ネットワークプログラミング第10回（応用ネットワークプログラミング）

２００９春学期　中村　修

授業Webページ

- SFC-SFS
  https://vu2.sfc.keio.ac.jp/sfc-sfs/

- 課題・授業資料などの情報を掲示します
  - **課題は毎回こちらに提出してください**！！
  - 今日の課題締め切り
    - 6/29(月)23:59まで！
    - 遅れて提出する方は要連絡

今期の授業スケジュール（予定）

- 第１回：イントロダクション　（4/14）
- 第２回：C言語の基礎～関数・ポインタ　（4/21）
- 第３回：C言語の基礎～コマンドライン引数　（4/28）
- 第４回：C言語の基礎～構造体・リスト構造　（5/12）
- 第５回：FileI/Oとシステムコール　（5/19）
- 第６回：ネットワークとプログラミング（1）　（5/26）
- 第７回：ネットワークとプログラミング（2）　（6/2）
- 第８回：ネットワークとプログラミング（3）　（6/9）
- 第9回：ネットワークとプログラミング（4）　（6/16）
- **第１０回：応用ネットワークプログラミング　（6/23）**
- 第１１回以降：ミニプロ　（6/30, 7/7, 7/14）

ミニプロジェクト(ミニプロ)について

- グループでプログラムを作る
- OSは問わない
  - 一応、公式サポート環境はccx
  - 他の環境でもTA/SAは（できるだけ）頑張ります
- ネットワークを使うプログラム
  - プログラムの難易度
  - **独創性！**

ミニプロの提出

- 提出は、ソースコードならびにレポート

- 提出ソースコードにはコメントを付加
  - 日本語 or 英語
- レポートには以下の項目を含める
  - 何を作ったのか
  - 使用方法
  - 面白さ
  - 実行環境
  - グループ内の、各自の担当個所

ミニプロのプレゼン

- 中間発表
  - 7/7(予定)
  - 何を作るかをプレゼンする
    - A41枚程度のレジュメを作成する

- 最終発表(7/14予定)
  - 授業最終回を予定
  - できあがりをプレゼンする
  - 評価

ミニプロのメンバー

- 申請したい人は6/19(金)までにTA/SAにメール(してもらいました)
  - 最大人数: 3人

- グループが決まり次第メールで連絡します！

ミニプロの例

- ネットワークゲーム
  - リバーシ
  - ○×ゲーム
  - 陣取りゲーム
- コミュニケーションツール
  - アドホックSkype

第9回課題
TCPエコーサーバの作成

- TCPを用いたechoサーバを作ろう。
  - 第一引数にポート番号を指定しよう
    %./a.out 49538
  - **ポート番号は、学籍番号下5桁利用**
- できたら、先に作成したTCPエコークライアントを用いて確認(または、telnet コマンドにても確認可能)

```
#include <stdio.h>
#include <netdb.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[]){
   int sock, accept_sock;
   int ret;
   char buf[256];
   struct addrinfo *ai, *res;
   struct addrinfo hints =
{AI_PASSIVE, AF_INET6, SOCK_STREAM
        , 0, 0, NULL, NULL, NULL};
   struct sockaddr_in6 peer;
   int addrlen_peer;
   ret = getaddrinfo(NULL, argv[1],
&hints, &res);
   if (ret != 0){
       fprintf(stderr, "getaddrinfo
(%s)¥n", gai_strerror(ret));
       exit(-1);
   }
```

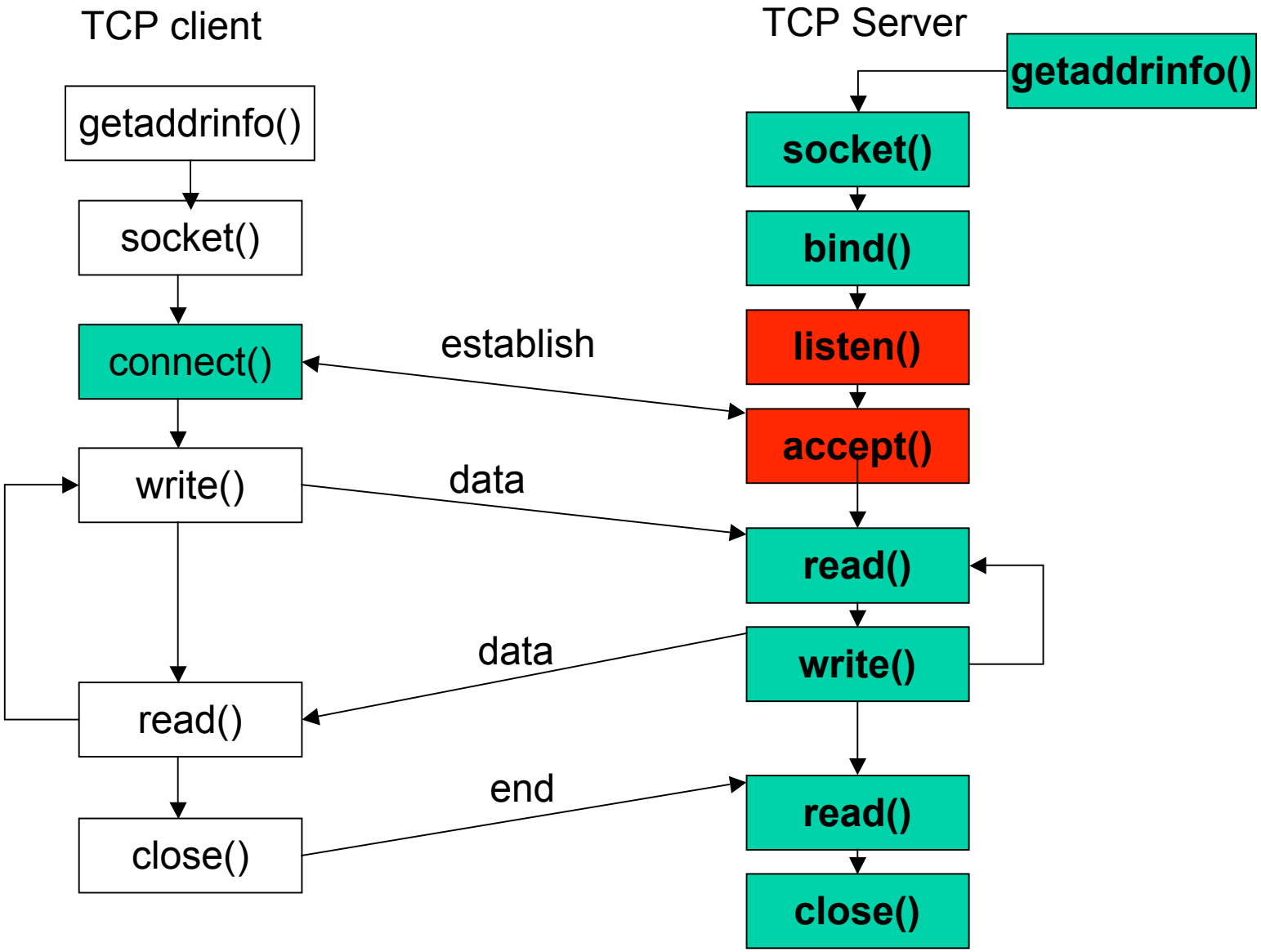
```
   for(ai = res; ai; ai = ai->ai_next){
       sock = socket(ai->ai_family, ai-
>ai_socktype, ai->ai_protocol);
       if (sock < 0){
               perror("socket");
               exit(-1);
       }
       ret = bind(sock, ai->ai_addr, ai-
>ai_addrlen);
       if (ret < 0){
               perror("bind");
               exit(-1);
       }
       ret = listen(sock, 5);
       if (ret < 0){
               perror("listen");
               exit(-1);
       }
       break;
   }
```

```
   memset (&peer, 0, sizeof (peer));
   addrlen_peer = sizeof (peer);
   accept_sock = accept (sock, (struct sockaddr *) &peer, &addrlen_peer);
    if (accept_sock < 0){
         perror("accept");
         exit(-1);
   }
    while (1){
         memset(buf, 0, sizeof(buf));
         ret = recv(accept_sock, buf, sizeof(buf), 0);
         if (ret < 0){
                 perror("recvfrom");
                 exit(-1);
         }
         printf("%s¥n", buf);
         ret = write(accept_sock, buf, ret);
         if (ret < 0){
                 perror("sendto");
                 exit(-1);
         }
   }
   freeaddrinfo(res);
   return 0;
}
```

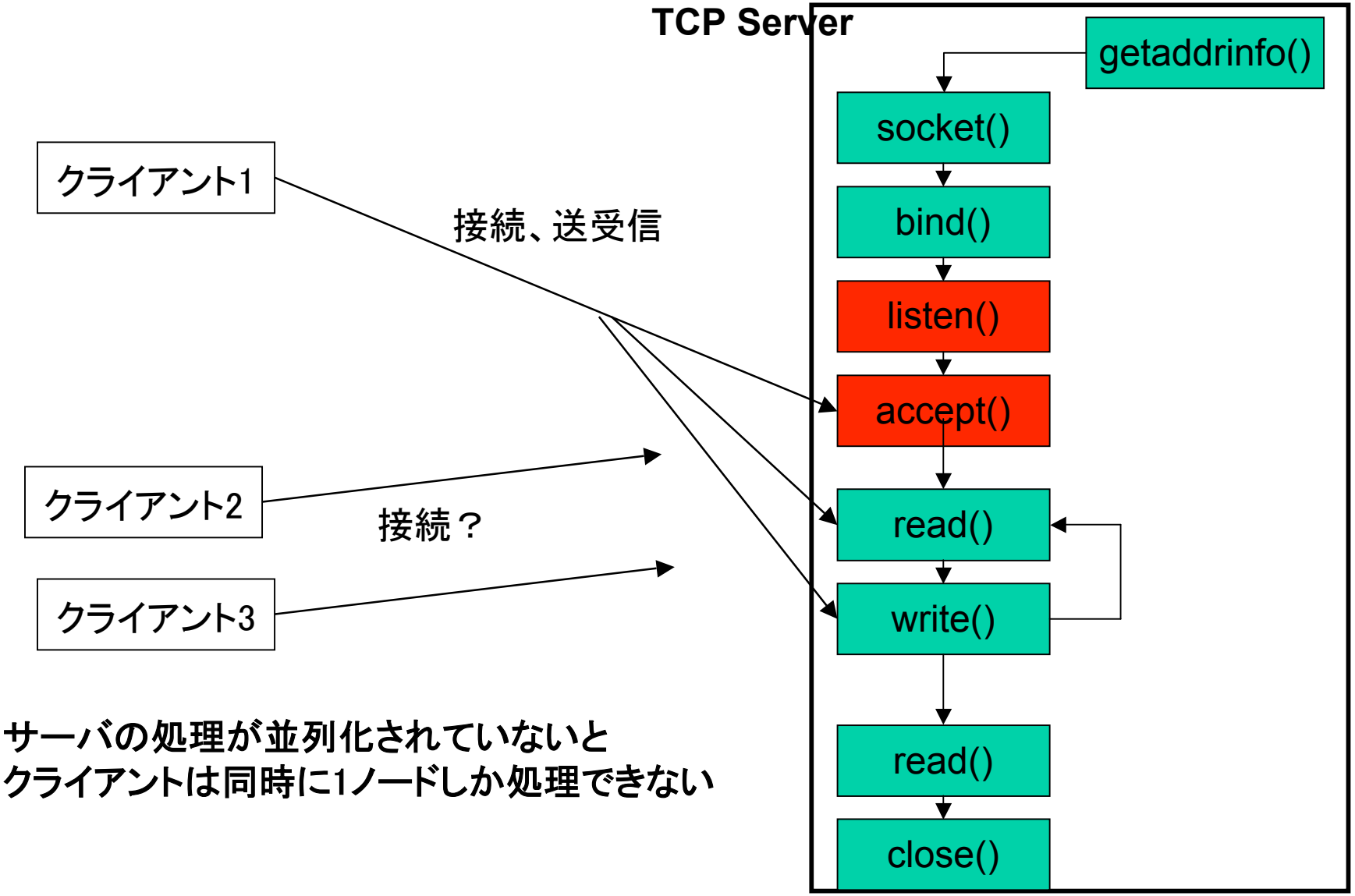
今日のお題

- 講義
  - 並列処理を用いた、TCPサーバ
  - fork()システムコールとは？
- 実習/課題：データの送受信
  - TCP echoサーバを改良し、複数クライアントの処理を行えるようにする

TCP 通信の流れ(server)



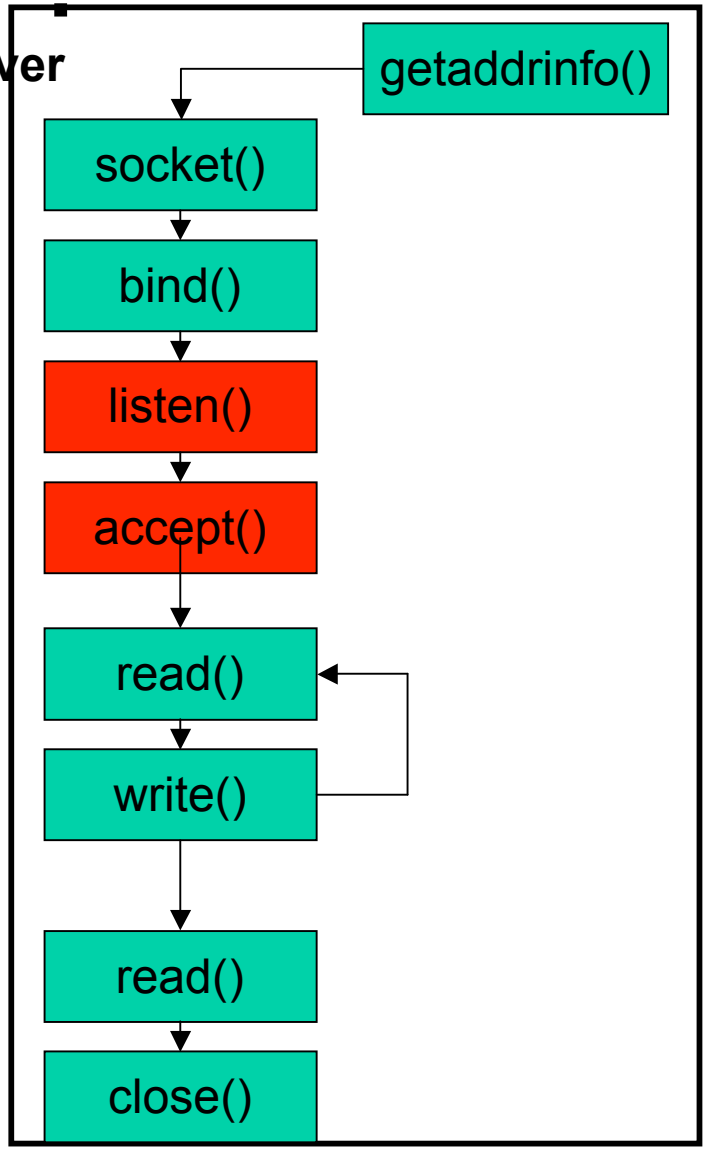
TCP 通信の流れ(server)



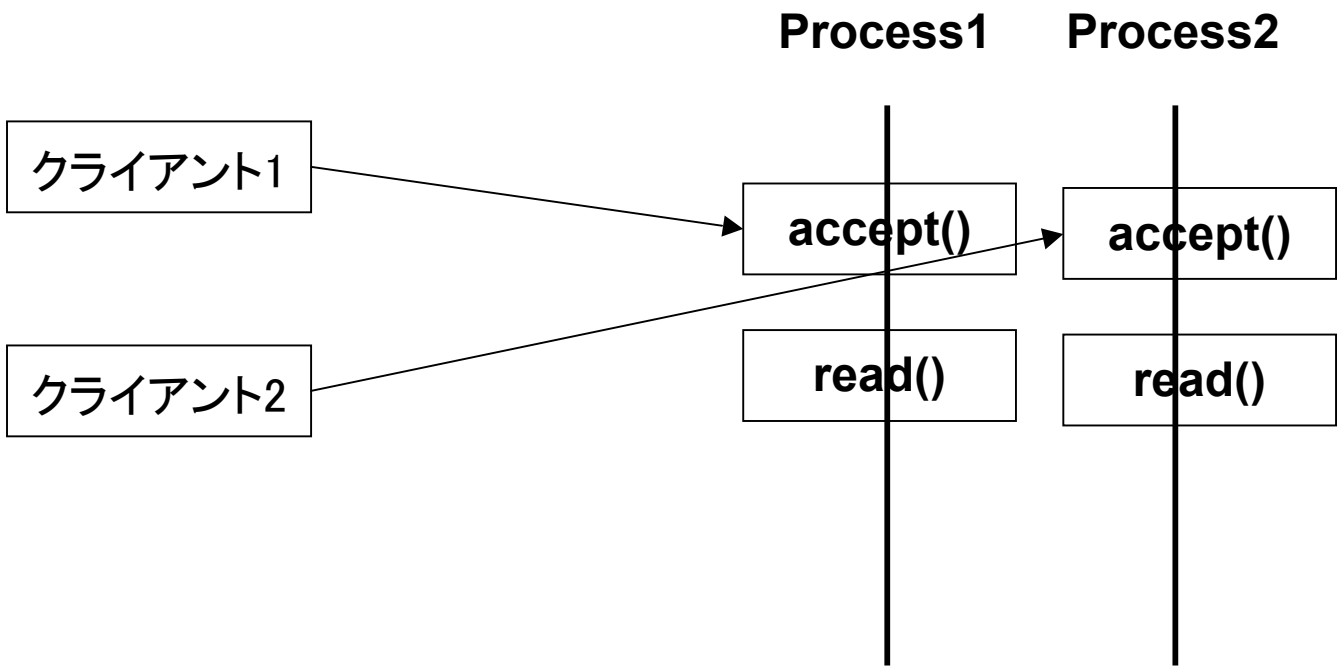
サーバの処理が並列化されていないと
クライアントは同時に1ノードしか処理できない

どこが問題？

- 多数のクライアントの処理を同時に受け付けるためには？
- accept()し、read()/write()している最中は他のクライアントからの接続要求(accept)を受け付けられない
- すべての処理を並列に処理できればよい？
- accept()したあと、処理を並列にできればよい？

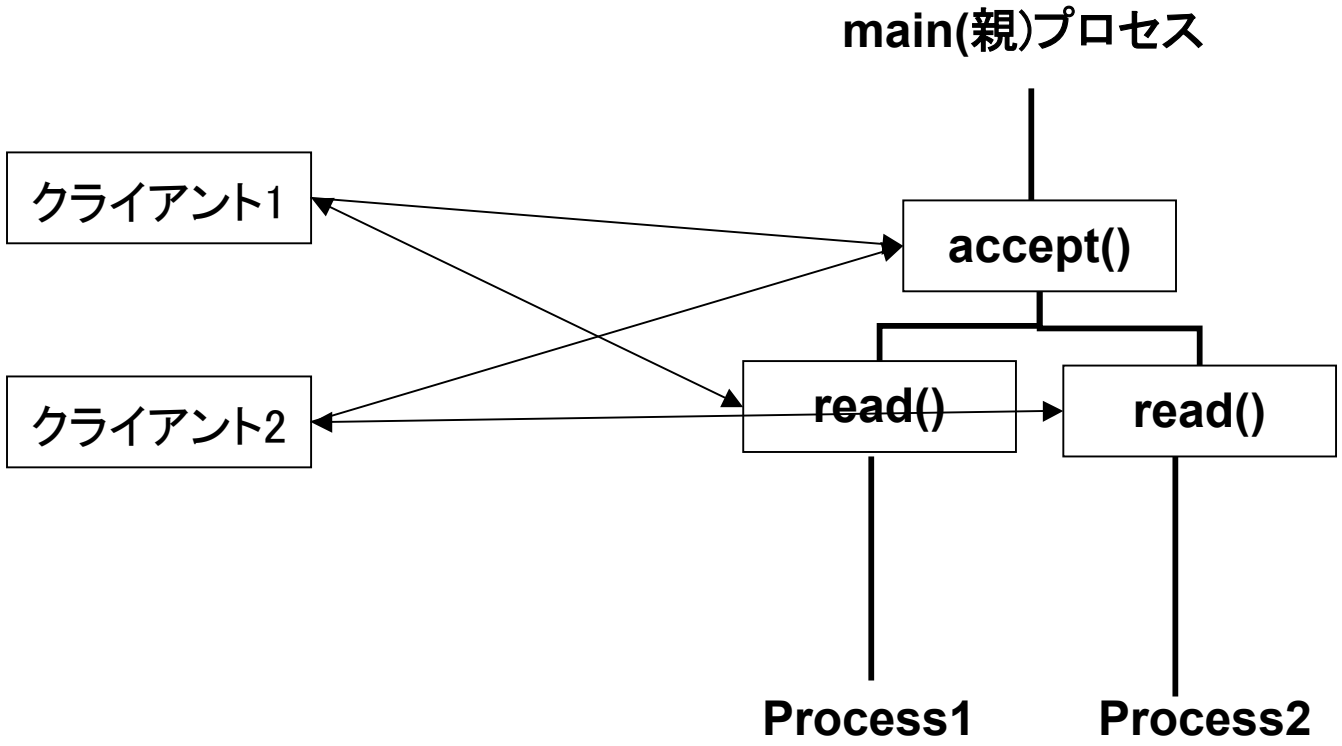


サーバプロセスを多重に起動



複数クライアントがひとつのサーバに接続可能だが、
サーバは複数のプロセス間で、同じポート番号を使用できない

プロセスをクライアント接続後複製



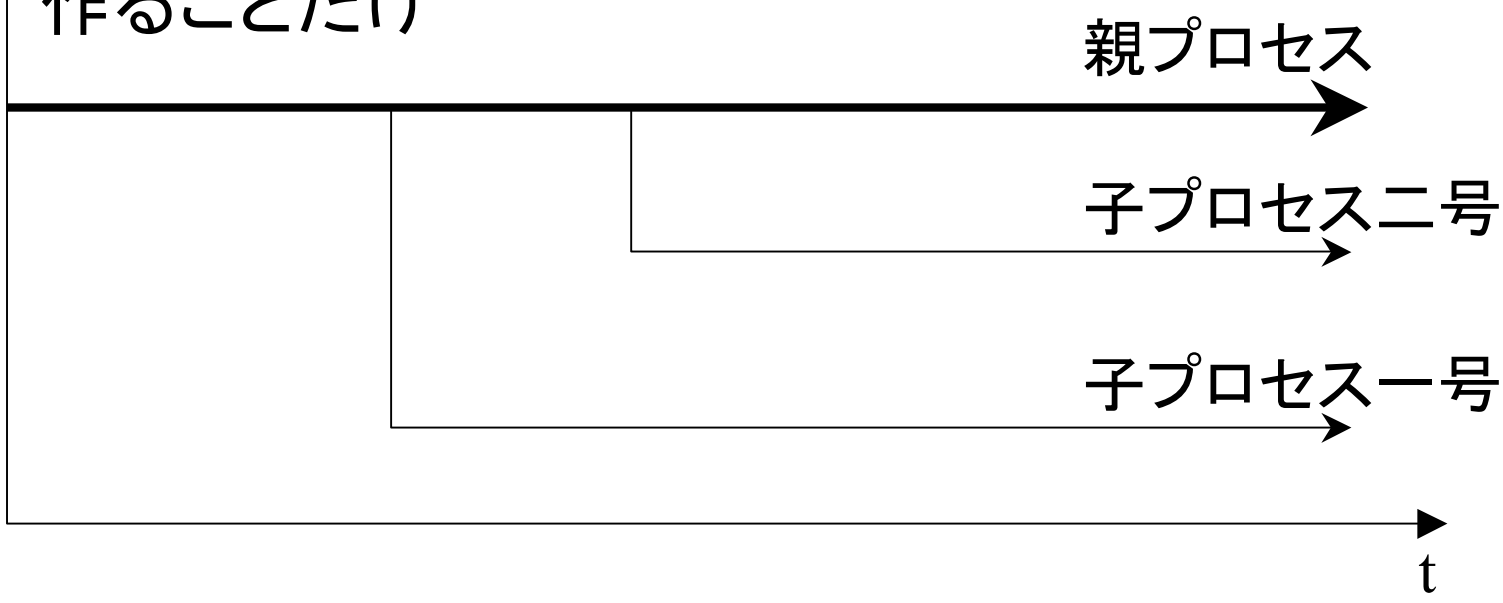
複数クライアントがひとつのサーバに接続可能で、
ポート番号も同じものでアクセス可能

並列処理の手法のいろいろ

- プロセス多重起動による、処理並列化
- fork()による、プロセスの複製
- pthread()による、スレッドの作成による処理並列化
- select()/poll()による、ディスクリプタ処理のみの処理の多重化

自分の分身を作ればいいんだ！

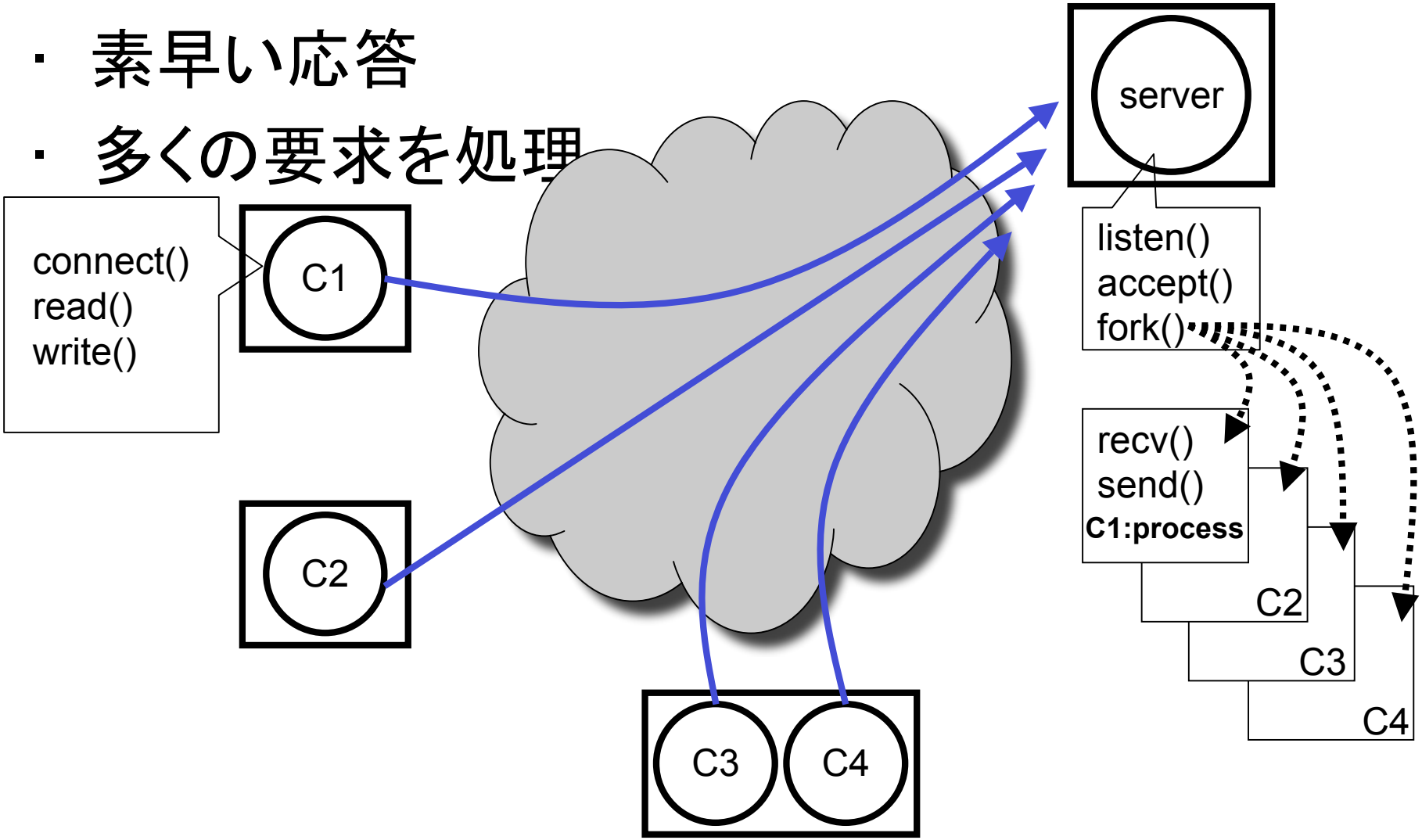
- 仕事は全部，分身にまかせよう
  - 親の仕事は，コネクションがきたときに分身を作ることだけ



forkシステムコールによる、プロセス複製

並列処理(concurrent processing)

- 素早い応答
- 多くの要求を処理



分身のための関数

- forkシステムコール（詳しくはman 2 fork）
  - int 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
  - 自分がコピーかどうかわからなくならない？
    - 0が帰ってきたら自分は子プロセス
    - 正の整数が帰ってきたら自分はfork()を呼び出した親プロセス
  - −1は失敗

プロセスを見る

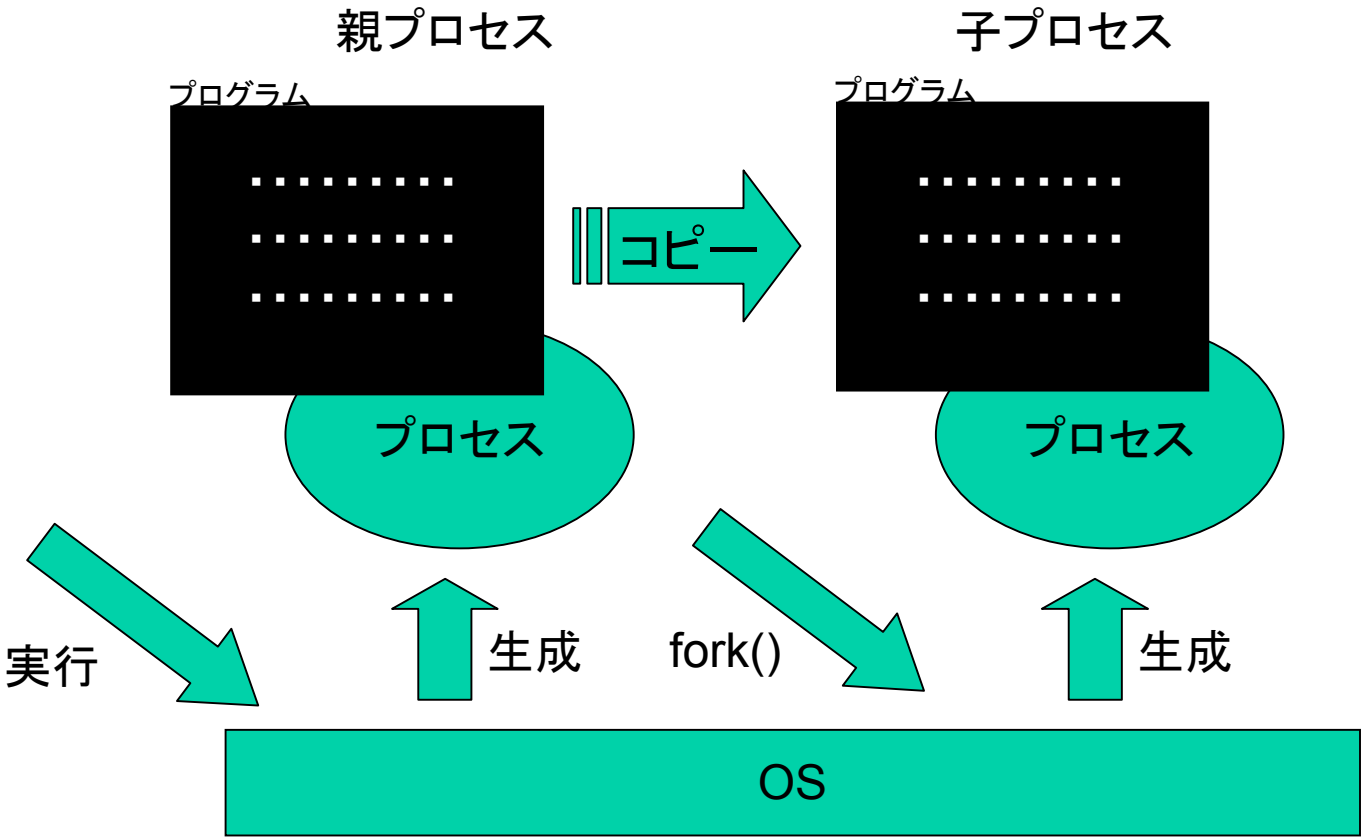
- Linux系(ccx00)
  - % ps -aux
- Solaris
  - % ps -efj
- BSD系
  - % ps auxj

USER	PID	PPID	PGID	……	COMMAND
nobody	123	89	89	……	httpd

プログラムの実行とプロセス

プログラム

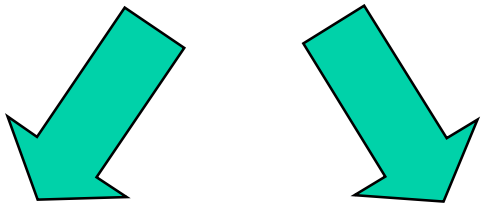
```
main(int argc, char
*argv)
{
    struct sockaddr_in
me, peer;
    .......
    if(pid = fork()) ==
0) {
                child
process.
    }
    else if(pid > 0) {
                parenet
process.
    }
    else {
                error
process.
    }
}
```



親と子の区別

fork()の返り値

```
int parent;
parent = fork();
```



親側

```
if(parent > 0){
   親の処理;
}
```

子側

```
else if(parent == 0){
  子の処理
}
```

練習1: forkに触れてみよう(次ページに解説あり)

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int  main(int argc, char *argv[]){
        int pid;
        int number = 5;
        char buf[1];

        printf("original process id: %d\n", getpid());
        printf("original parent process id: %d\n", getppid());

        if( (pid = fork()) == 0){ /* child process */
                printf("this is child process\n");
                number += 10;
                printf("number: %d\n", number);
                printf("fork() return value: %d\n", pid);
                printf("child process id: %d\n", getpid());
                printf("child parent process id: %d\n", getppid());
        } else if(pid > 0){ /* parent process */
                printf("this is parent process\n");
                number += 20;
                printf("number: %d\n", number);
                printf("fork() return value: %d\n", pid);
                printf("parent process id: %d\n", getpid());
                printf("parent parent process id: %d\n", getppid());
        } else {
                perror("fork()");
        }
        printf("number: %d\n", number);
        exit(0);
}
```

練習1:覚えた方がお得かも

- int getpid()
  - プロセスidを取得
- int getppid()
  - 親プロセスidを取得

練習1：重要な点①

- forkの基本構文

```
    int pid;

if( (pid = fork()) == 0){
        /* child process */
} else if(pid > 0){
        /* parent process */
} else {
        perror("fork()");
}
```

練習1:重要な点②

- 変数がコピーされる
  - int numberの中身を考えてみよう
- 別のメモリ領域が確保される

original process id: 21146
original parent process id: 20587

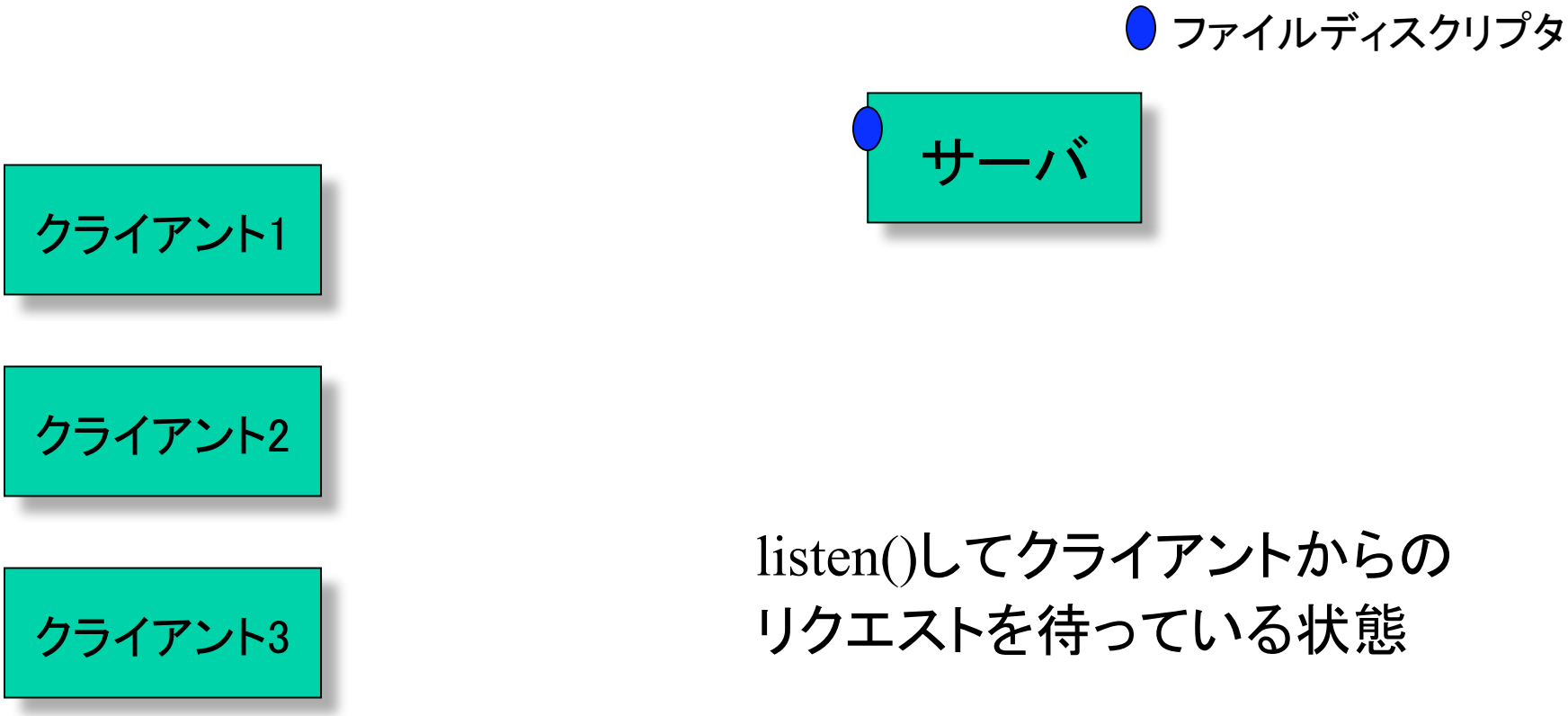
this is child process
number: 15
fork() return value: 0
child process id: 21147
child parent process id: 21146
number: 15

this is parent process
number: 25
fork() return value: 21147
parent process id: 21146
parent parent process id: 20587
number: 25
press enter

よくあるサーバでの処理の流れ

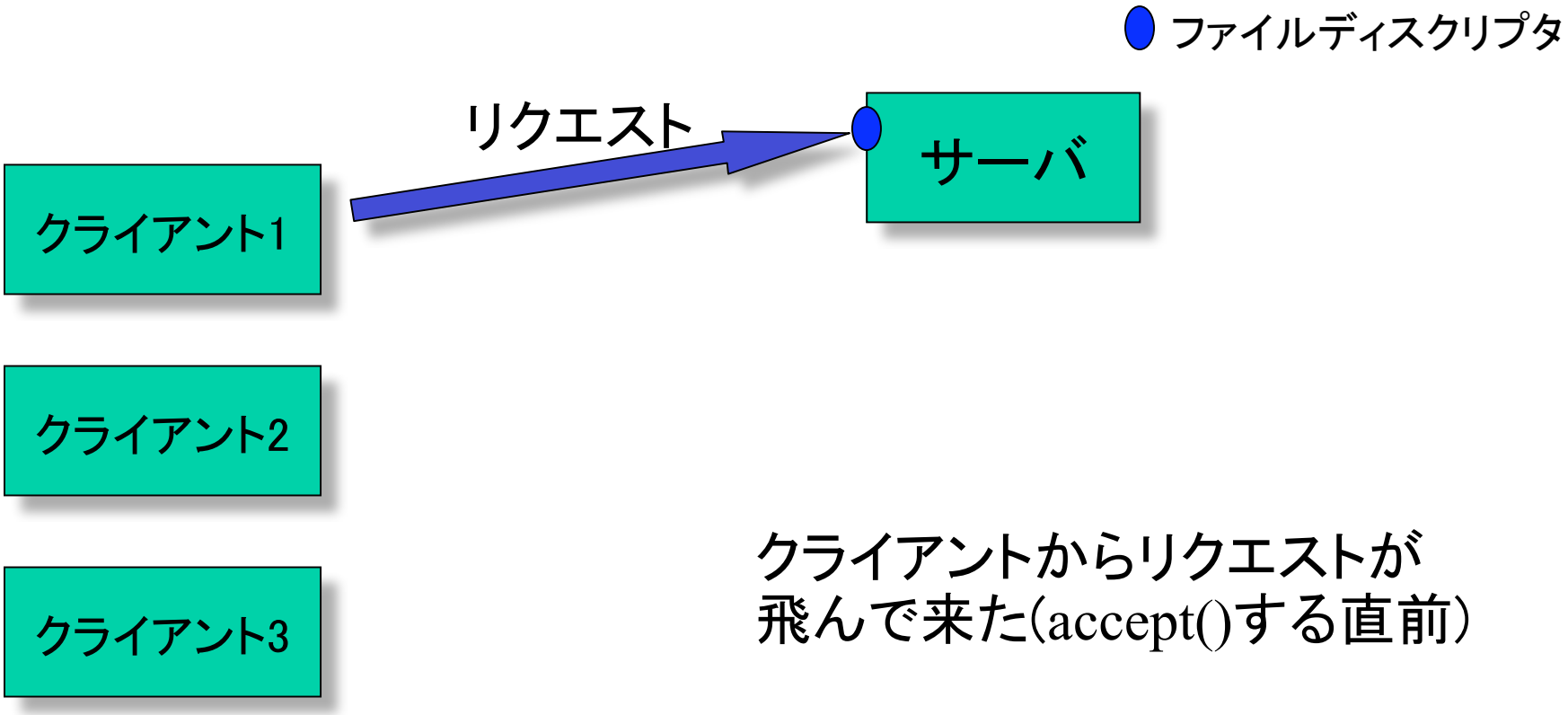
- 時間のかかるサービス(telnet, ssh, ftpなど)を提供するサーバでよくある処理
- 接続要求があったら, acceptして, とりあえずfork();
  - 子プロセスだったら、サービスの提供開始
  - 親プロセスだったら、接続要求待ち
- 自分でたくさんのソケットの面倒を見なくてよい

fork()を用いたマルチクライアントサーバの仕組み



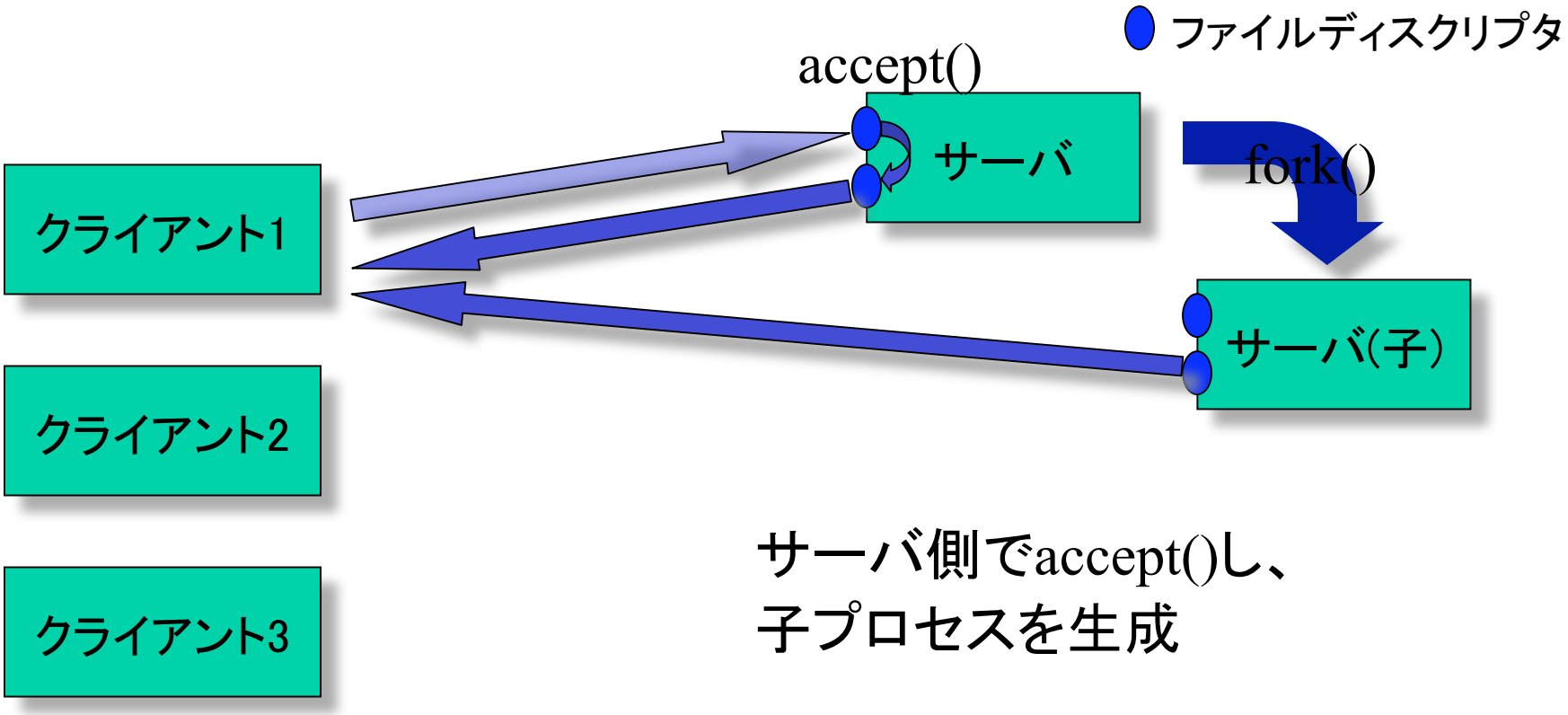
listen()してクライアントからのリクエストを待っている状態

fork()を用いたマルチクライアントサーバの仕組み



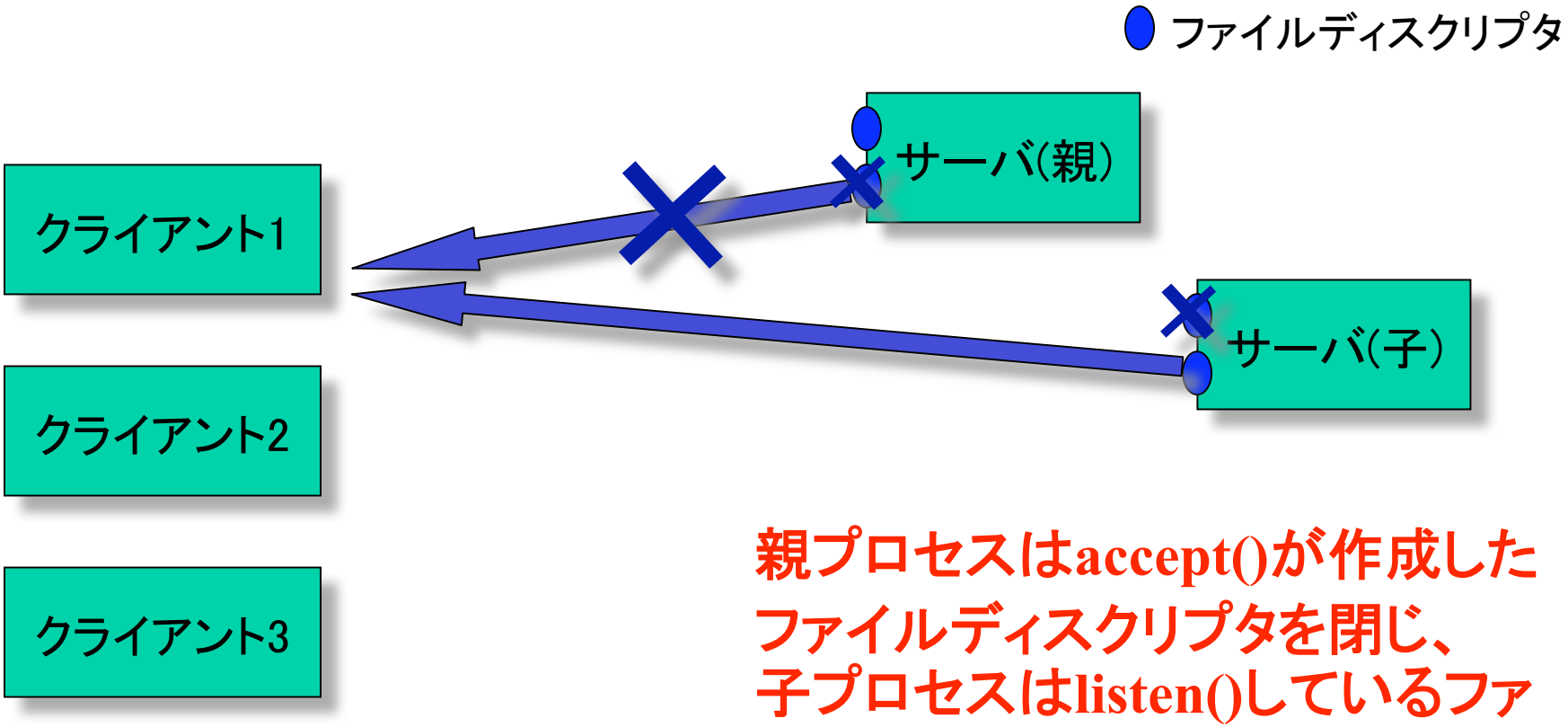
クライアントからリクエストが
飛んで来た(accept()する直前)

fork()を用いたマルチクライアントサーバの仕組み



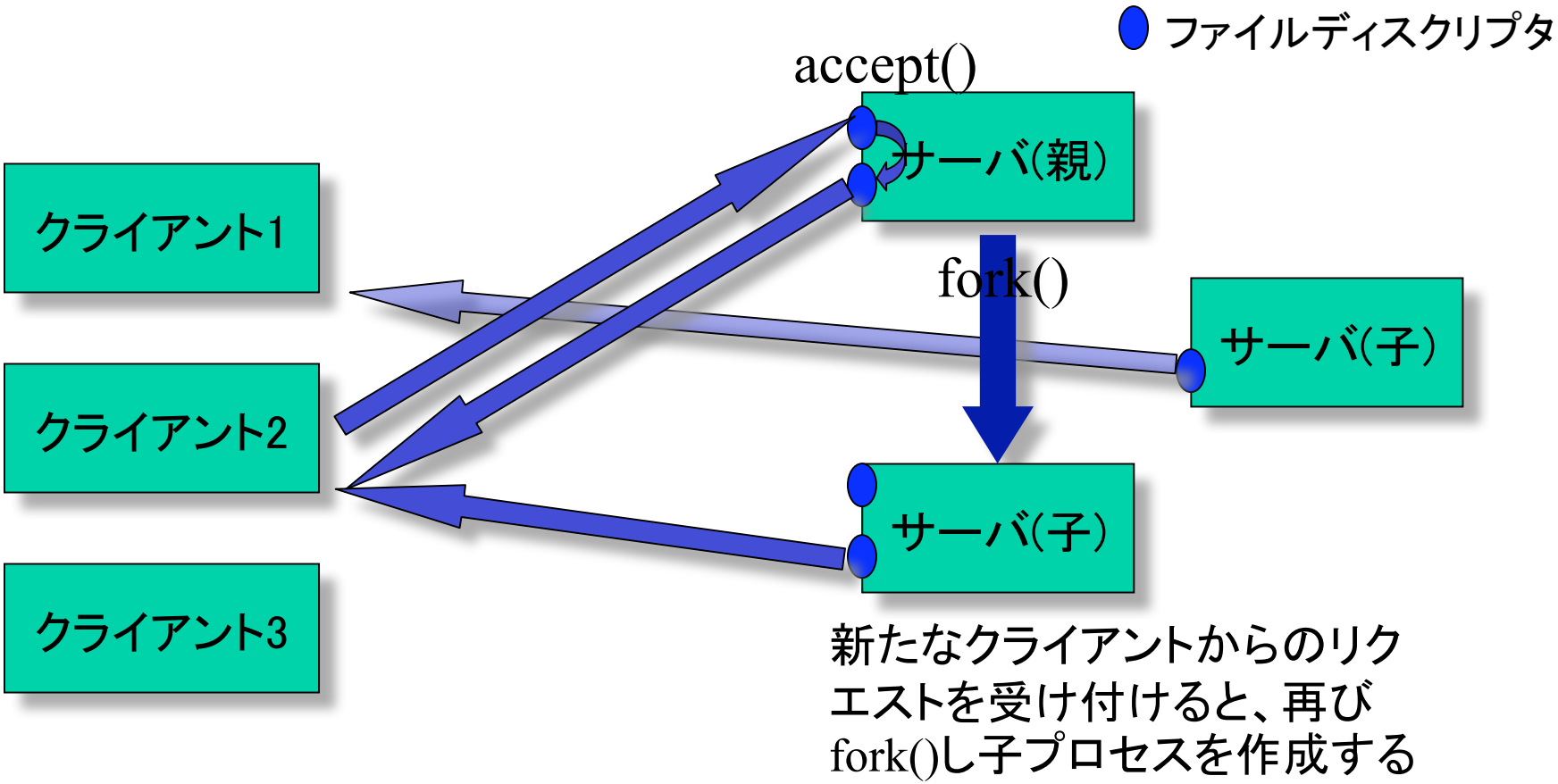
サーバ側でaccept()し、
子プロセスを生成

fork()を用いたマルチクライアントサーバの仕組み

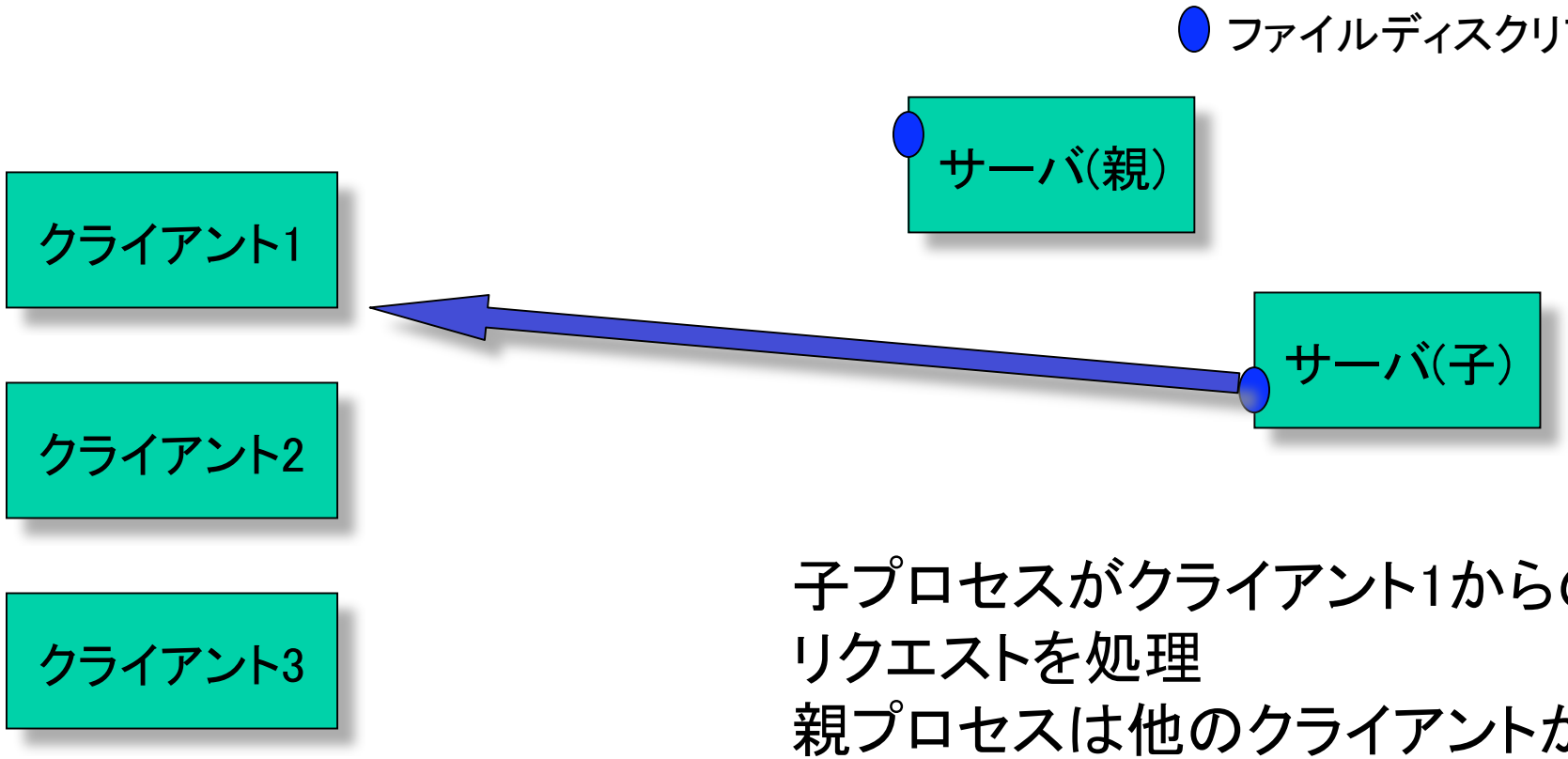


親プロセスはaccept()が作成したファイルディスクリプタを閉じ、子プロセスはlisten()しているファイルディスクリプタを閉じる

fork()を用いたマルチクライアントサーバの仕組み



fork()を用いたマルチクライアントサーバの仕組み

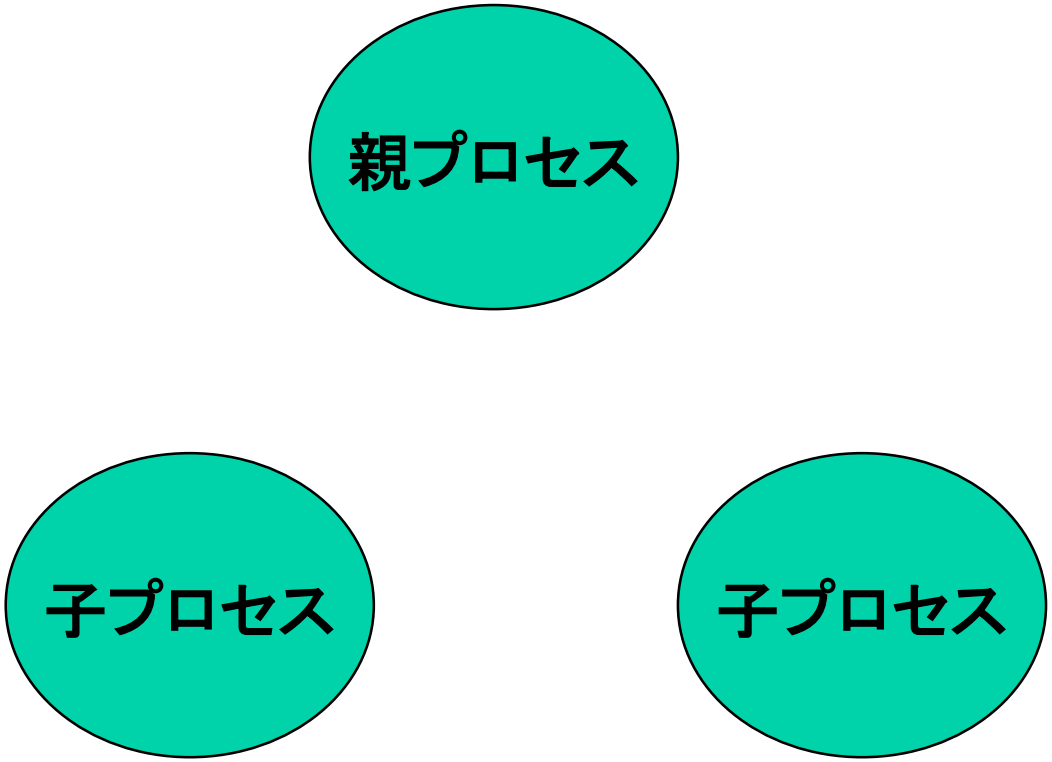


子プロセスがクライアント1からのリクエストを処理
親プロセスは他のクライアントからの接続要求を待ちつづける

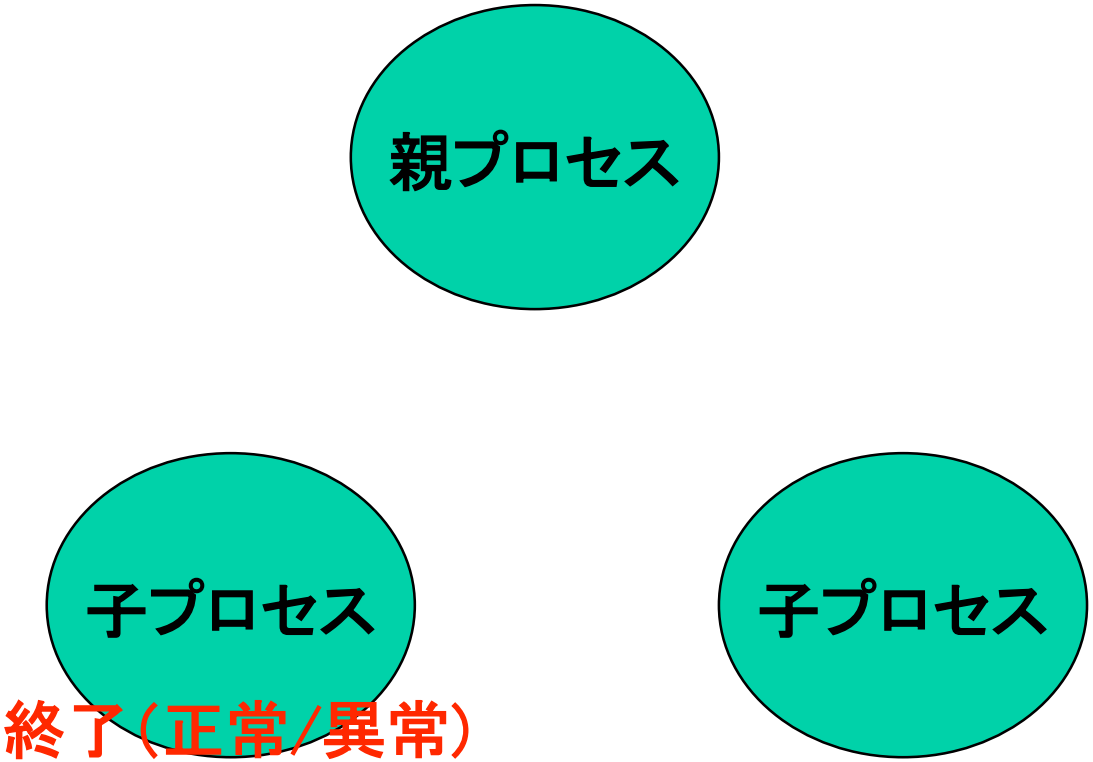
fork()を利用した並行サーバの注意点　（親プロセスの義務）

- 子プロセスが終了すると、「どのような死に様だったか」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
    - 非同期な事象をプログラムで扱うための方法
- 親プロセスは子プロセスの死に様を見届けなければならない
  - 親プロセスが子プロセスの終了状態を受け取らなくては，子プロセスは成仏できずにゾンビプロセスに変わる
    - psコマンドで確認すると，a.out <defunct>と表示される
- 子プロセスが死んで，終了状態を受け取らずに親プロセスが死ぬと，ゾンビプロセスがいつまでも残る
- 子プロセスが終了する前に親プロセスが死ぬと？
  - プロセスID１のinitが養子として引き取ってくれる

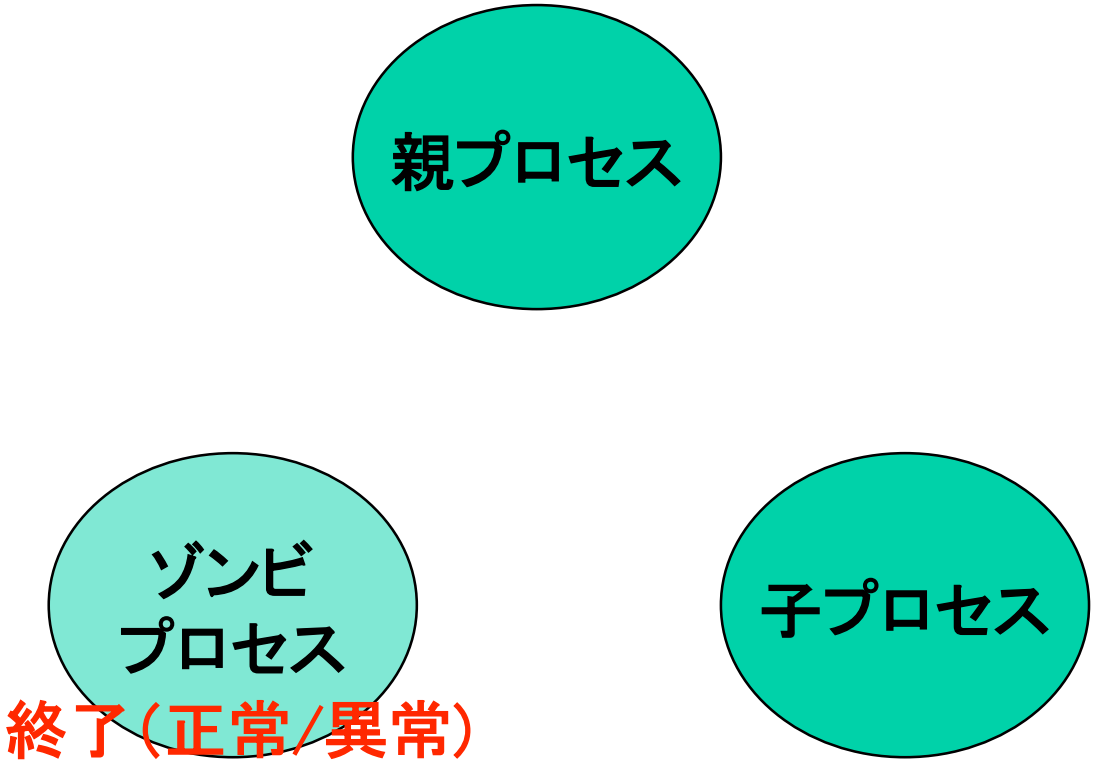
終了状態の受け渡し



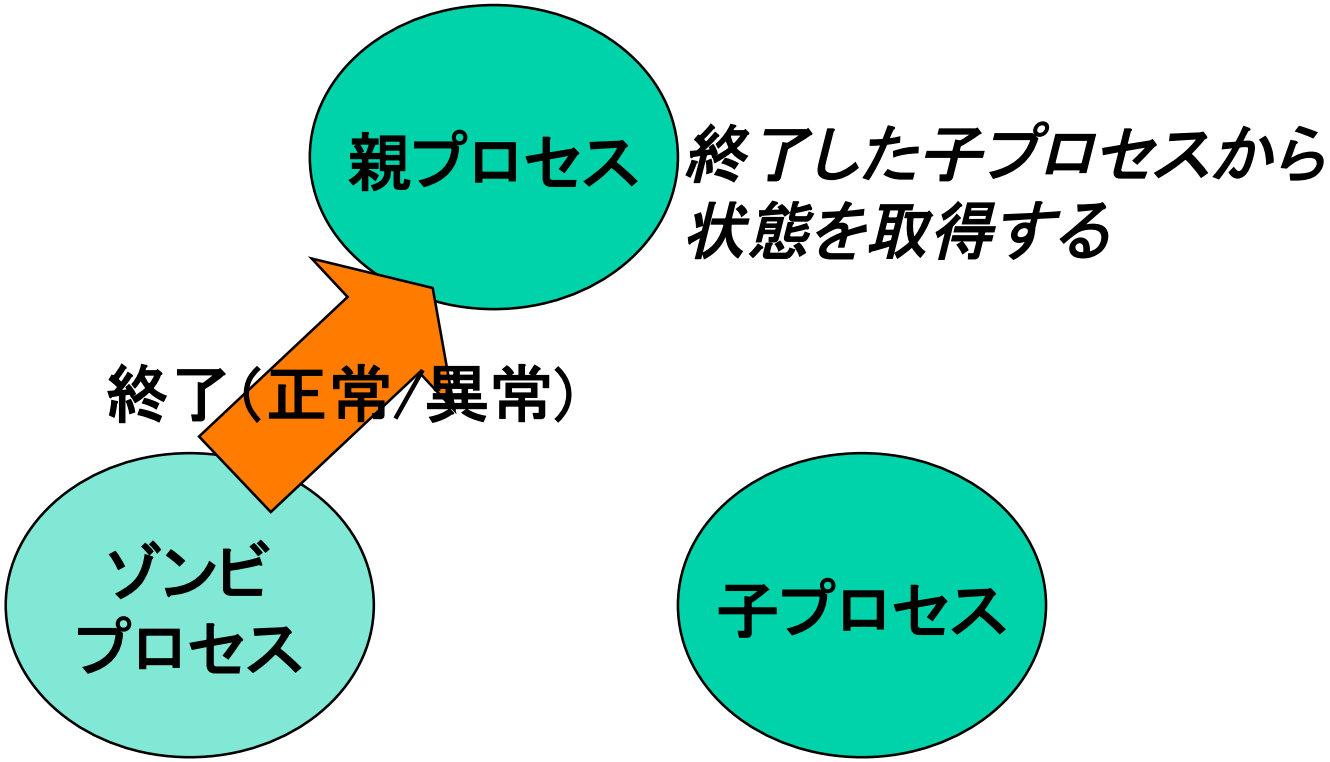
終了状態の受け渡し



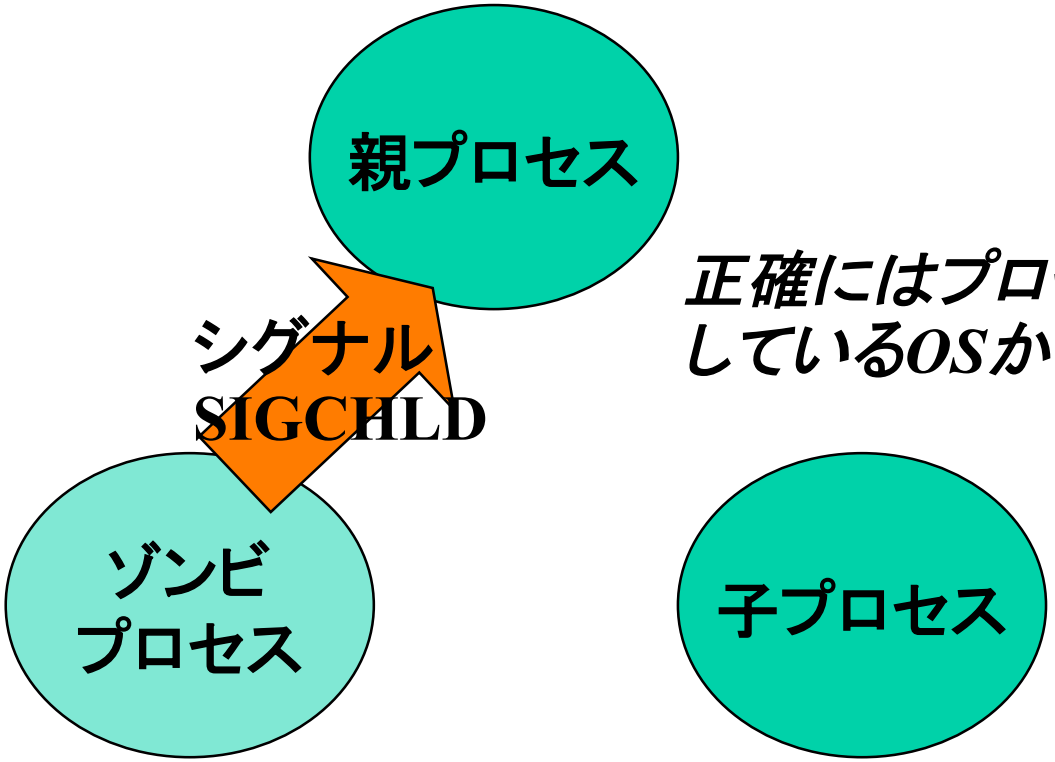
終了状態の受け渡し



終了状態の受け渡し

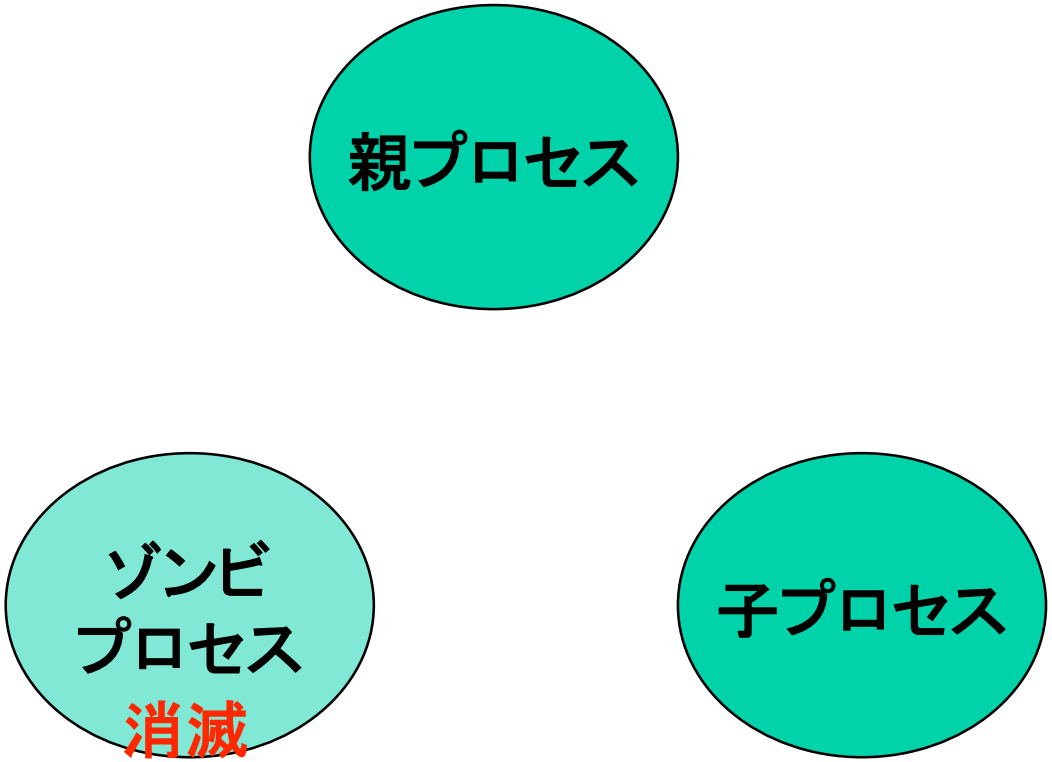


終了状態の受け渡し



正確にはプロセスを管理しているOSから通知される

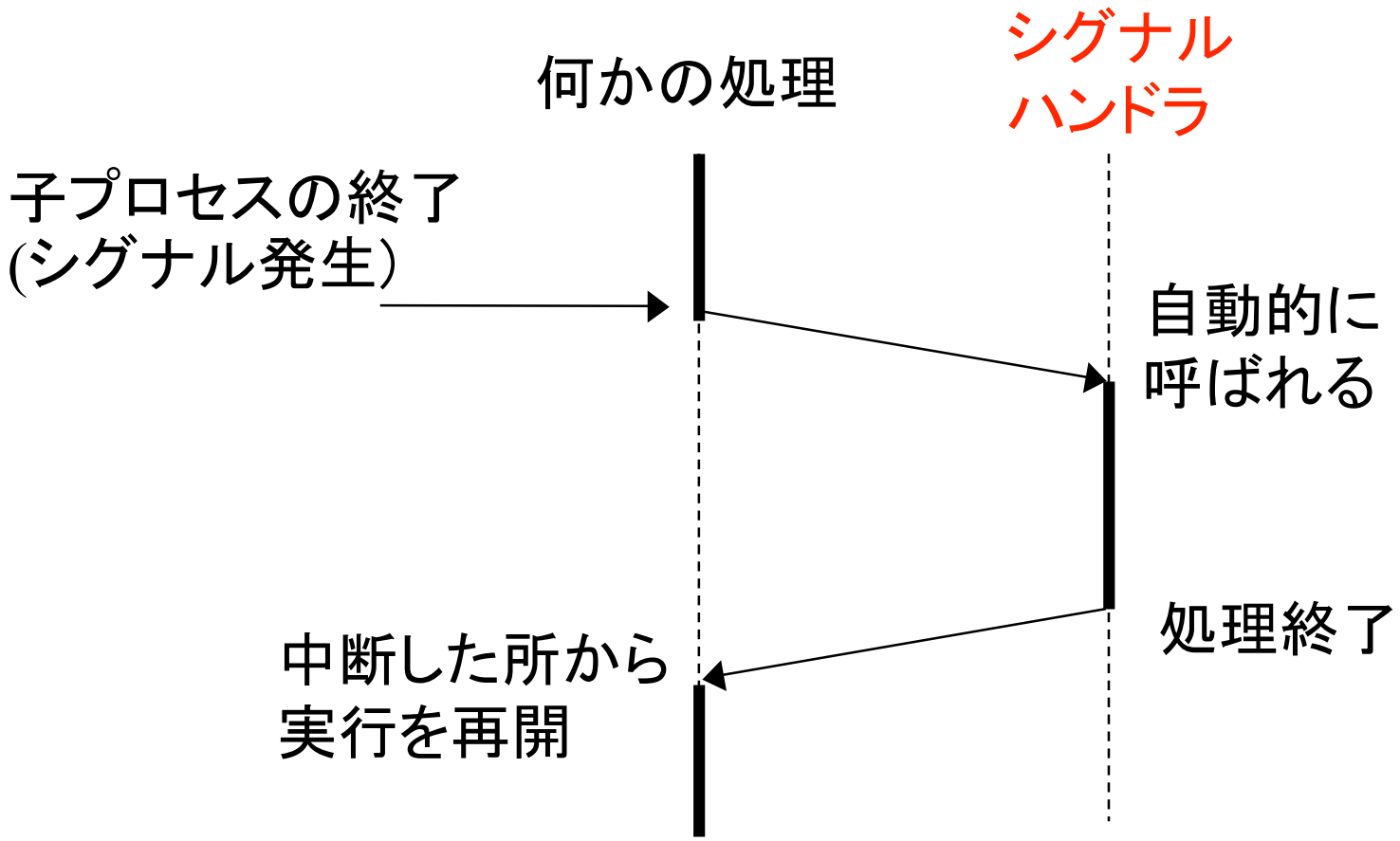
終了状態の受け渡し



親プロセスの義務

- つまり、子プロセスが終了したというシグナルを受け取ったら「南無阿弥陀仏」と唱えなければいけない
  - 子プロセスの終了状態(ステータス)を受け取る
- 呪文： wait() システムコール
  - 子プロセスの状態を受け取り、変数へ代入
  - 子プロセスの終了状態から、適切な処理を進められる
- 注意： シグナルのキューイング
  - 複数のシグナルを同時に受信しても「1つ」しか通知されない
  - 複数の子プロセスが同時に死んだ場合の処理を工夫

時間的な処理の流れ


何かの処理
シグナル
ハンドラ
子プロセスの終了
(シグナル発生)
自動的に
呼ばれる
処理終了
中断した所から
実行を再開

waitシステムコール群

- 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
- pid_t wait(int *status);
- pid_t waitpid(pid_t wpid, int *status, int option);
  - 子プロセスのpidが戻り値
  - 失敗: -1

wait()とwaitpid()の違い

- waitは子プロセスが終了する（ほんとは状態が変わる）までブロックする
  - 親プロセスは別の処理を行えない。
- 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場合
  - pidを指定できる
    - −1 を指定すると最初に状態の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

サンプルコード

```
void sig_child(int signo)
{
        int pid, status;
        while((pid = waitpid(-1, &status, WNOHANG)) > 0) {
                printf("PID: %d, terminated\n", pid);
        }
}

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

実習・課題

- 以前つくったTCPエコーサーバをforkを用いて並行処理可能にする
- 前回との違い
  - Server:
    - forkする
    - クライアントから“bye”を受け取ると通信を終了する
      - bye以外は全てclientにエコーする
    - waitpidする(子プロセスの終了を受け取る)
- 複数クライアントからサーバに接続し，psコマンドを用いてforkが出来ていることを確認すること