11.205.167.13	SYN_SENT
tcp4	0	0	192.168.1.20.52100	133.11.205.167.13	ESTABLISHED
tcp4	0	0	192.168.1.20.52103	133.11.205.167.13	TIME_WAIT

...

# Results of tcpdump

Clinet : 192.168.1.20

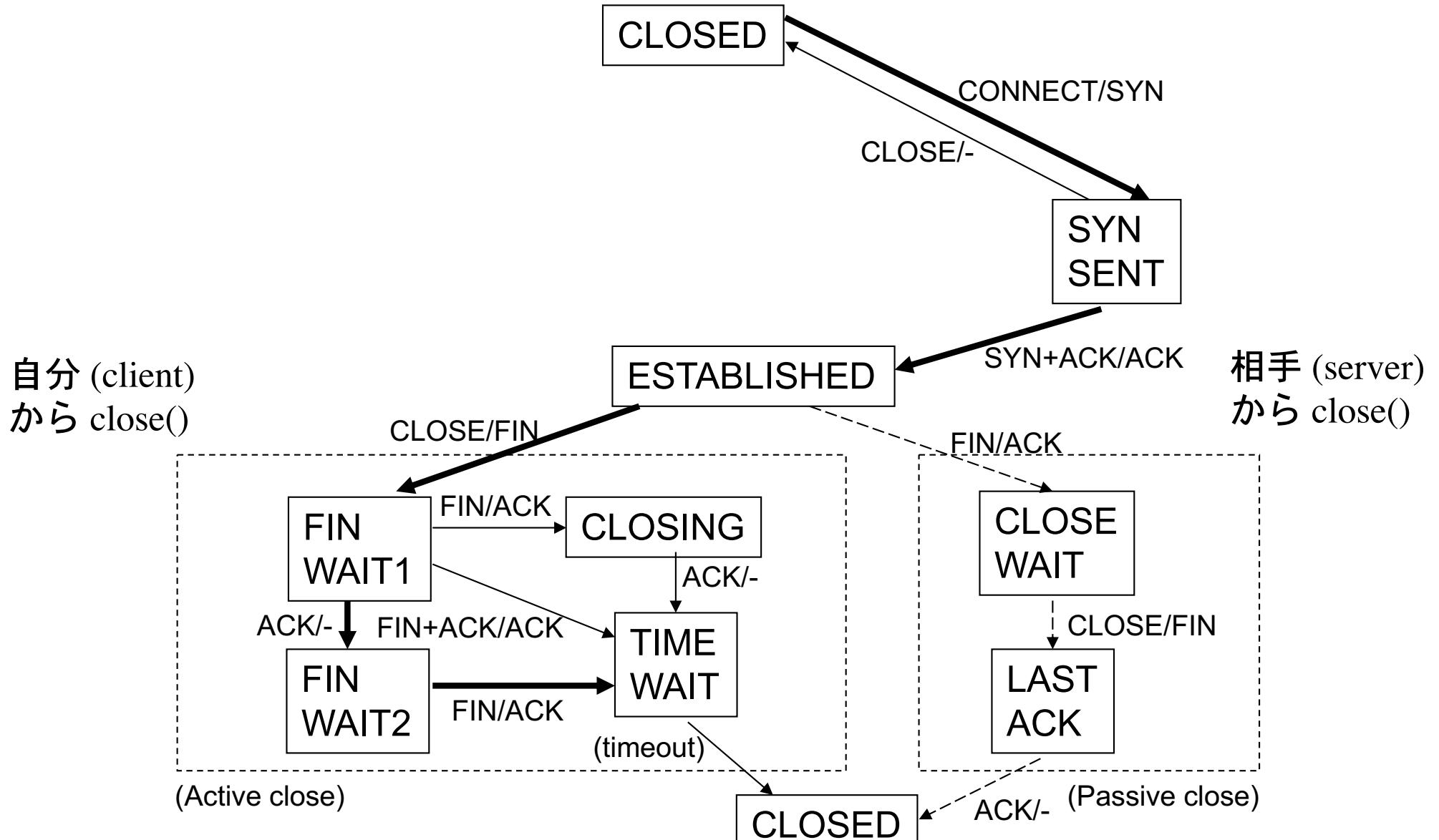
Server : 133.11.205.167

```
# tcpdump -S -n
```

```
22:58:54.558383 IP 192.168.1.20.52116 > 133.11.205.167.13: Flags [S], seq 246930  
8377, win 65535, options [mss 1460,nop,wscale 3,nop,nop,TS val 820465661 ecr  
0,sackOK,eol], length 0  
22:58:54.654911 IP 133.11.205.167.13 > 192.168.1.20.52116: Flags [S.], seq 31457  
80173, ack 2469308378, win 65535, options [mss 1372,nop,wscale 3,sackOK,TS val 2  
217087031 ecr 820465661], length 0  
22:58:54.654973 IP 192.168.1.20.52116 > 133.11.205.167.13: Flags [.], ack 3145780174,  
win 65535, options [nop,nop,TS val 820465662 ecr 2217087031], length 0
```

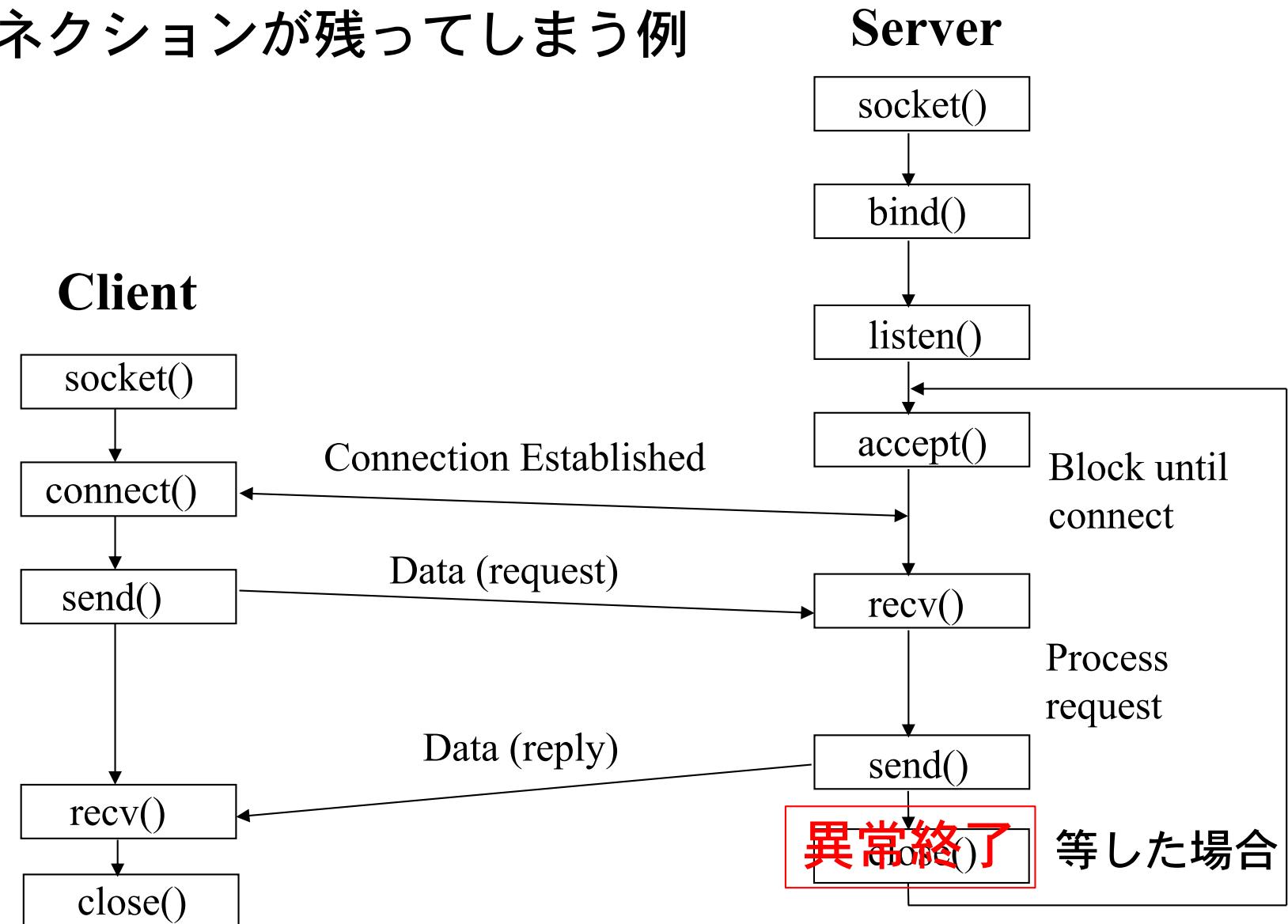
```
22:58:54.785113 IP 133.11.205.167.13 > 192.168.1.20.52116: Flags [FP.], seq 3145  
780174:3145780200, ack 2469308378, win 8330, options [nop,nop,TS val 2217087043  
ecr 820465662], length 26  
22:58:54.785148 IP 192.168.1.20.52116 > 133.11.205.167.13: Flags [.], ack 3145780201,  
win 65535, options [nop,nop,TS val 820465664 ecr 2217087043], length  
022:59:03.665575 IP 192.168.1.20.52116 > 133.11.205.167.13: Flags [F.], seq  
2469308378, ack 3145780201, win 65535, options [nop,nop,TS val 820465752 ecr  
2217087043], length 0  
22:59:03.775580 IP 133.11.205.167.13 > 192.168.1.20.52116: Flags [.], ack 2469308379,  
win 8329, options [nop,nop,TS val 2217087945 ecr 820465752], length 0
```

# 参考: TCP Client State Transition Diagram



# Server programming における注意事項

TCPコネクションが残ってしまう例



## bind : Address already in use

- サーバのプログラムを再起動しようとすると。。。
  - bind: Address already in use

のように、エラーで再起動できない場合がある

- これは、TCP の状態に TIME\_WAIT や CLOSE\_WAIT
  - 終了処理が完成していない
  - 終了処理後、一定時間が経過していない

があるため

- しばらくの間、同一アドレス/ポートでサーバのプログラムを起動することができない



- SO\_REUSEADDR を利用すると TIME\_WAIT 状態でもサーバプログラムを再起動することが可能となる

## setsockopt()

- ```
int setsockopt(int sockfd, int level, int optname,
               const void *optval, socklen_t optlen);
```
- ソケットに対して様々なオプションを指定する関数
- level = SOL\_SOCKET
- optname = SO\_REUSEADDR, SO\_REUSEPORT, ...
- 返り値
  - 成功 0
  - エラー -1

# SO\_REUSEADDR の利用例

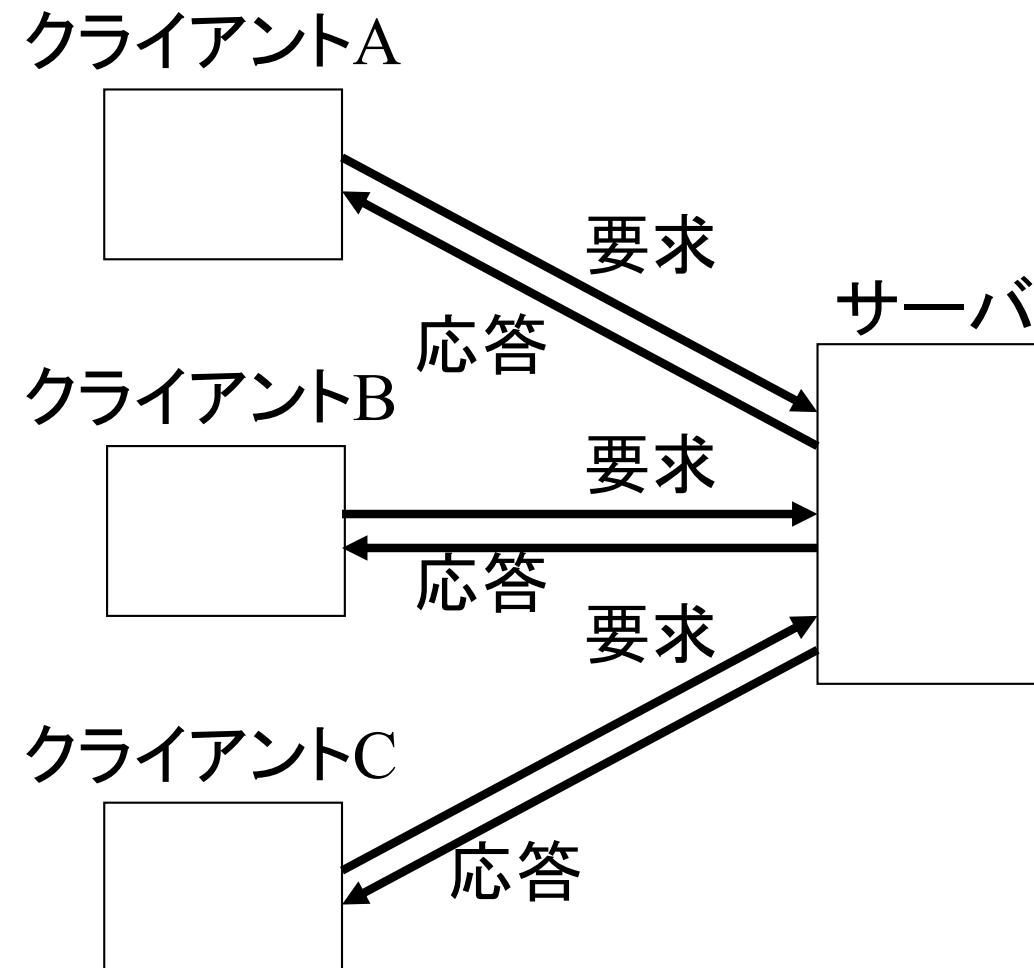
```
int yes = 1;

sock = socket(AF_INET, SOCK_STREAM, 0);
addr.sin_family = AF_INET;
addr.sin_port = htons(12345);
addr.sin_addr.s_addr = INADDR_ANY;

setsockopt(sock, SOL_SOCKET, SO_REUSEADDR,
           (const char *)&yes, sizeof(yes));

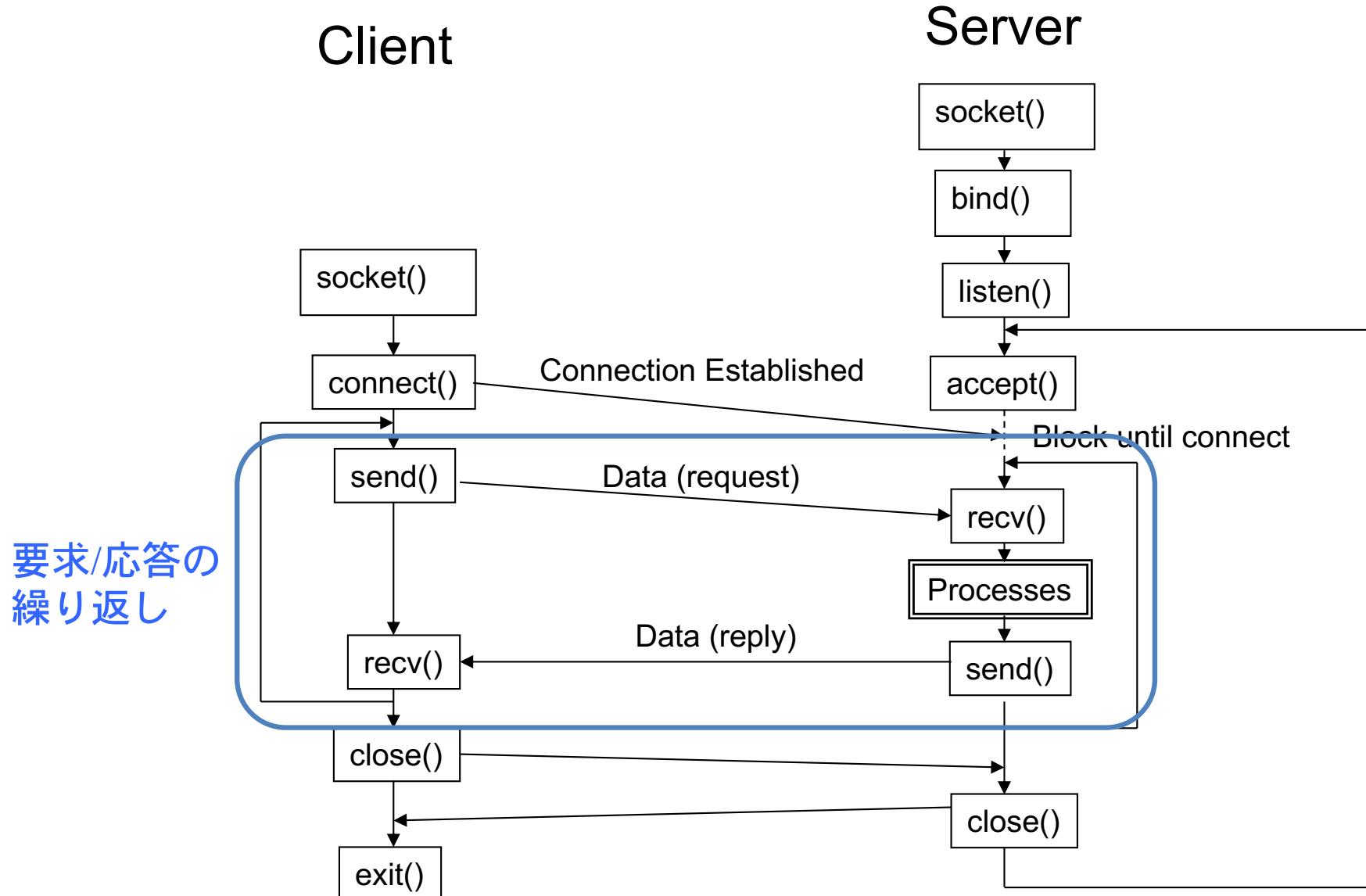
bind(sock, (struct sockaddr *)&addr, sizeof(addr));
```

# サーバ / クライアントモデル（一般）



サーバは、(同時に)複数のクライアントからの  
要求に対する処理をするケースにも対応が必要

# Basic Scenario (TCP)



## echo プロトコル

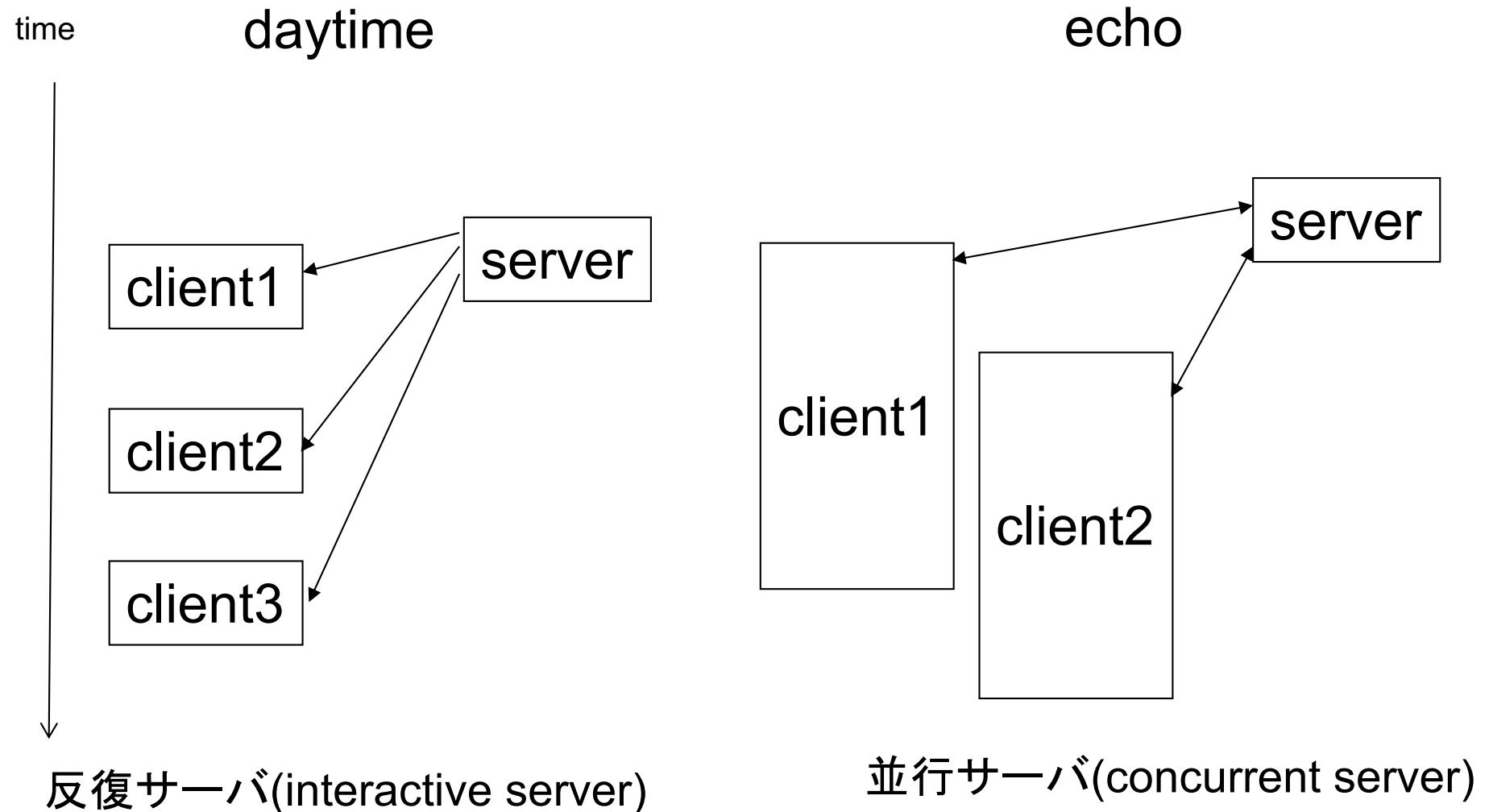
- RFC862 で定義されるプロトコル
- 受信したデータをそのまま送り返すプロトコル
- TCP/7 もしくは UDP/7 で提供される

```
nakayama% telnet momo4.nc.u-tokyo.ac.jp 7
Trying 133.11.205.167...
Connected to momo4.nc.u-tokyo.ac.jp.
Escape character is '^]'.
THIS IS TEST.
THIS IS TEST.
```

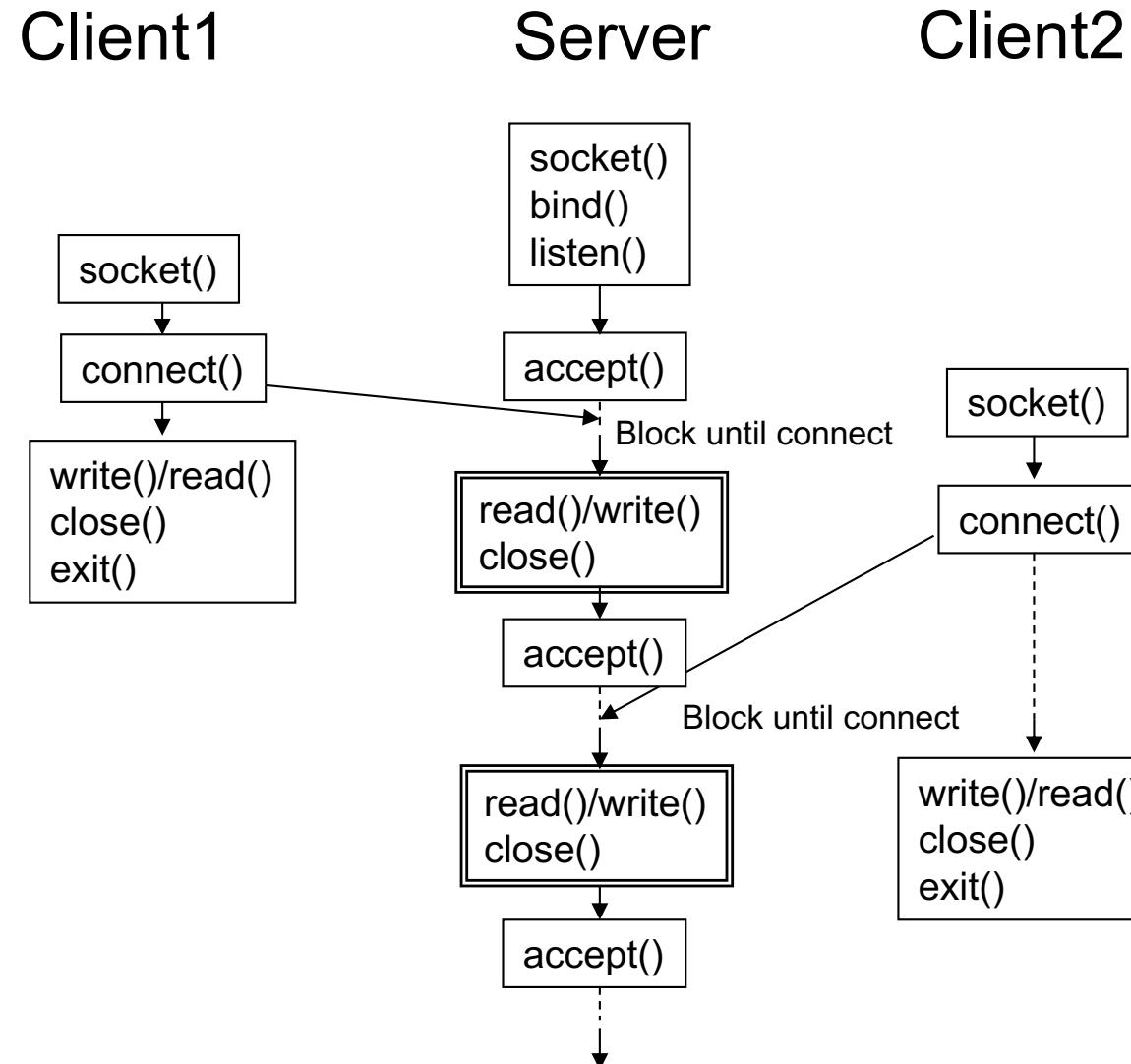
# サーバプログラミング

- 反復サーバ (interactive server)
  - 複数のクライアントからの要求を順次処理するサーバ
    - daytime などの簡素な応答サービスで利用される
    - 同時に複数のクライアントから要求がある場合、応答が待たされることがある
- 並行サーバ (concurrent server)
  - 複数のクライアントからの要求を並行処理するサーバ
    - echo など長時間の応答サービスで利用される
    - 同時に多数のクライアントから要求があっても、応答が待たされることはない

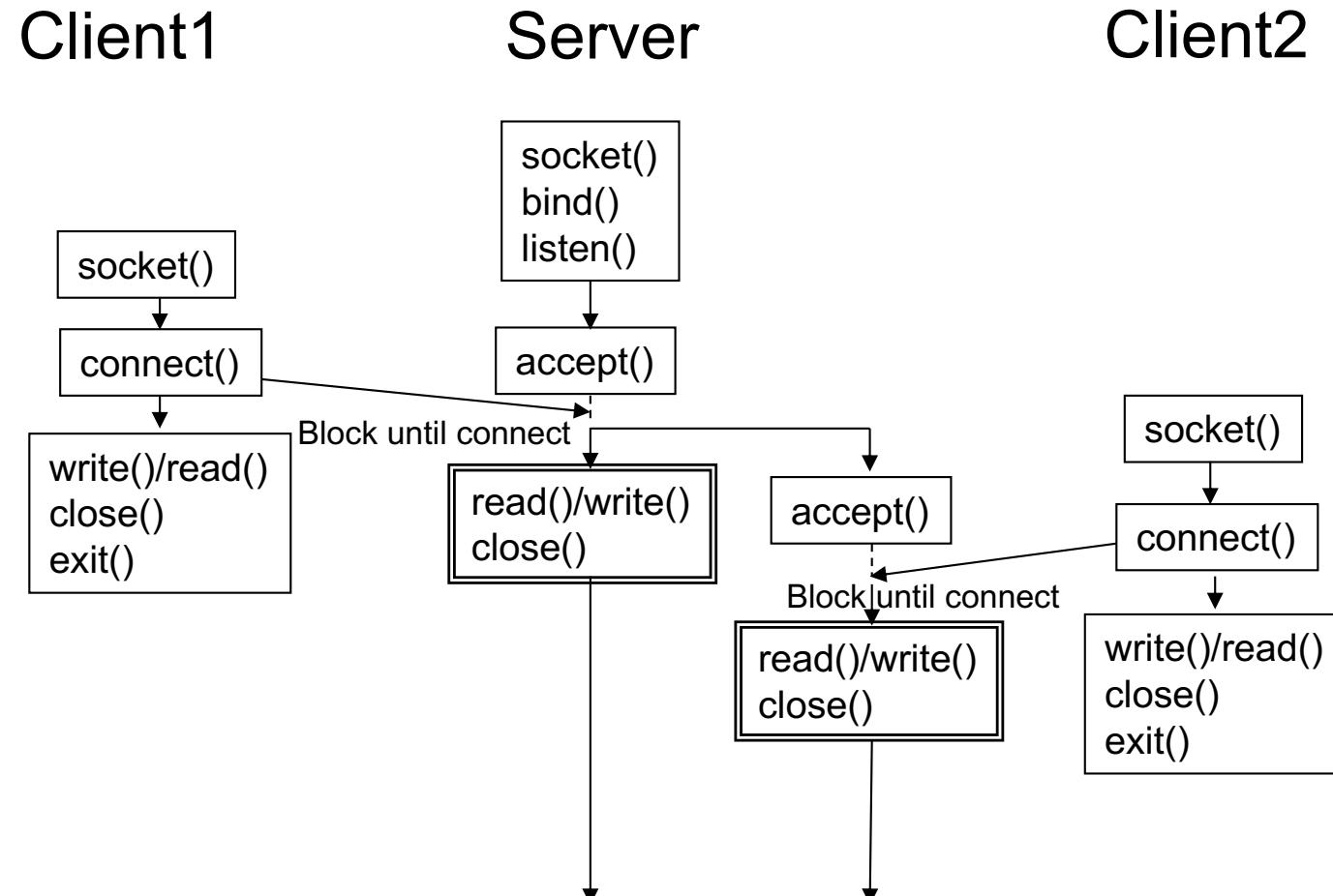
# daytime server / echo server



# Interactive Server (TCP)



# Concurrent Server (TCP)



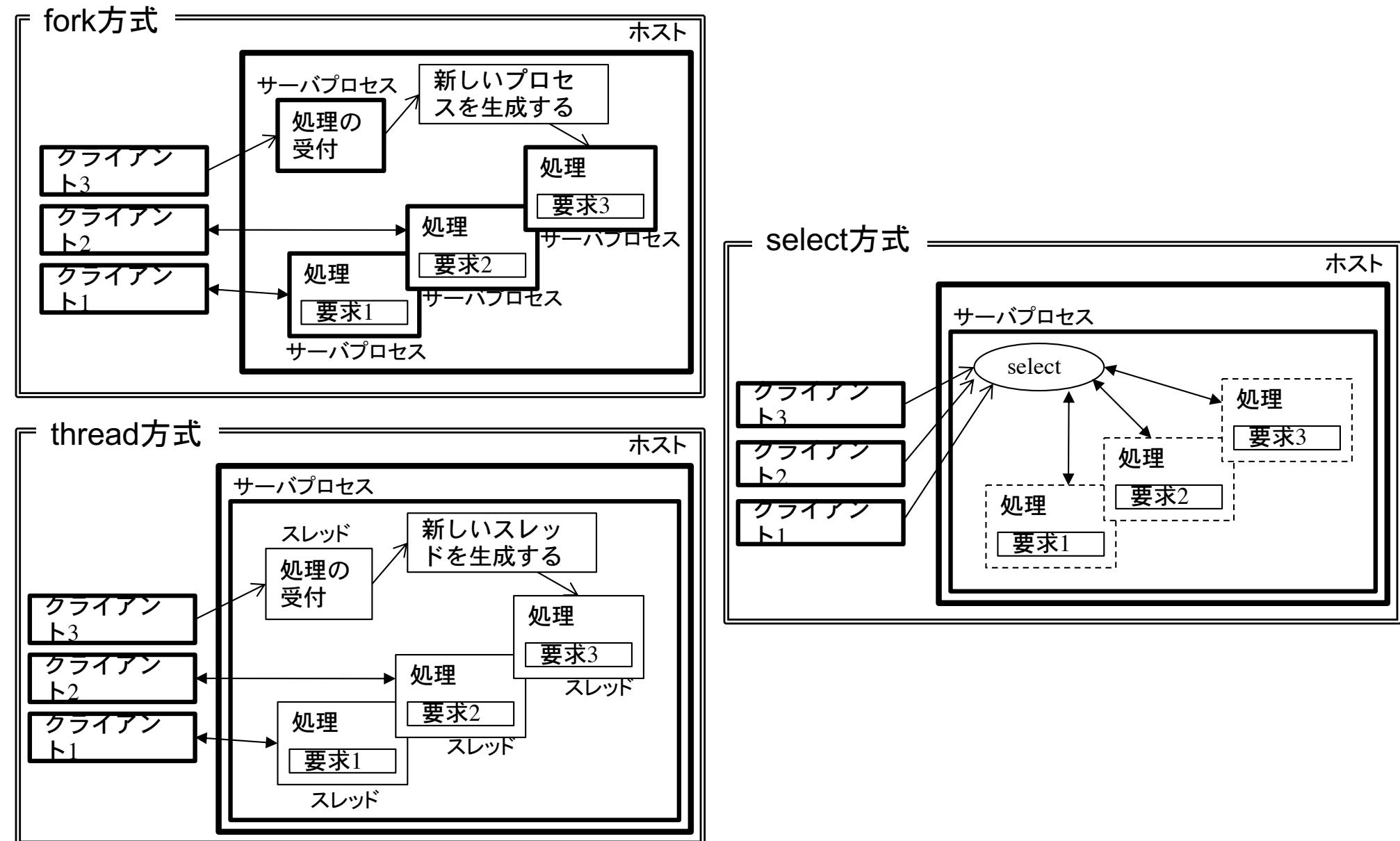
# TCP/IP Server programming

- サーバプログラムは、同時に複数のクライアントプログラムの処理をする必要がある場合がある
- サーバプログラムでは、`read()/write()` や `send()/recv()` 関数でクライアントからの要求を待つ必要があるが、これらの I/O 関数では、同時に複数のクライアントの処理を行うことができない。
- このため、並行サーバプログラムでは、以下の 3種類の方法を用いることになる。
  - `fork`, `thread`, `select`

# 各方式の利点・欠点

| 方式     | 利点                      | 欠点                                                                       |
|--------|-------------------------|--------------------------------------------------------------------------|
| fork   | 変数空間が独立<br>プロセスが独立      | メモリ利用効率<br>利用できるOSに制限                                                    |
| thread | メモリ有効利用<br>ほとんどのOSで利用可能 | デバッグが難しい                                                                 |
| select | 資源の有効利用<br>プロセス内部処理     | 処理が複雑になりがち<br><br>ソケットにパケットが届いて<br>いるかを逐一確認しなけれ<br>ばならないため、処理が重<br>くなりがち |

# 並行サーバの構成方法



# 並行サーバの作り方 (fork編)

- サンプルコード

```

pid_t pid;
Int listenfd, connfd;

listenfd = socket( ... );
bind(listenfd, ... );
listen(listenfd, LISTENQ);

for ( ; ; ) {
    connfd = accept(listenfd, .... );

    if ( (pid = fork() ) == 0 ) {
        close(listenfd); doit (connfd);
        close(connfd); exit(0);
    }
    close(connfd);
}

```

- fork()について

```
#include <unistd.h>

pid_t fork(void)
```

戻り値: 子プロセスなら 0  
 親プロセスなら子プロセスの PID  
 エラーなら -1

- プロセスの確認方法

```
% ps -l
```

# 並行サーバの作り方 (thread編)

- サンプルコード

```
static void *doit(void *connfd)
{
    pthread_detach(pthread_self());
    do_task((int) connfd);
    close((int) connfd);
    return(NULL);
}

int main()
{
    int l connfd;
    pthread_t tid;

    socket(...); bind(...); listen(...);

    for (;;) {
        connfd = accpet(...);
        pthread_create(&tid, NULL, &doit, (void)connfd);
    }
}
```

- pthread\_create()について

```
#include <pthread.h>
```

```
int pthread_create(pthread_t *restrict thread,
                  const pthread_attr_t *restrict att,
                  void *(*start_routine)(void *),
                  void *restrict arg)
```

戻り値: thread が生成できたら 0

thread が生成できない時はエラー値

# 並行サーバの作り方 (select編)

- サンプルコード

```

for ( i=0 ; i < FD_SETSIZE ; i++ )
    client[i] = -1;
FD_ZERO(&allset);
FD_SET(listenfd);

for (;;) {
    rset = allset;
    nready = select(maxfd+1, &rset, NULL, NULL, NULL);
    if (FD_ISSET(listenfd, &rset)) {
        connfd = accept(...);
        client[x] = connfd;
        FD_SET(connfd, &allset);
    }
    for ( i=0 ; i <= maxi ; i++ ) {
        if (FD_ISSET(client[i], &rset) {
            n = read(client[i], buff, sizeof(buff));
        } else {
            write(client[i], buff, n);
        }
    }
}

```

- select()について

```

#include <sys/select.h>

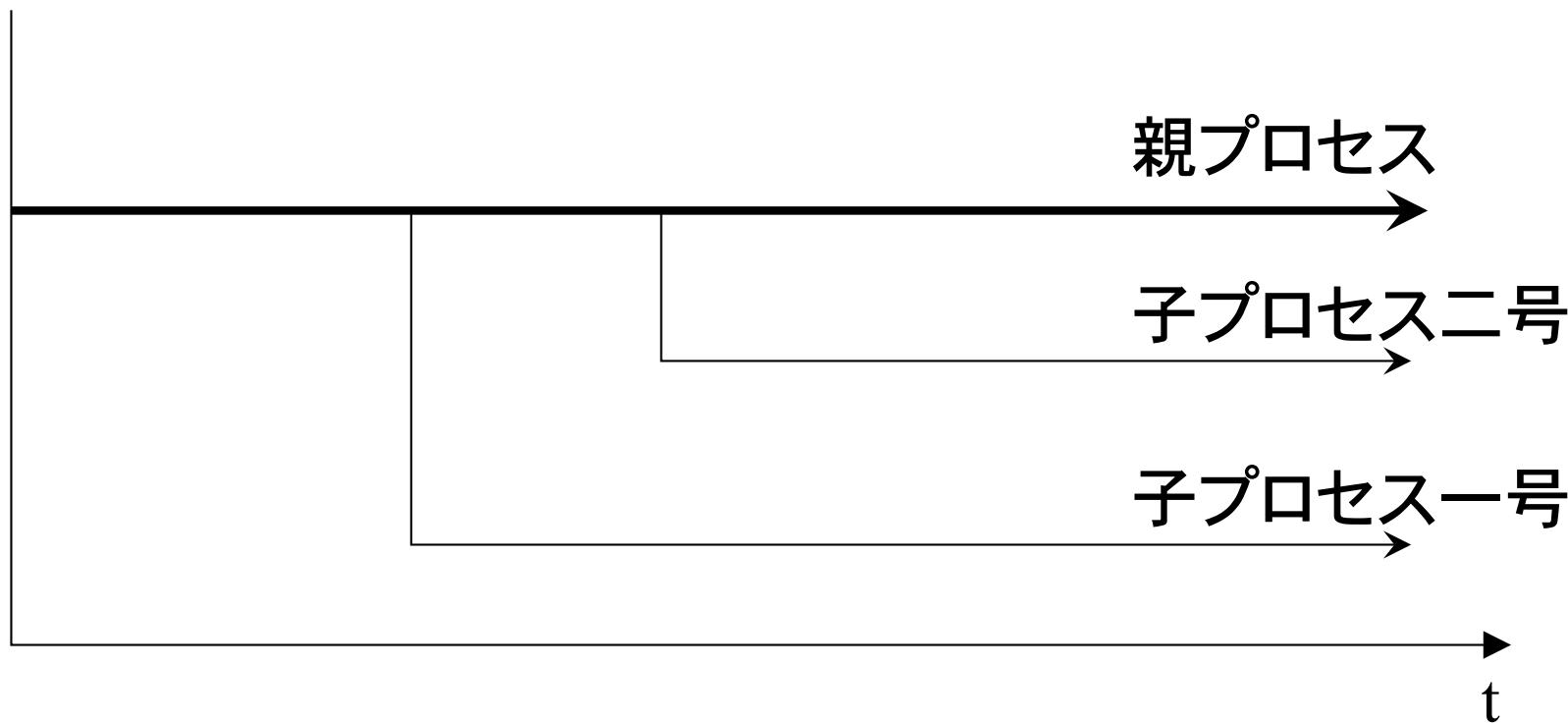
int select(int nfds,
           fd_set *restrict readfds,
           fd_set *restrict writefds,
           fd_set *restrict errorfds,
           struct timeval *restrict timeout);

```

戻り値: ready descriptor の総数  
エラーの時は、-1

# fork – 自分の分身を作る

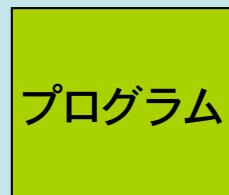
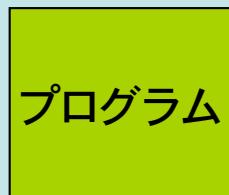
- システムコールで自分の分身を作成する
- 親の仕事は、コネクションがきたときに分身を作ることだけ



# プロセスとは？

- アプリケーションの実行
  - プログラムをプロセスとして実行
- プロセスはプログラム(=仕事)を実行している担当者のようなもの
  - 同じプログラムが複数プロセスで動くこともある

OS内



## プロセス ID

**nakayama@login1:~\$ ps**

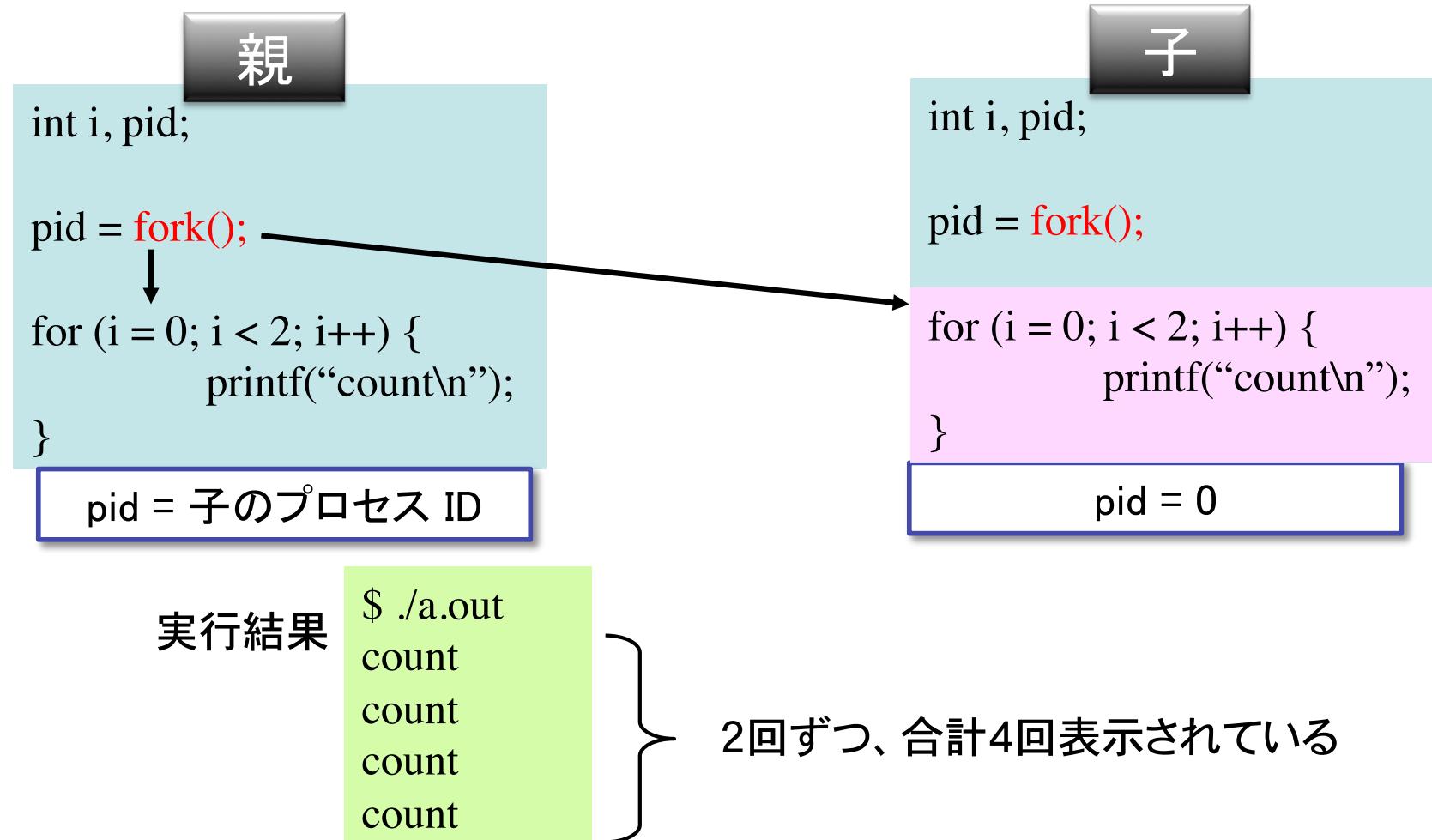
| PID  | TTY   | TIME     | CMD  |
|------|-------|----------|------|
| 4044 | pts/0 | 00:00:00 | bash |
| 4049 | pts/0 | 00:00:00 | ps   |

**nakayama@login1:~\$ pstree -pu nakayama**

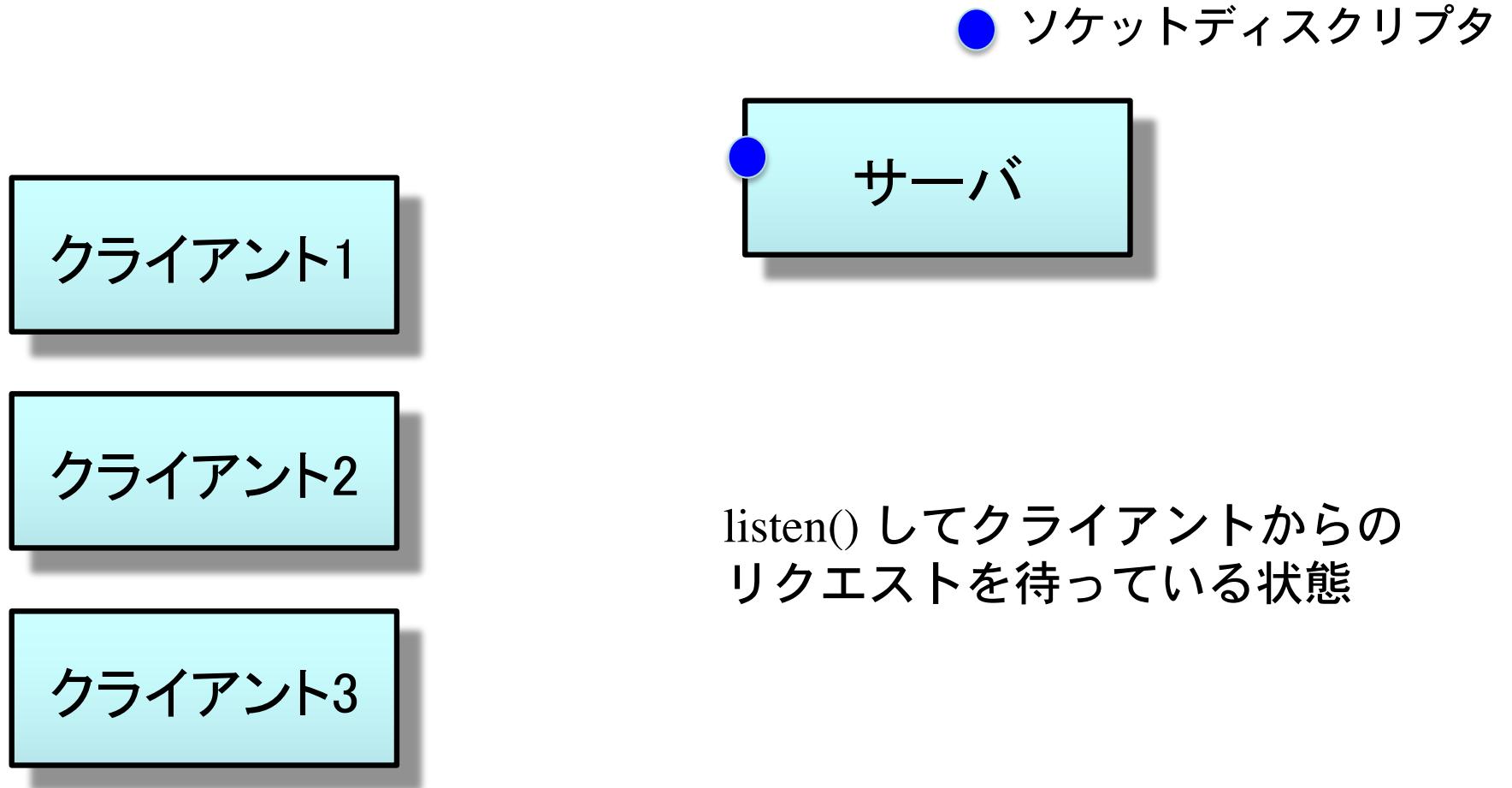
sshd(4043)---bash(4044)---pstree(4050)

# fork()

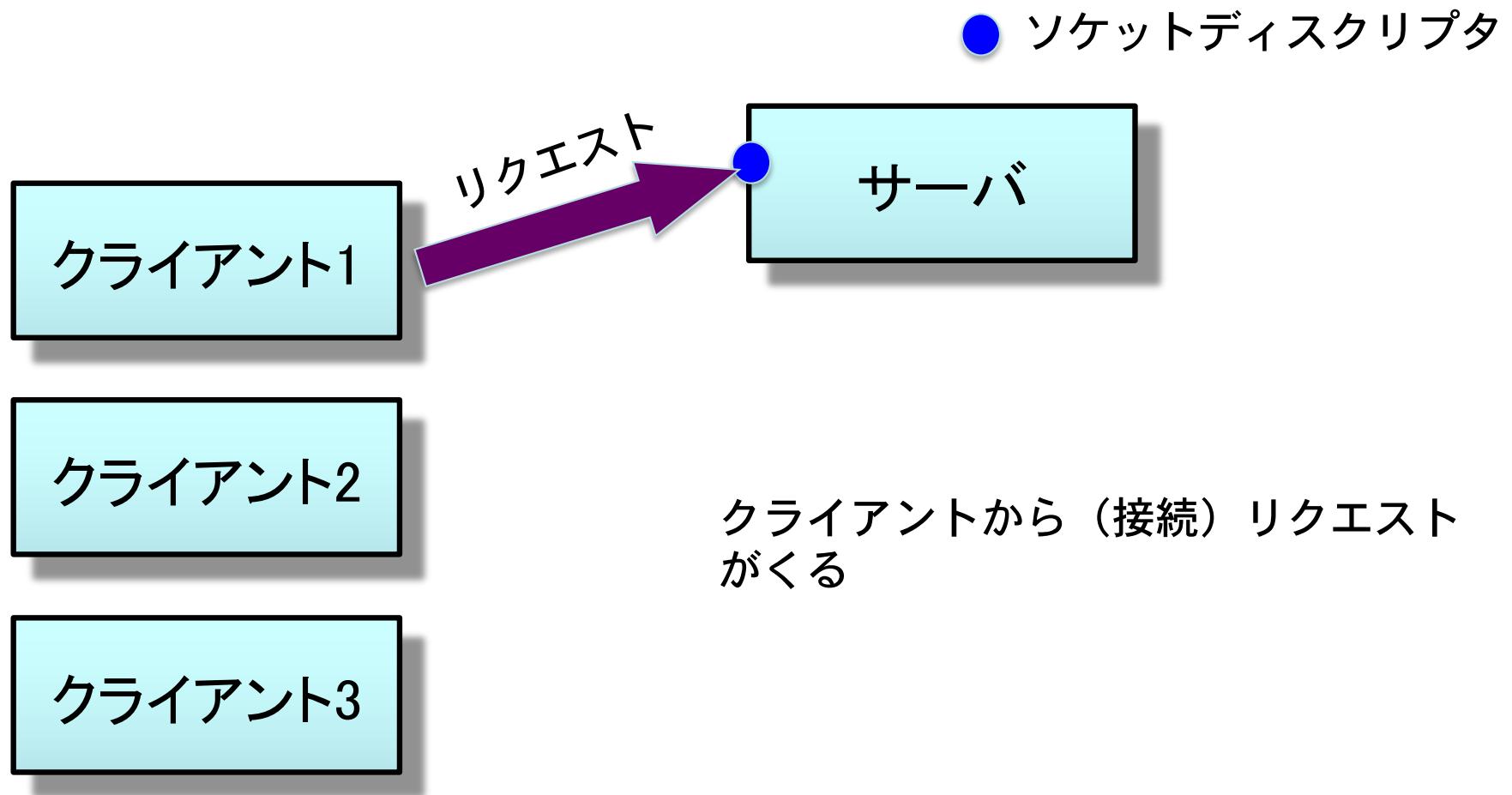
- 自分とまったく同じ複製プロセスを作るシステムコール



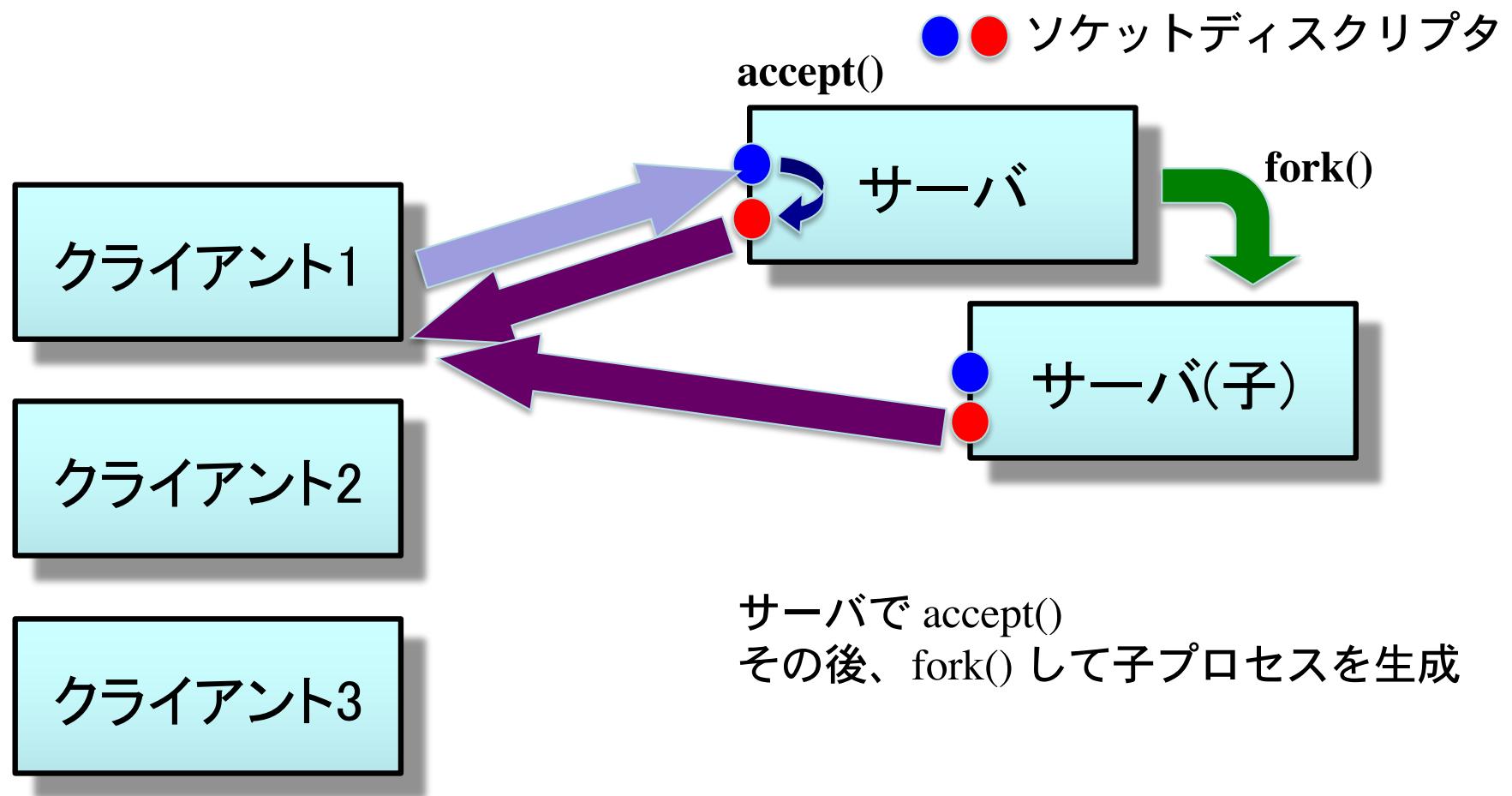
# fork() を用いたサーバの仕組み



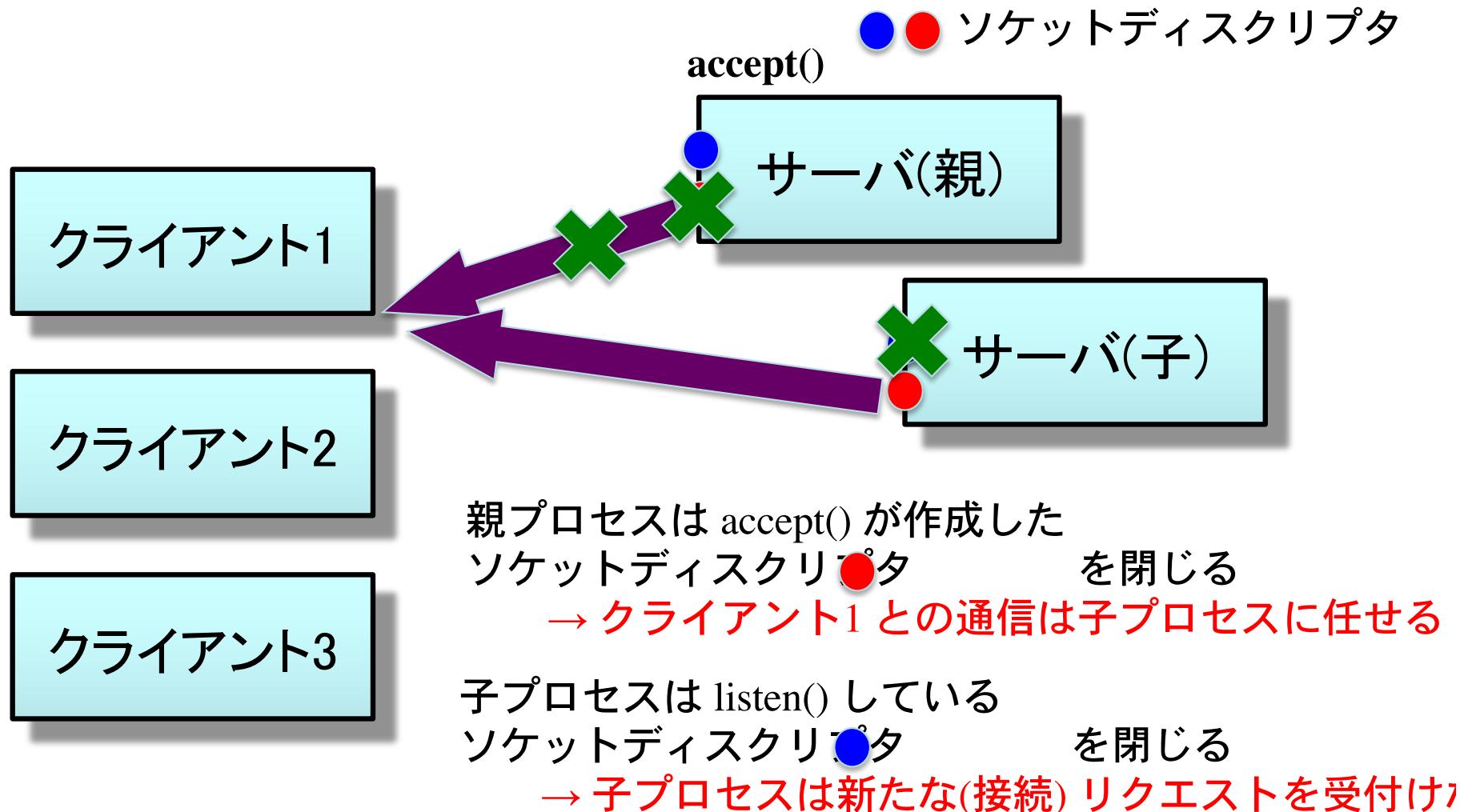
# fork() を用いたサーバの仕組み



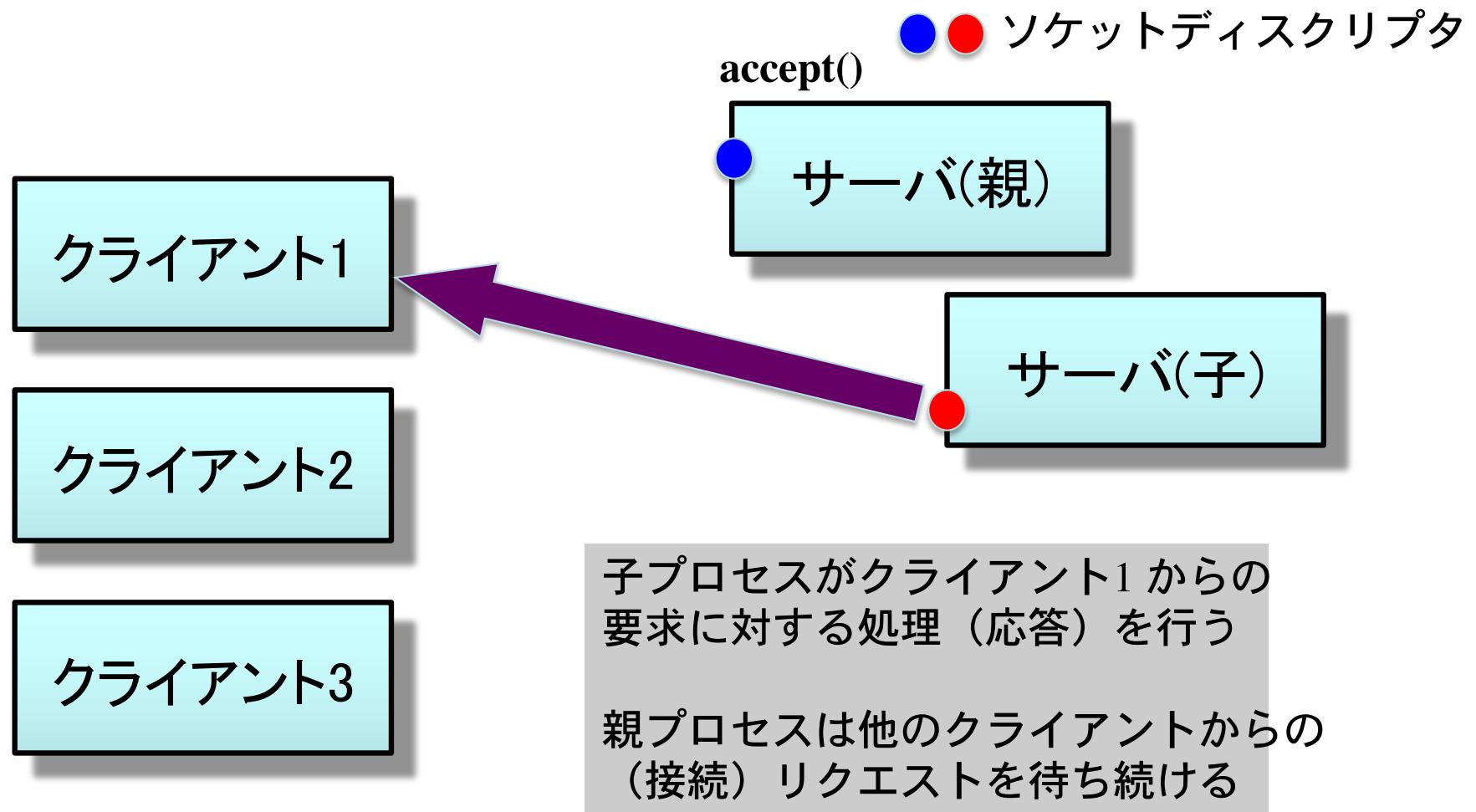
# fork() を用いたサーバの仕組み



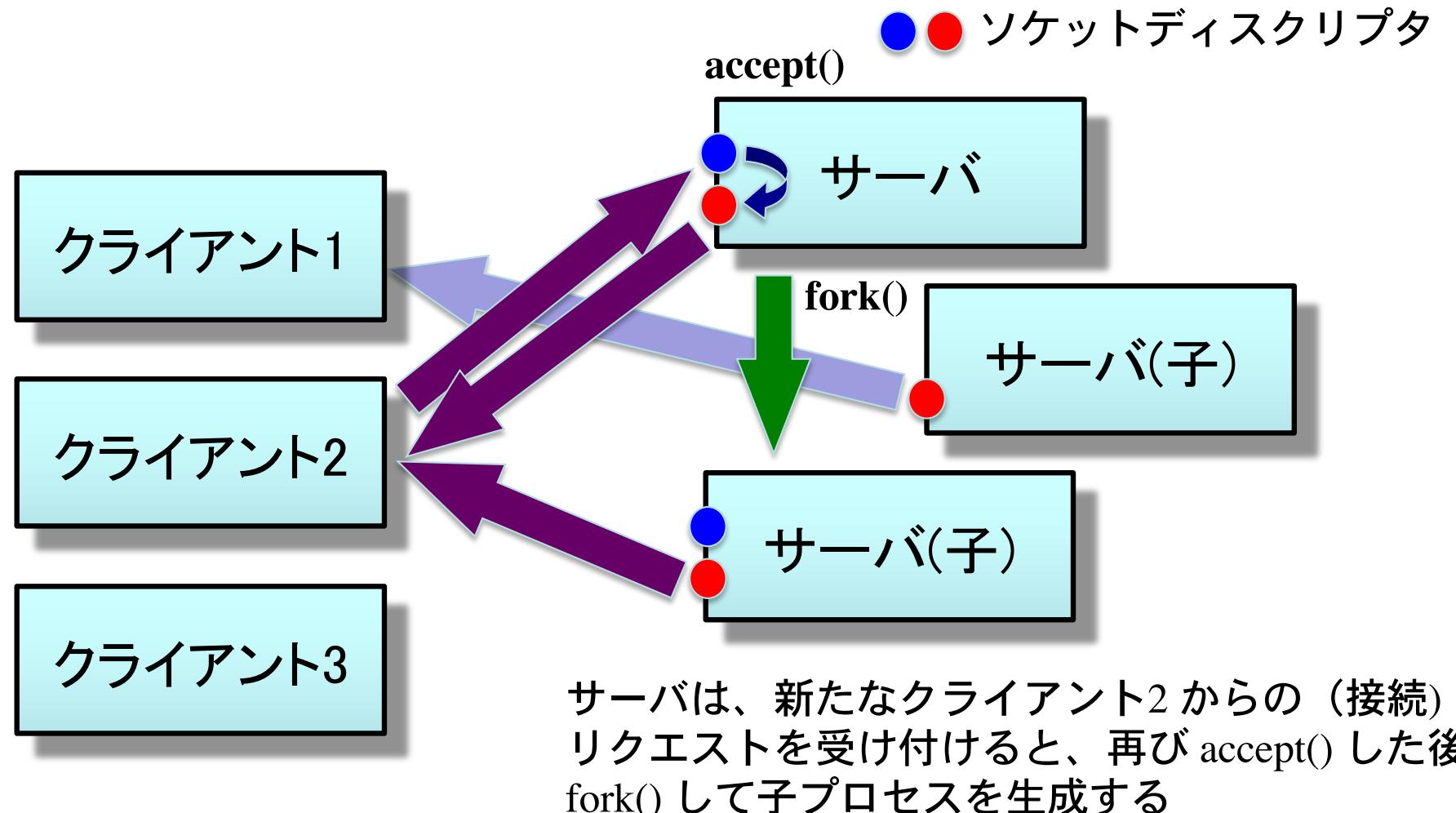
# fork() を用いたサーバの仕組み



# fork() を用いたサーバの仕組み



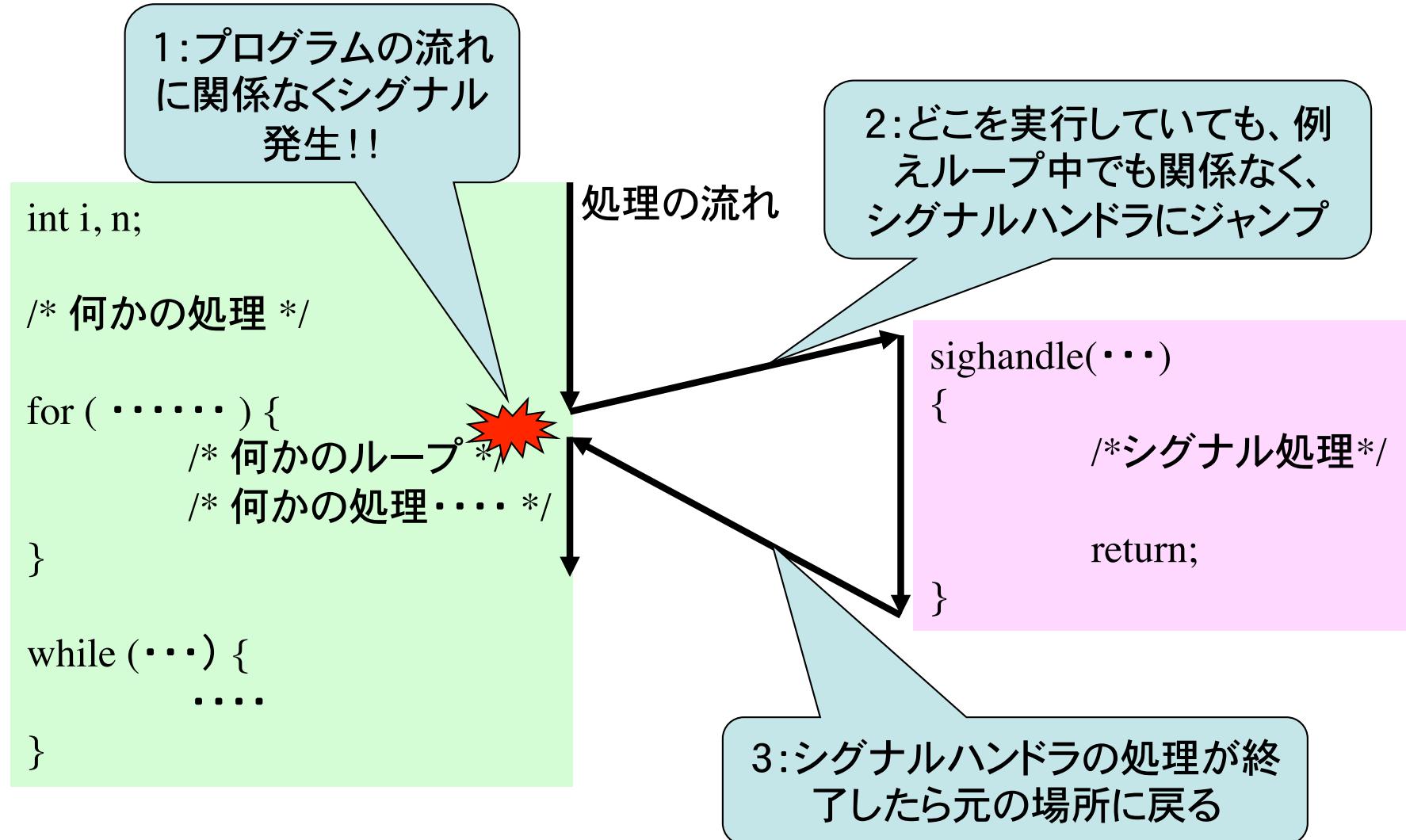
# fork() を用いたサーバの仕組み



## fork() と終了

- 子プロセスは終了すると親に終了状態を通知
- 親が通知されたメッセージを処理しないと実際には終了されずに残る（ゾンビプロセス）
- 通知はシグナルという OS の機能を使って行われる
- 親プロセスが子プロセスの後始末をする必要がある

# シグナルハンドラ



# シグナルハンドラの登録

- `signal()` 関数
  - シグナルハンドラ（関数）の関数ポインタを渡す
  - シグナルハンドラの形式は固定
    - `int` 型の引数を持つ関数
    - 引数にはシグナルの種類が入る

```
typedef void (*sighandler_t) (int);
```

```
sighandler_t signal ( int signum, sighandler_t sighandler);
```

## wait()

- 子プロセスの状態（ステータス）を取得するためのシステムコール
  - 戻り値：子プロセスの pid
  - 失敗：-1
- 子プロセスを正常に終了させるために必要

```
pid_t wait( int *status);
```

## waitpid()

- wait() は、子プロセスが終了する（状態が変わる）までブロックする
  - 親プロセスは、（その間）別の処理を行うことができない
- waitpid()
  - ブロックしないようにするオプションがある
    - WNOHANG
    - 戻り値 0 は、もう状態の変化したプロセスがないことを表す
  - pid を指定できる
    - -1 を指定すると、最初に状態が変化した子プロセス
    - 0 を指定すると、同一プロセスグループ id を持つ子プロセス

```
pid_t waitpid ( pid_t wpid,           // wait する対象のプロセス
                 int *status,          // status
                 int option);         // オプション
```

# 主なシグナルの種類

- SIGINT
  - Ctrl-C を押した時に発生、プロセスを終了するために使われる
- SIGTERM
  - プロセスを終了するために使われる。kill コマンドが default で送るシグナル
- SIGSTP
  - Ctrl-Z を押した時に発生、プロセスをサスペンドするために使われる
- SIGCHLD
  - fork() で生成した子プロセスが終了したことを通知するシグナル。これを受けて wait() などの後処理を実行する
- SIGARM
  - alarm() によって指定された秒数後に送られるシグナル。タイマーの実装でよく使われる

# 代表的な使い方

```
void sig_child(int signo)
{
    int pid, status;
    pid = wait(&status);
    printf("PID: %d, terminated\n", pid);
}

int main(int argc, char *argv[])
{
    ...
    signal(SIGCHLD, sig_child);
    ...
}
```

シグナルハンドラを定義。  
wait() でプロセスの終了  
状態を取得する



SIGCHLD シグナルが来た  
ときに sig\_child() 関数を呼び  
出すように登録する

# TCP サーバ サンプルコード(1)

```
int main(int argc, char **argv)
{
    int listen_sock, accept_sock, pid;
    socklen_t sin_siz;
    struct sockaddr_in serv, clt;
    unsigned short port;

    signal (SIGCHLD, signal_handler);

    if ((listen_sock = socket(PF_INET, SOCK_STREAM, 0)) < 0)
    {
        perror("socket");
        exit(1);
    }
```

# TCP サーバ サンプルコード(2)

```
while(1) {
    accept_sock = accept(listen_sock, (struct sockaddr *)&clt,
&sin_siz);
    pid = fork();
    if (pid < 0) {
        fork に失敗したのでエラー処理
    } else if (pid == 0) {
        close(listen_sock);
        実際に子プロセスにやらせたい処理
        exit(0);
    }
    close(accept_sock);
}
```

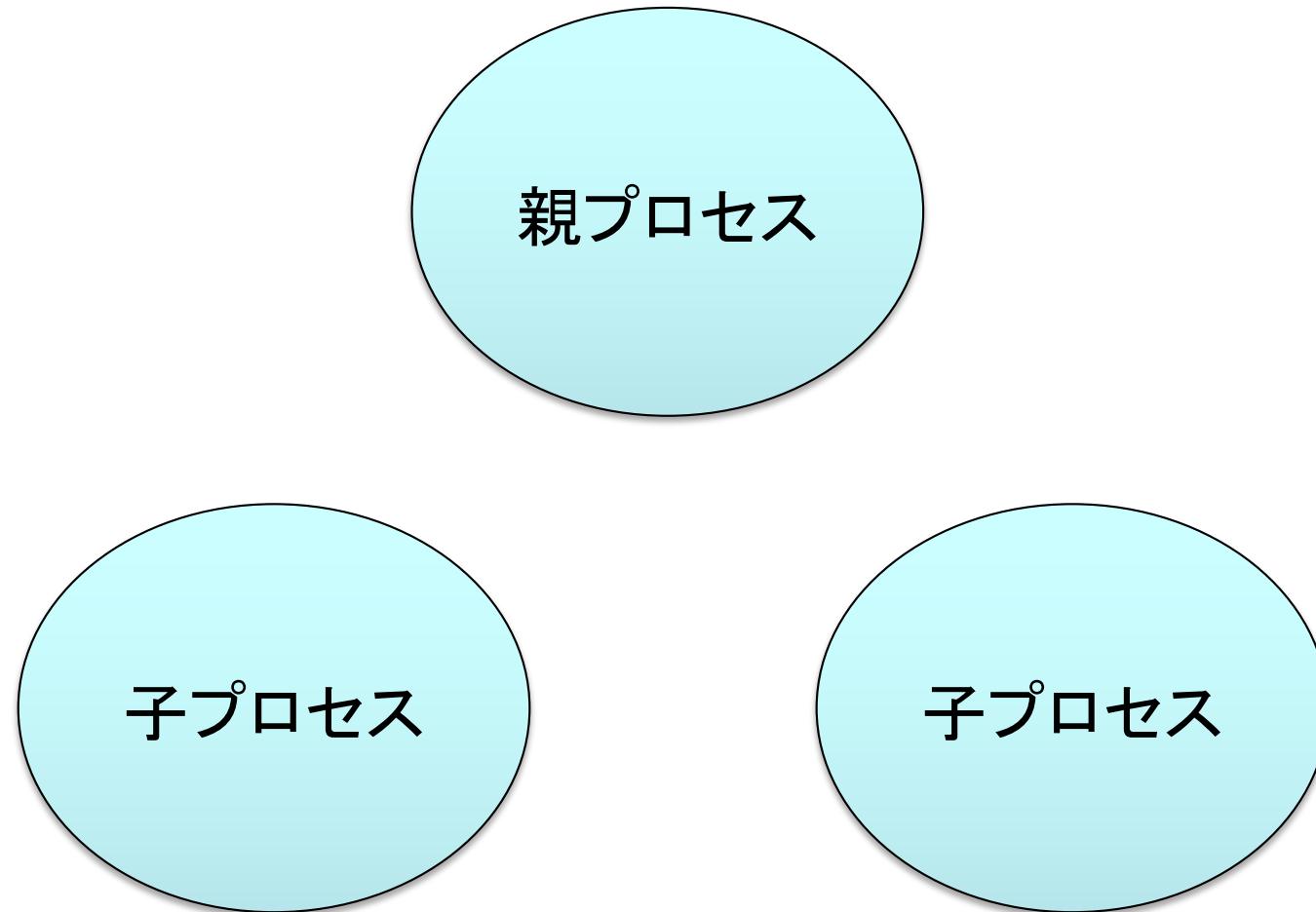
# TCP サーバ サンプルコード(3)

```
void signal_handler (int sig) {  
    int status, retval;  
  
    do {  
        retval = waitpid(-1, &status, WNOHANG);  
    } while (retval > 0);  
  
    signal(SIGCHLD, sigchld_handler);  
}
```

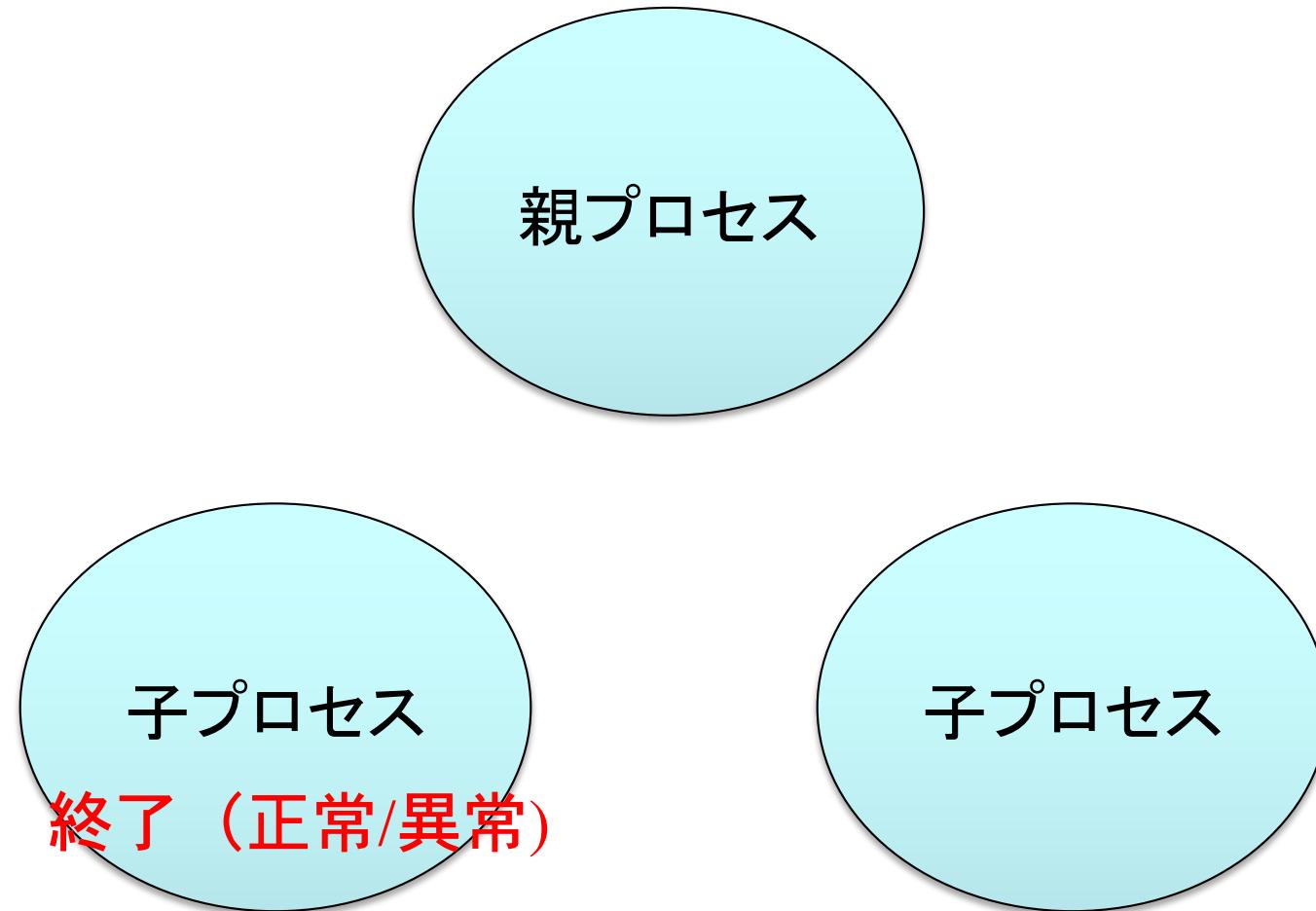
# fork() を利用した並行サーバの注意点

- 子プロセスが終了すると「どのような終了状況だったか」が親プロセスに伝えられる
  - システムが親プロセスに通知
  - シグナルを利用
- 親プロセスは、子プロセスの終了状況を見届けなければならぬ
- 子プロセスが終了しても、終了状態を受け取らないで親プロセスが(異常)終了すると、ゾンビプロセスがいつまでも残ることになる

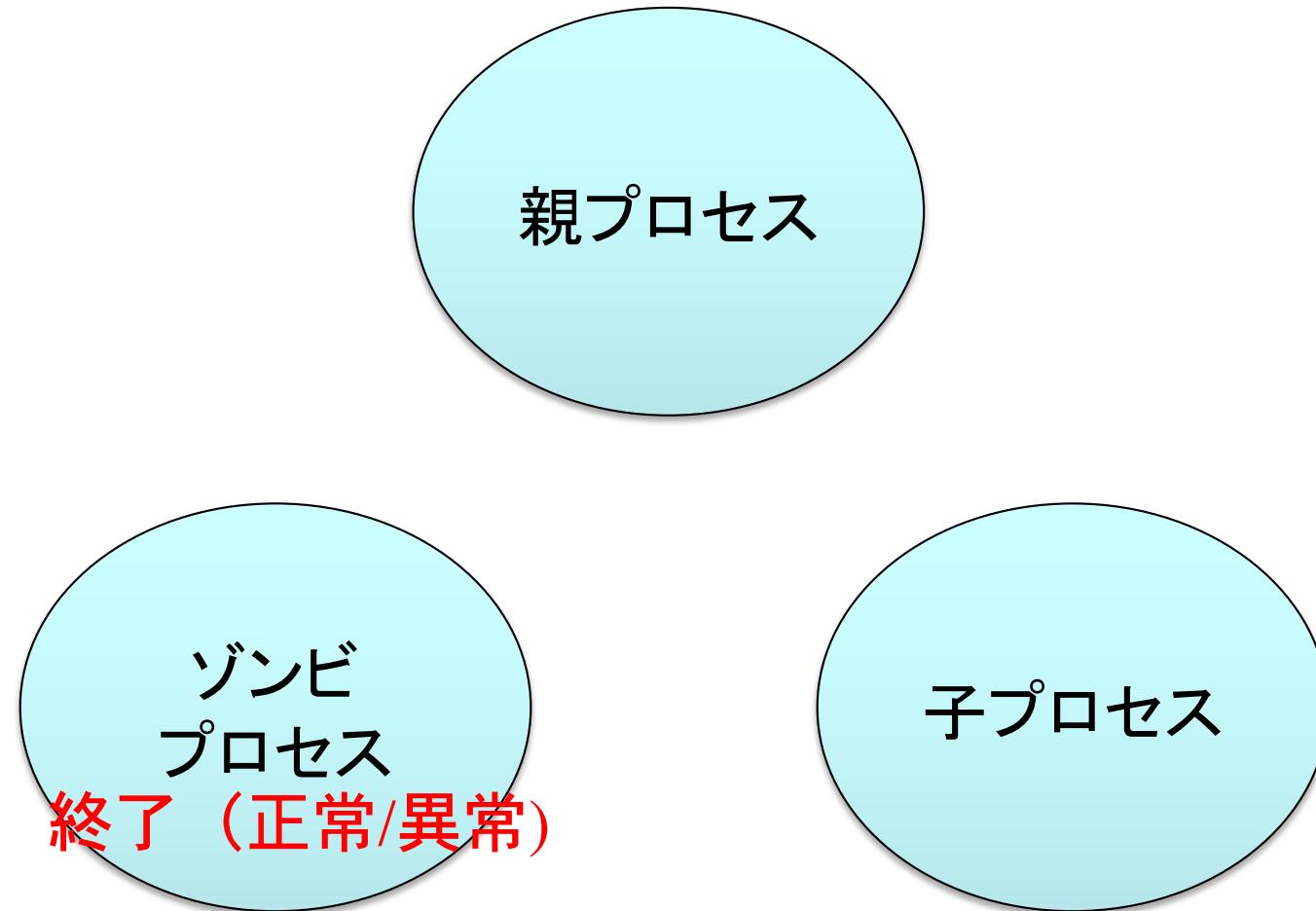
# 終了状態の受け渡し



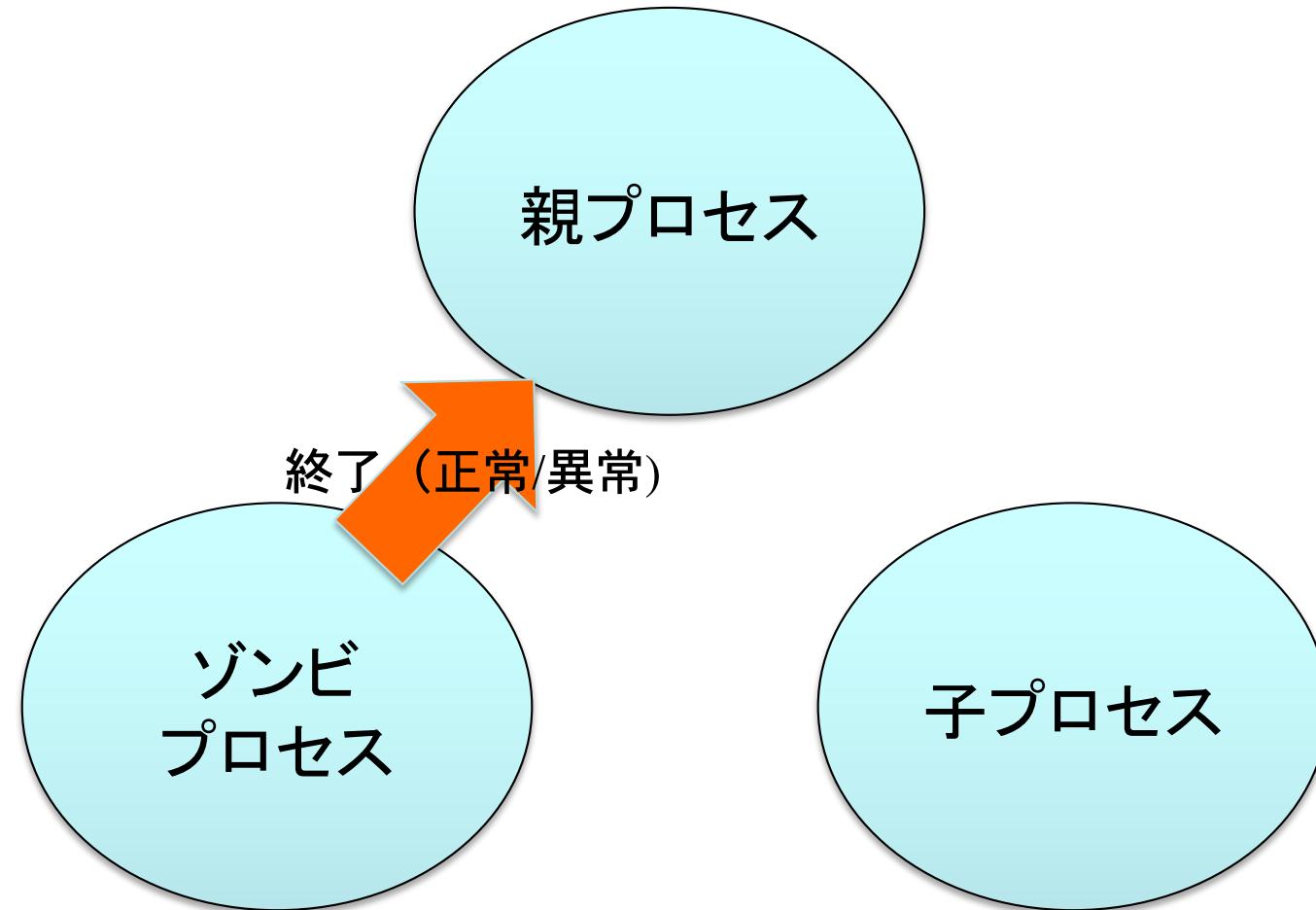
# 終了状態の受け渡し



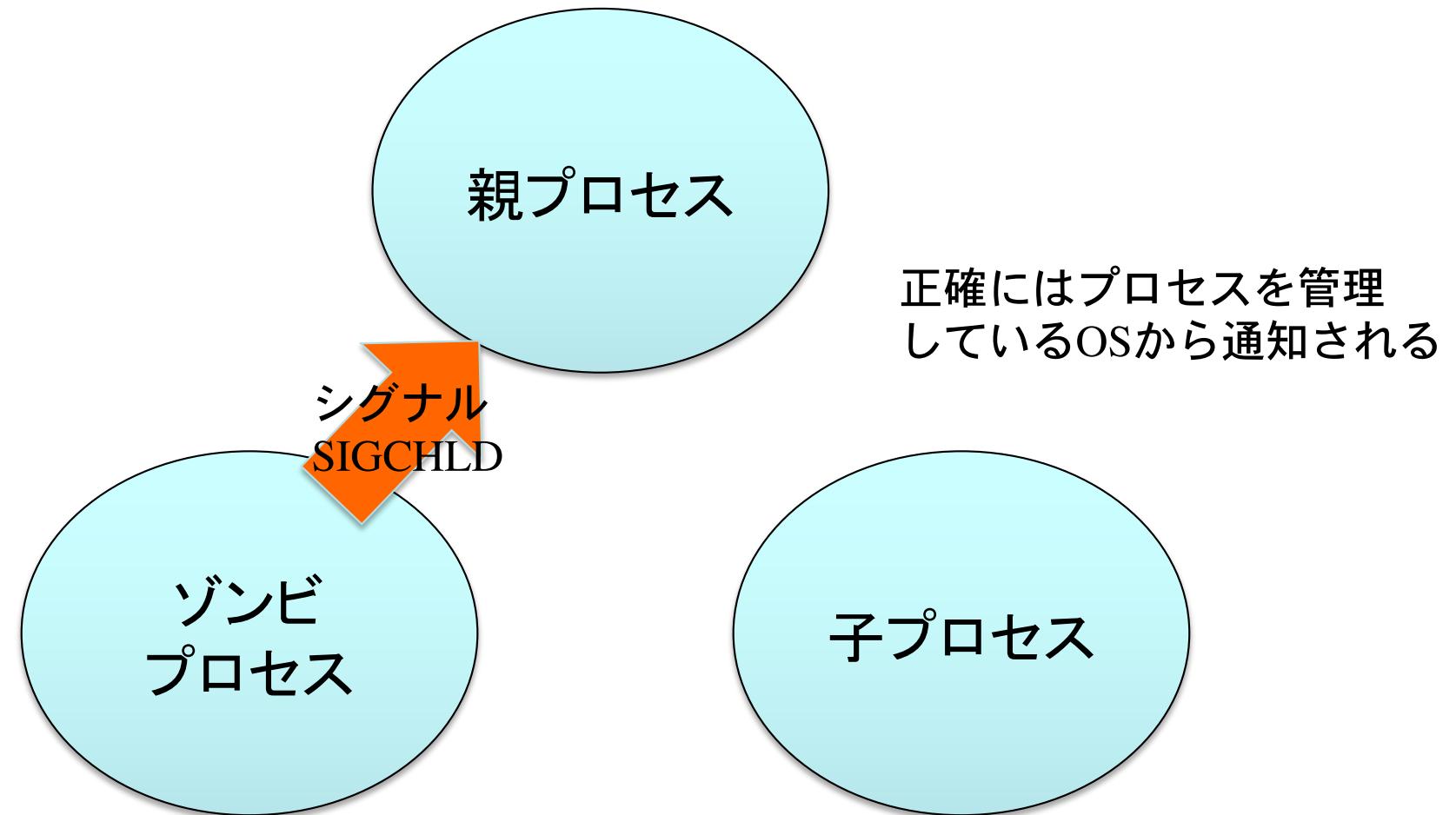
# 終了状態の受け渡し



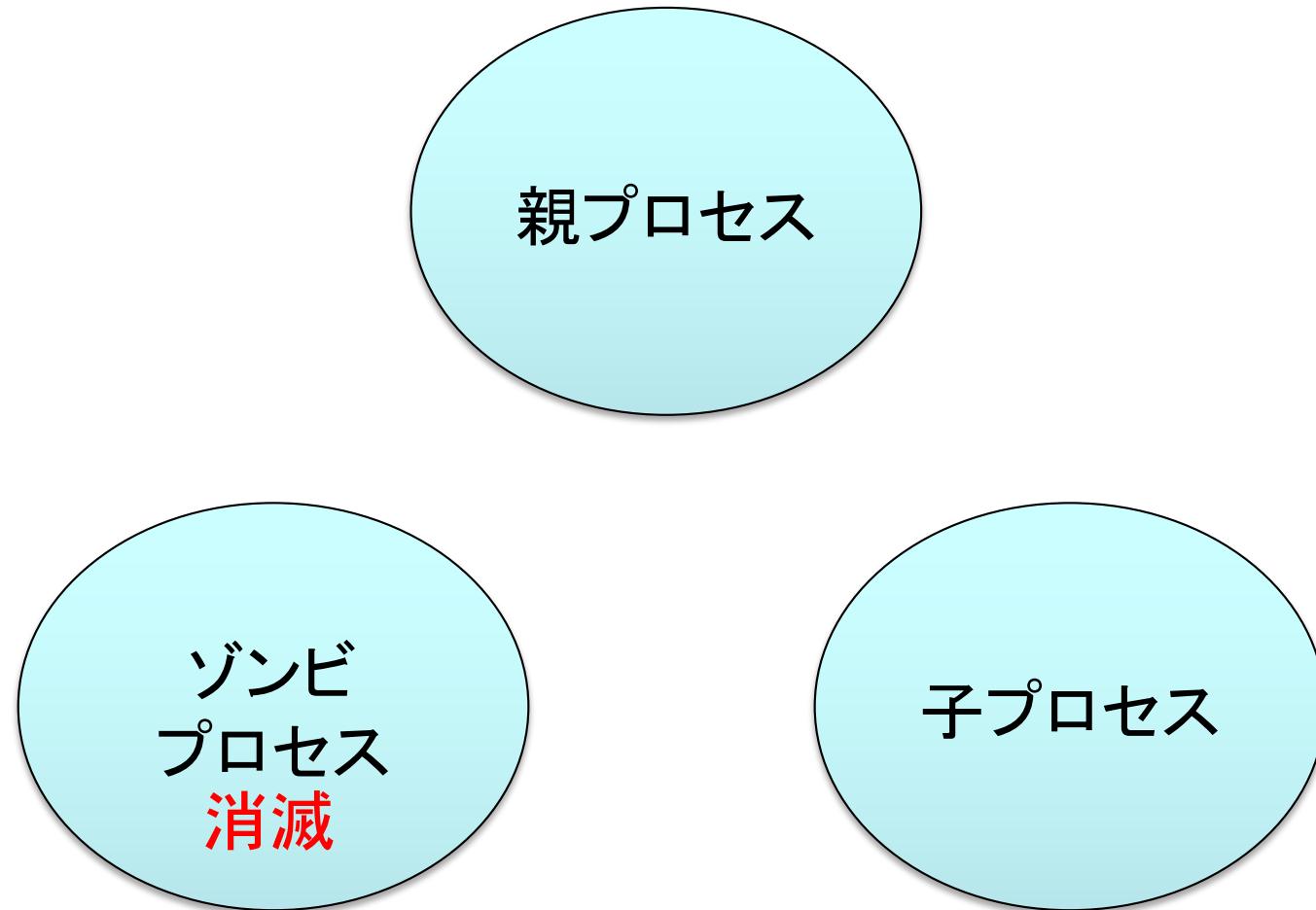
# 終了状態の受け渡し



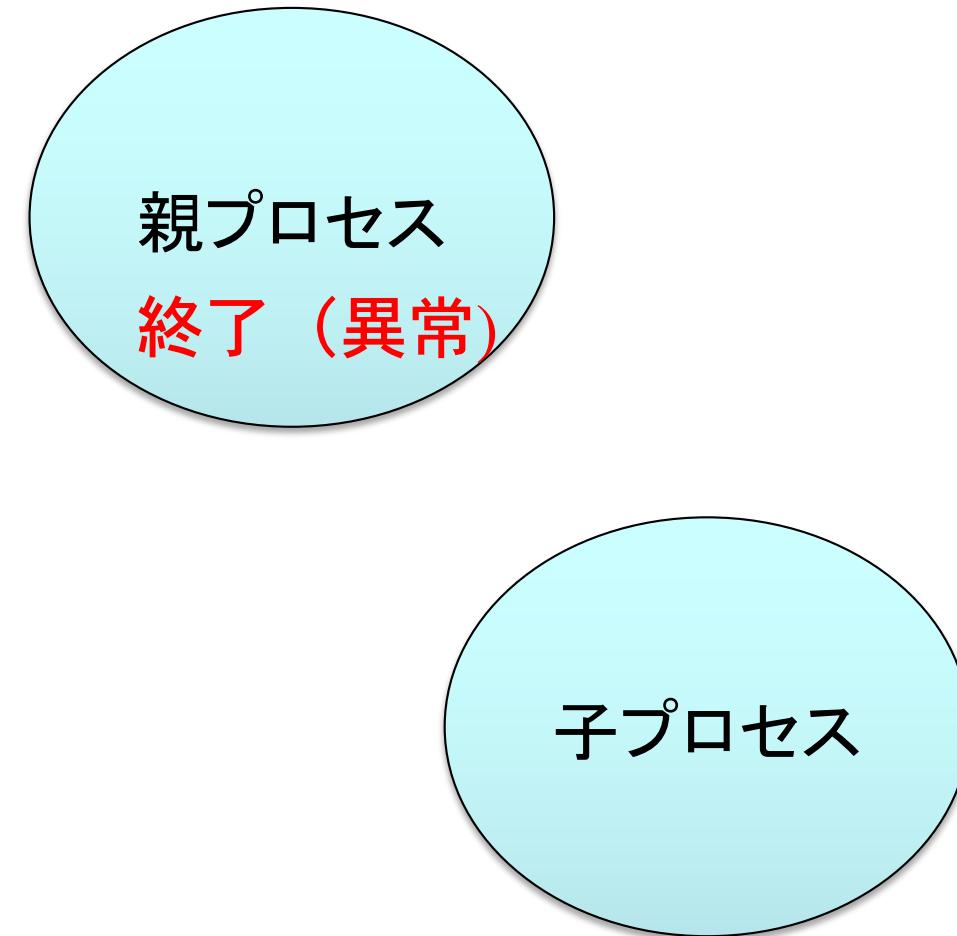
# 終了状態の受け渡し



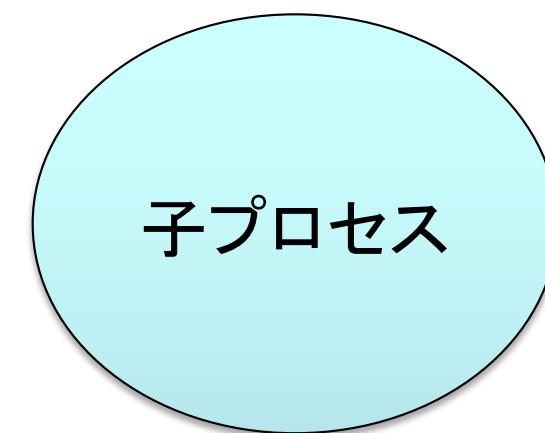
# 終了状態の受け渡し



# 終了状態の受け渡し



# 終了状態の受け渡し



# 終了状態の受け渡し

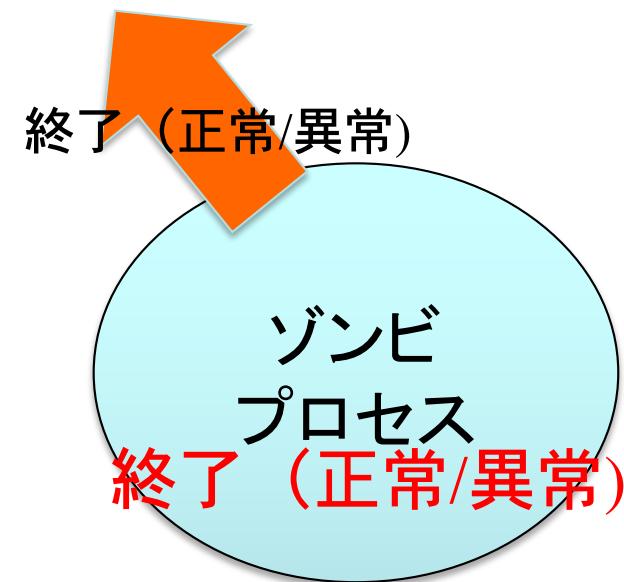


# 終了状態の受け渡し



# 終了状態の受け渡し

親プロセスが終了状態を  
受け取れない



# 終了状態の受け渡し

ゾンビプロセスが消滅できない

