

ネットワークコンピューティング 第5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
- 授業用ログインノード
 - login1.sekiya-lab.info
 - login2.sekiya-lab.info
 - とともに外部から ssh で login できる Linux マシン

課題1 (クライアントプログラム)

- daytime client を作成する (×切 2018/05/09)
 - 引数でアドレスを取れるようにする
 - エラー処理をきちんとする
 - IPv4 だけではなく IPv6 にも対応する
- 提出されたレポートの概要
 - IPv4 用プログラムと IPv6用プログラムを別に作成
 - IPv4 アドレスも IPv6 アドレスも引数にできるプログラム
 - IP アドレスを引数にしたプログラム
 - (まだ説明していない) 名前解決機構を使ったプログラム
- 提出された方には受領メールを送信しました

daytime プログラム例 (エラー処理なし)

- IPv4用

```
int main(int argc, char **argv) {
    int sockfd, n;
    char recvline[MAXLINE+1];
    struct sockaddr_in servaddr4;

    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    memset(&servaddr4, 0, sizeof(servaddr4));
    servaddr4.sin_family = AF_INET;
    servaddr4.sin_port = htons(13);
    inet_pton(AF_INET, argv[1], &servaddr4.sin_addr);
    connect(sockfd, (SA *)&servaddr4, sizeof(servaddr4));

    while ((n = read(sockfd, recvline, MAXLINE)) > 0) {
        recvline[n] = '\0';
        fputs(recvline, stdout);
    }
    close(sockfd);
}
```

- IPv6用

```
int main(int argc, char **argv) {
    int sockfd, n;
    char recvline[MAXLINE+1];
    struct sockaddr_in6 servaddr6;

    sockfd = socket(AF_INET6, SOCK_STREAM, 0);
    memset(&servaddr6, 0, sizeof(servaddr6));
    servaddr6.sin6_family = AF_INET6;
    servaddr6.sin6_port = htons(13);
    inet_pton(AF_INET6, argv[1], &servaddr6.sin6_addr);
    connect(sockfd, (SA *)&servaddr6, sizeof(servaddr6));

    while ((n = read(sockfd, recvline, MAXLINE)) > 0) {
        recvline[n] = '\0';
        fputs(recvline, stdout);
    }
    close(sockfd);
}
```

IPv4 と IPv6 で異なるのは通信の確立までの処理のみ

daytime プログラム例 (エラー処理なし)

- IPv4/IPv6両用

```
int main(int argc, char **argv) {
    int sockfd, n;
    char recvline[MAXLINE+1];
    struct sockaddr_in servaddr4;
    struct sockaddr_in6 servaddr6;
```

```
    memset(&servaddr4,0,sizeof(servaddr4));
    memset(&servaddr6,0,sizeof(servaddr6));
```

```
    if (inet_pton(AF_INET,argv[1],&servaddr4.sin_addr) > 0 ) {
        sockfd = socket(AF_INET, SOCK_STREAM,0);
        servaddr4.sin_family = AF_INET;
        servaddr4.sin_port = htons(13);
        connect(sockfd, (SA *) &servaddr4, sizeof(servaddr4));
    } else if (inet_pton(AF_INET6,argv[1],&servaddr6.sin6_addr) > 0) {
        sockfd = socket(AF_INET6, SOCK_STREAM,0);
        servaddr6.sin6_family = AF_INET6;
        servaddr6.sin6_port = htons(13);
        connect(sockfd, (SA *) &servaddr6, sizeof(servaddr6));
    } else {
```

```
        perror("inet_pton");
    }
```

```
    while ((n = read(sockfd, recvline, MAXLINE)) > 0 ) {
        recvline[n] = '\0';
        fputs(recvline, stdout);
    }
    close(sockfd);
}
```

他のクライアントプログラム例 (echo)

- daytime プログラムの実質的な処理部分

- (サーバと接続した) sockfd からデータを読み込んで、その内容を標準出力に出力する処理

```
while ((n = read(sockfd, recvline, MAXLINE)) > 0) {  
    recvline[n] = '\0';  
    fputs(recvline, stdout);  
}
```

- echo プログラムの実質的な処理部分

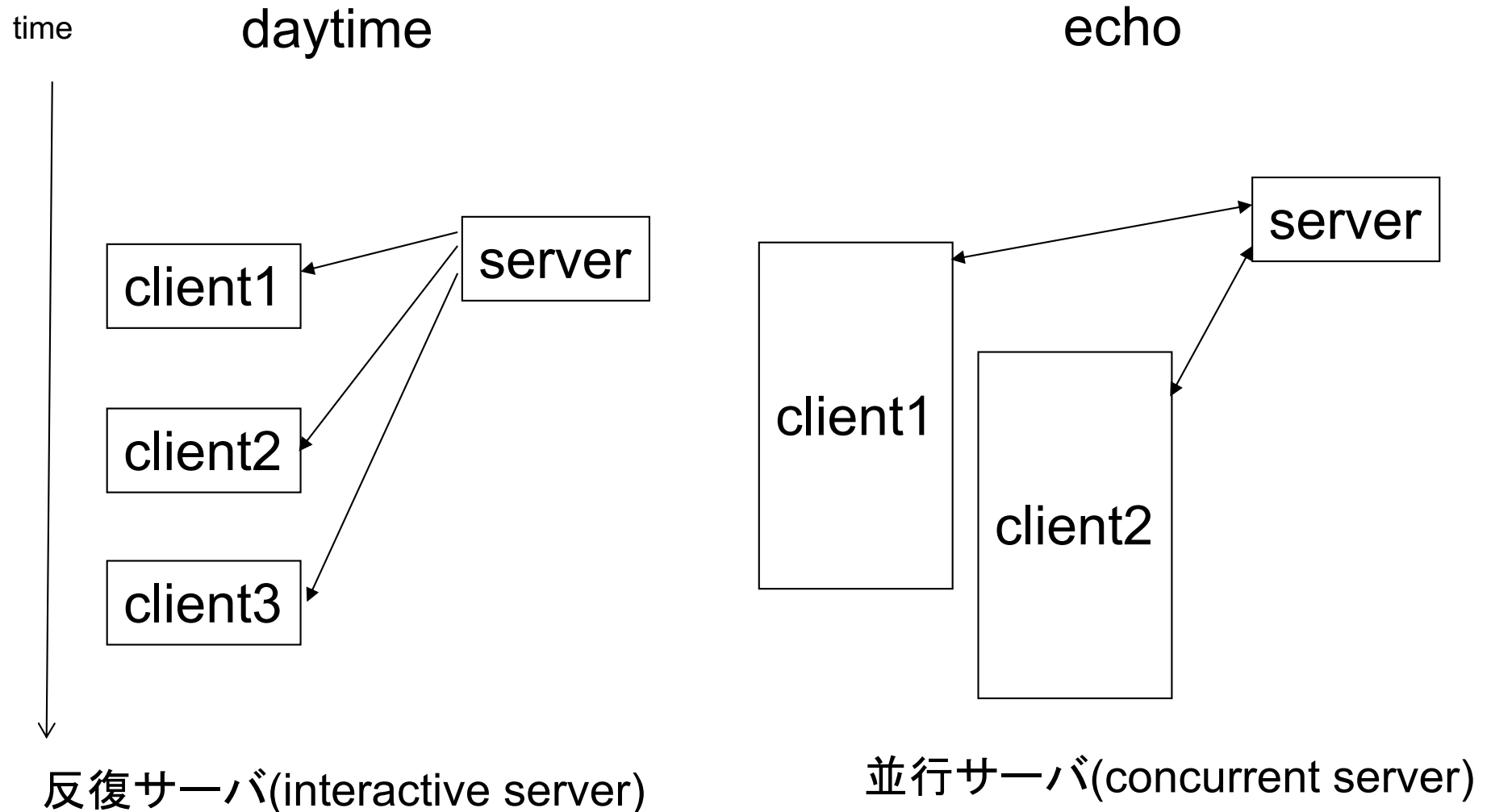
- キーボードから読み込んだデータを、(サーバと接続した) sockfd に書き込んだ後、sockfd からデータを読み込んで、その内容を標準出力に出力する処理

```
while (fgets(sendline, MAXLINE, stdin) != NULL) {  
    write(sockfd, sendline, strlen(sendline));  
    n = read(sockfd, recvline, MAXLINE);  
    recvline[n] = '\0';  
    fputs(recvline, stdout);  
}
```

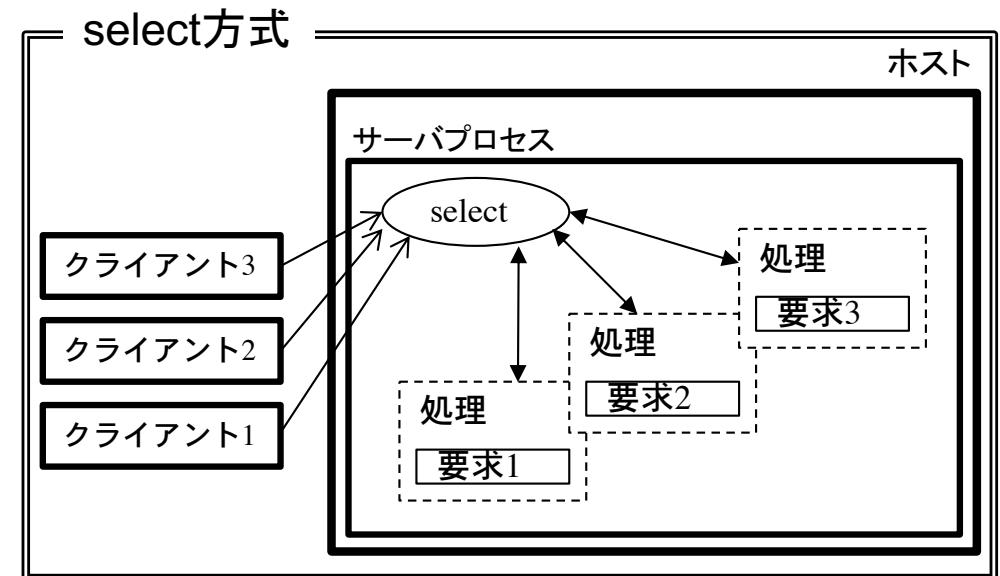
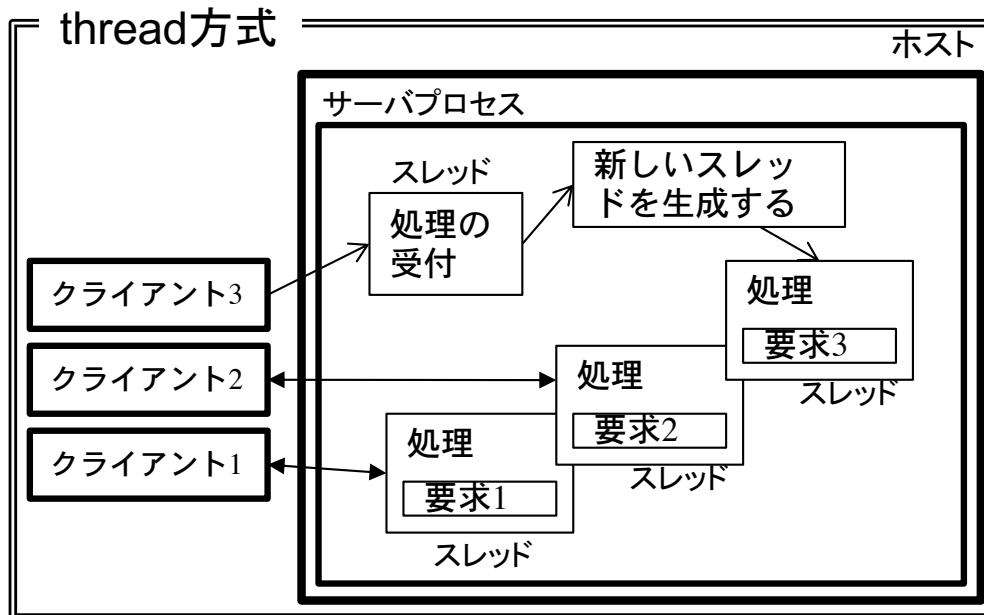
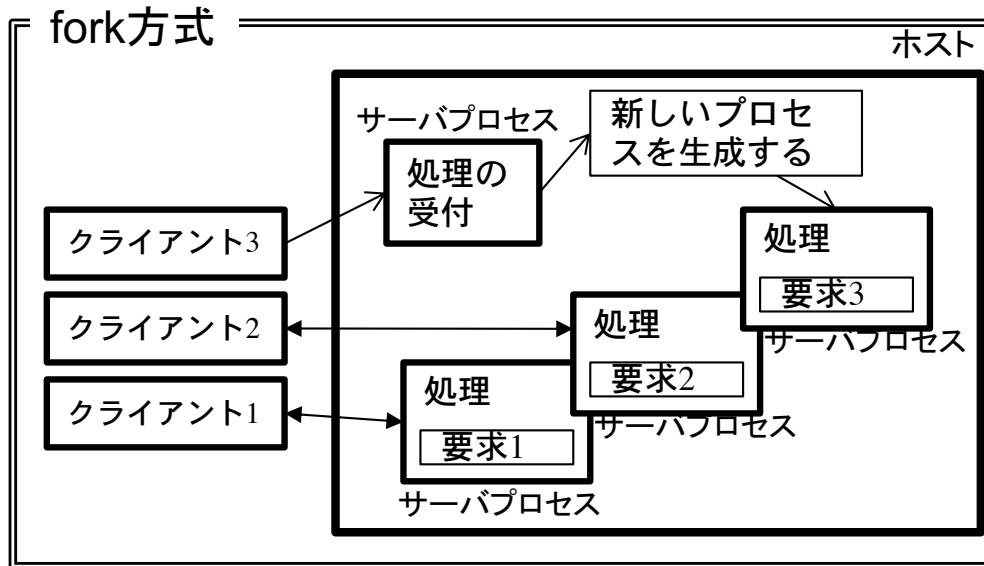
サーバプログラミング

- 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 などの簡潔な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



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 nfd,
           fd_set *restrict readfds,
           fd_set *restrict writefds,
           fd_set *restrict errorfds,
           struct timeval *restrict timeout);
```

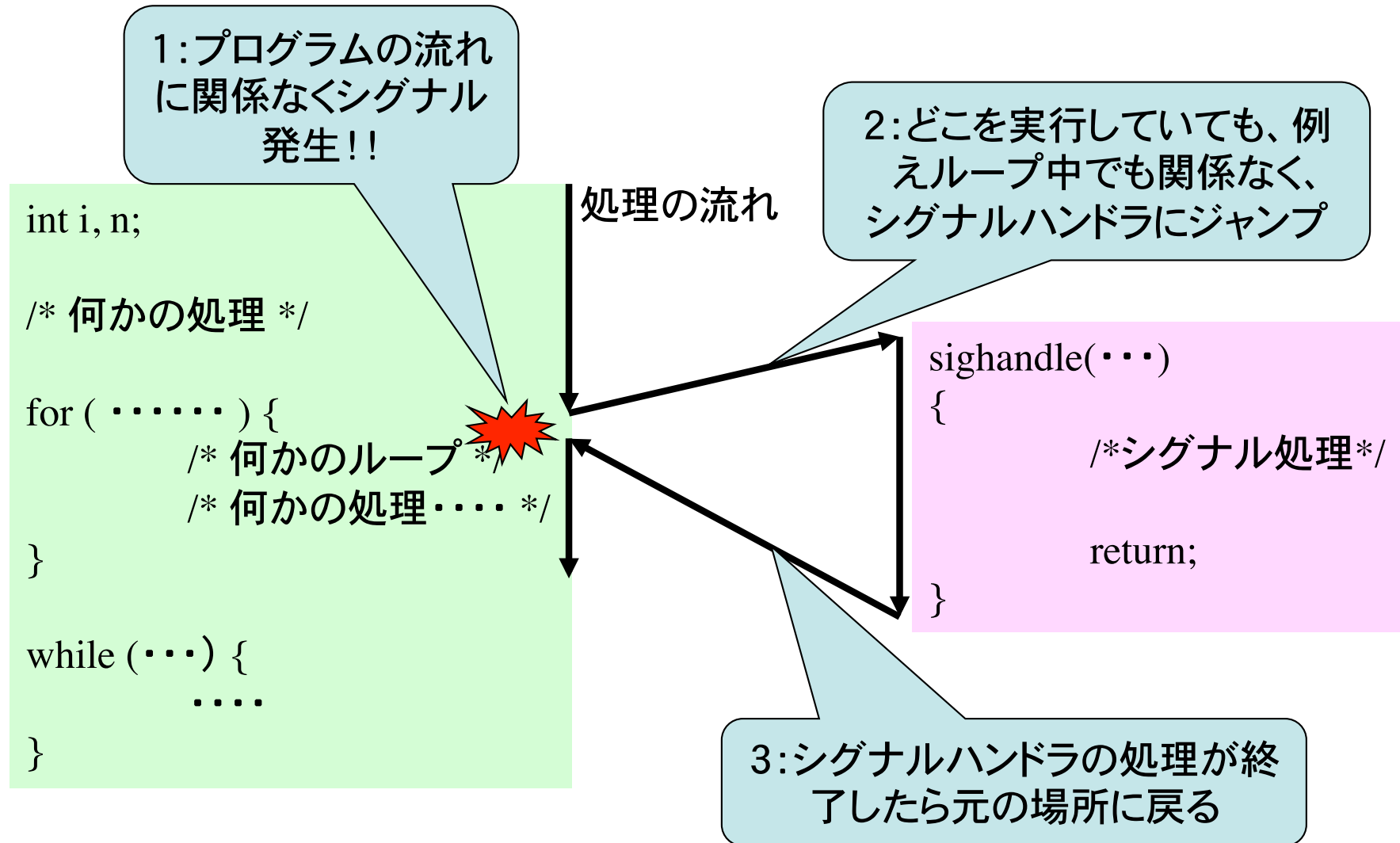
戻り値: ready descriptor の総数
エラーの時は、-1

FORK

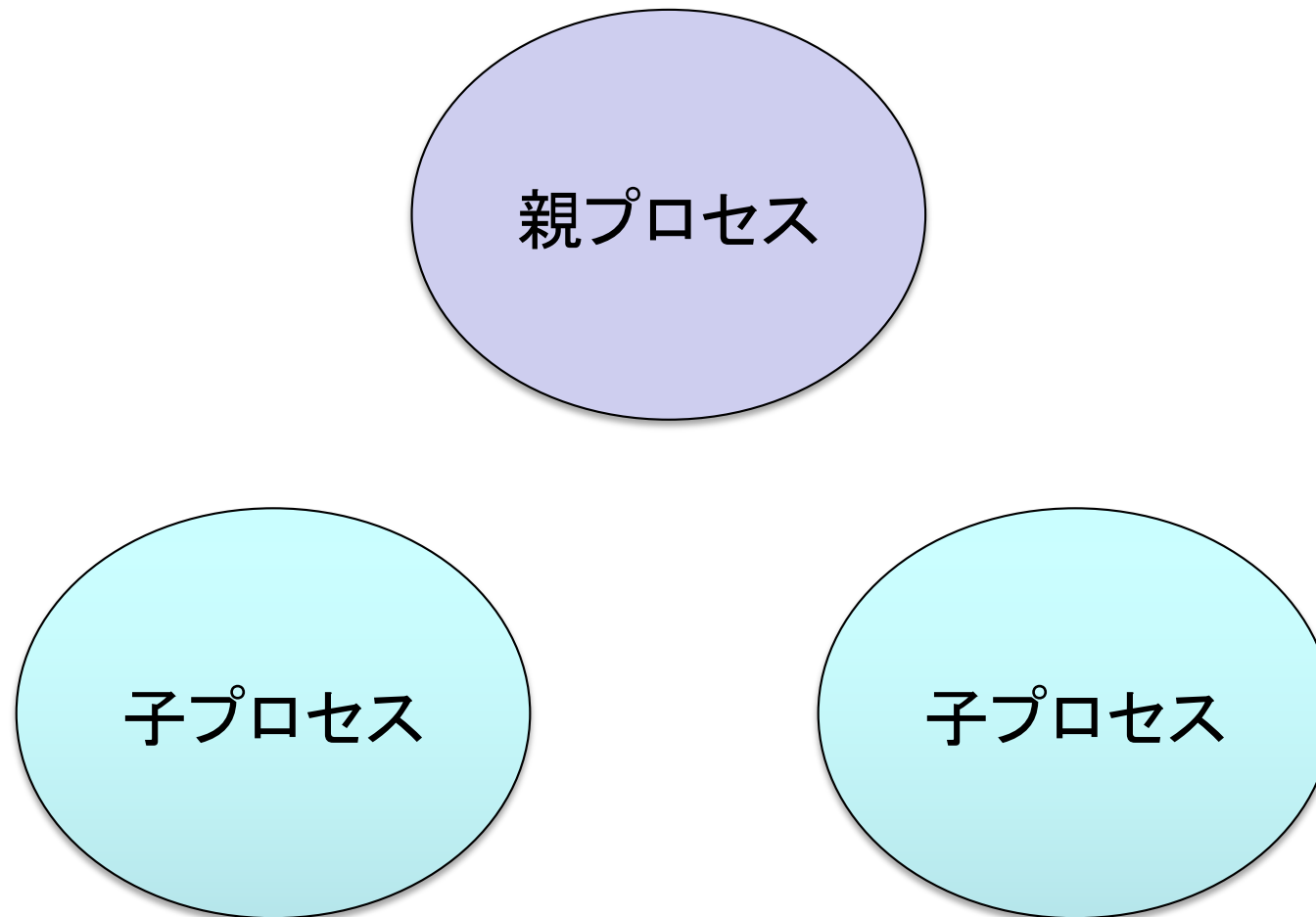
fork() を用いたサーバプログラミング

- 第4回 (2018/05/10) 資料を参照のこと
- 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ない
 - 子プロセスが終了し、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

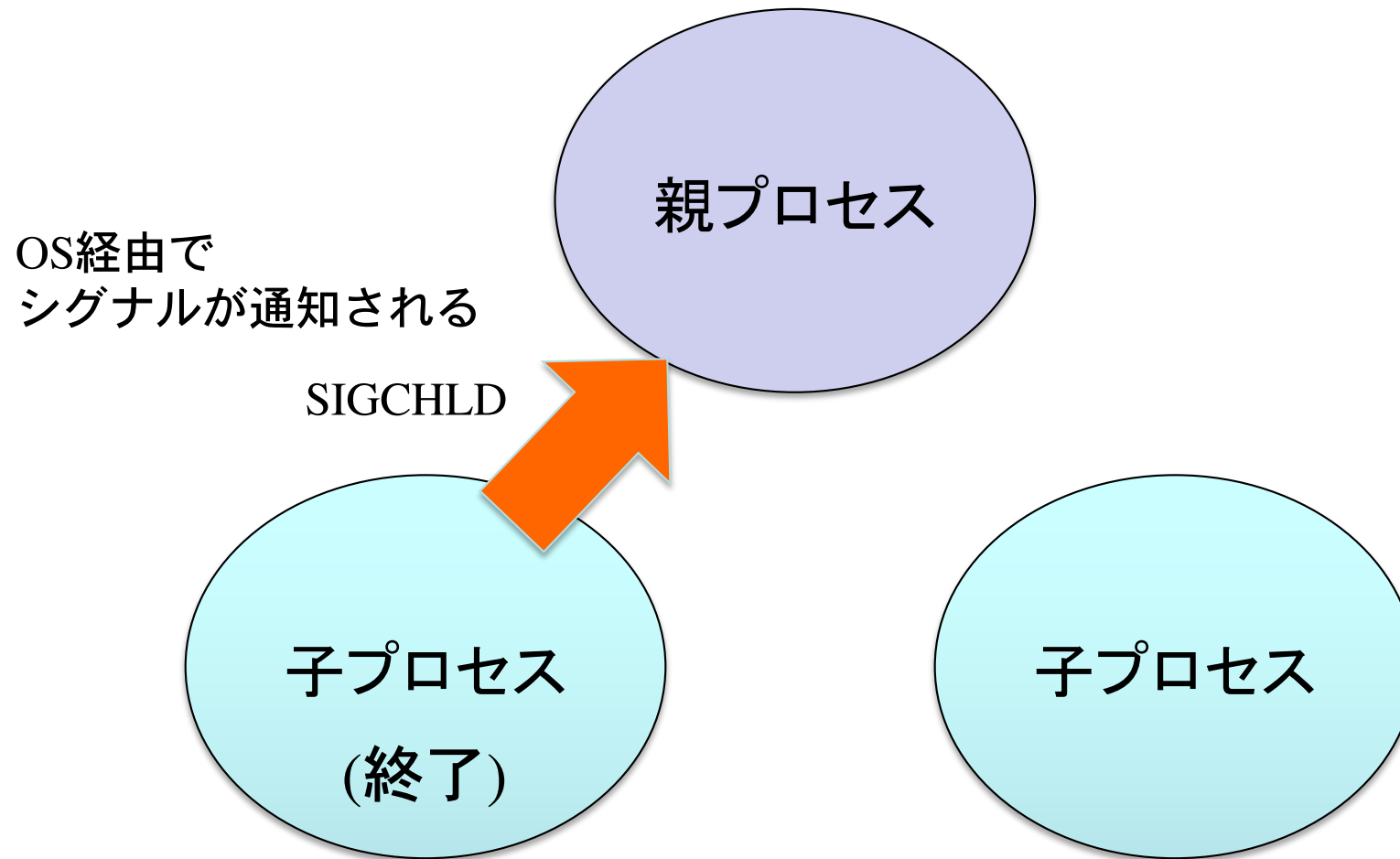
シグナルハンドラ



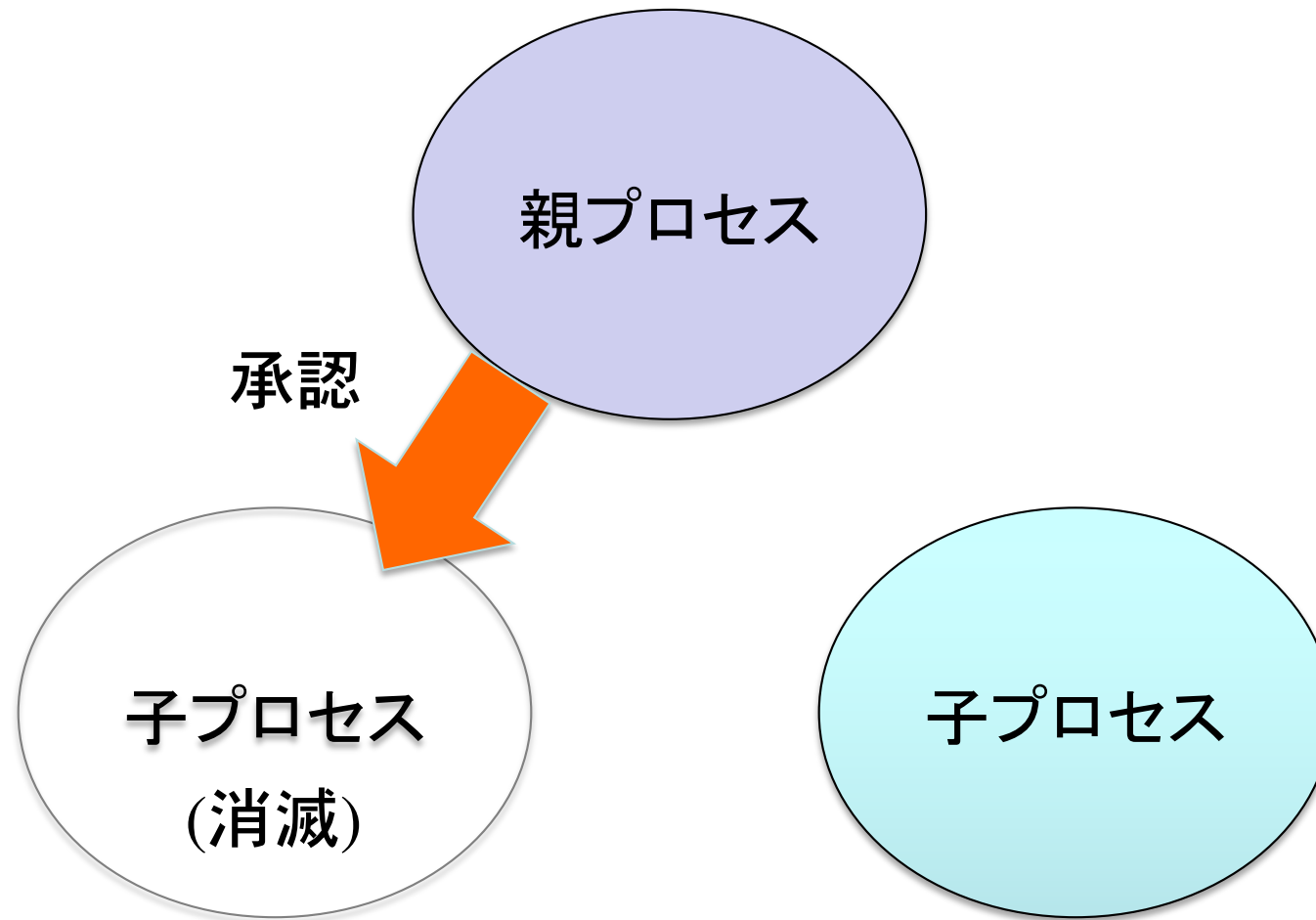
終了状態の受け渡し(正常)



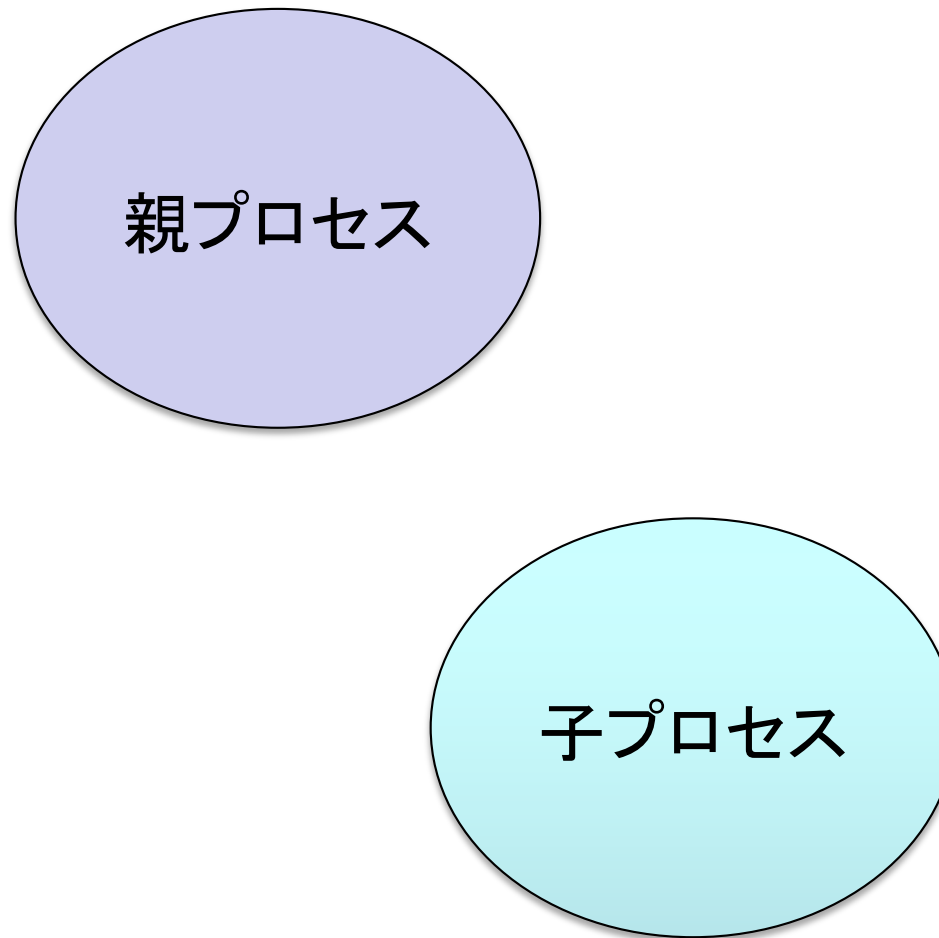
終了状態の受け渡し（正常）



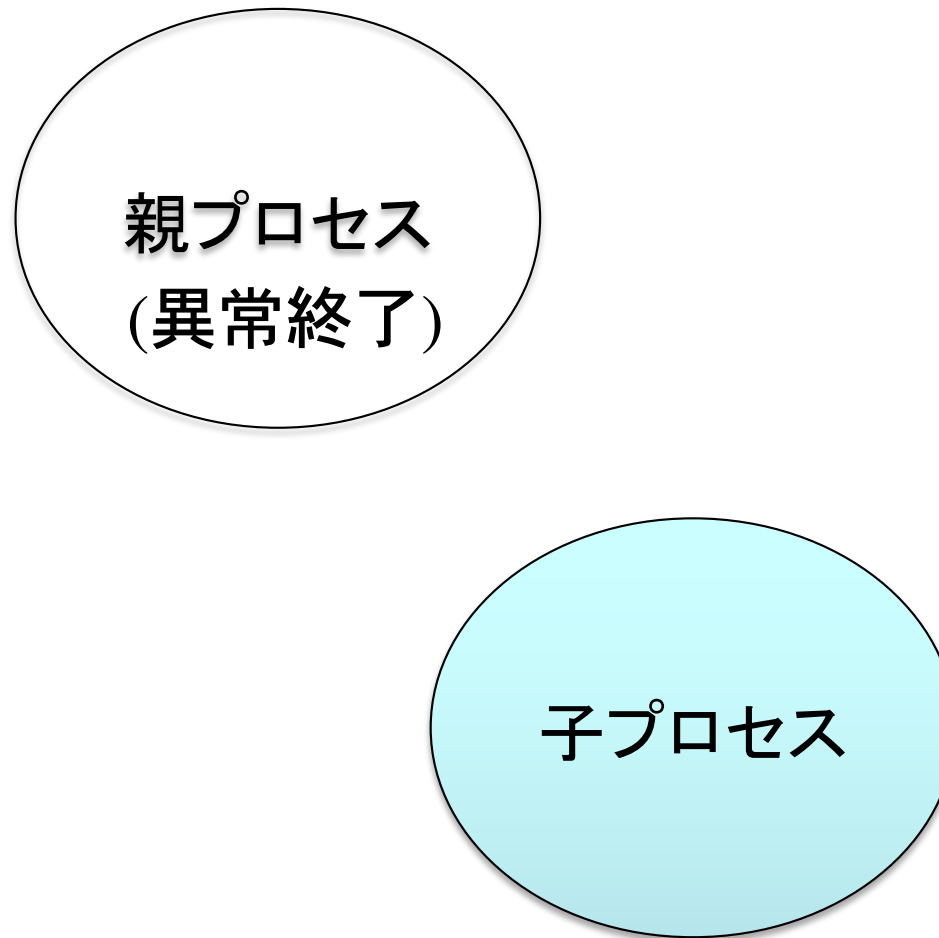
終了状態の受け渡し(正常)



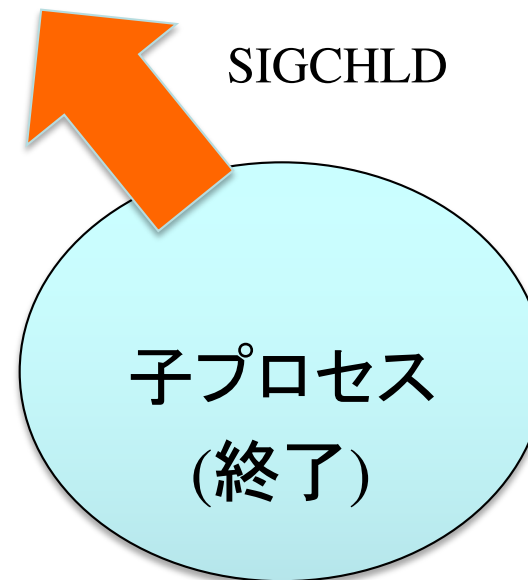
終了状態の受け渡し(異常)




終了状態の受け渡し(異常)



終了状態の受け渡し (異常)



終了状態の受け渡し (異常)



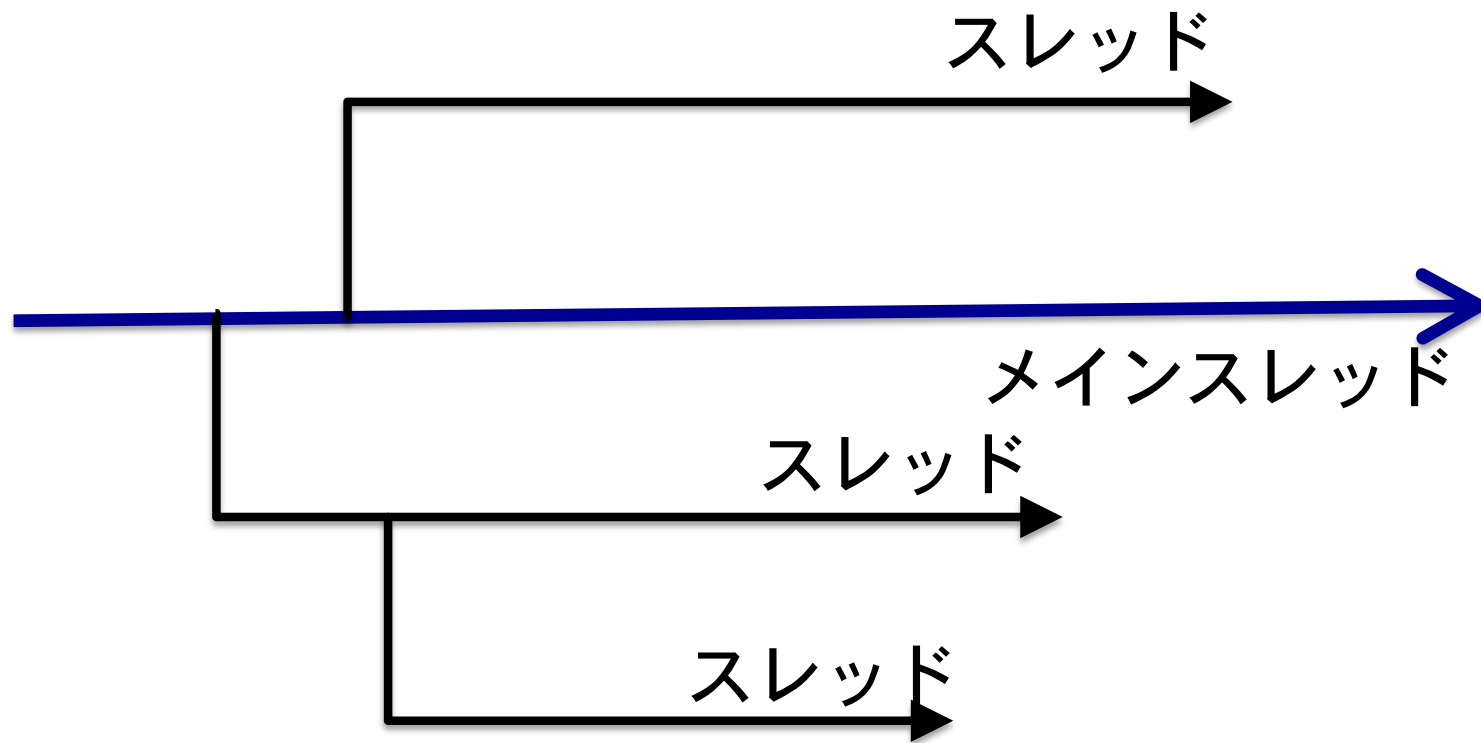
子プロセス
(ゾンビ)

THREAD

Thread を利用したサーバプログラミング

- Listen ソケットを生成
- accept() したらソケットを引数として thread を生成
 - クライアントとの通信は生成された thread が担当
- メインスレッドは再び accept() で待機

Thread の概念



多段に派生させることが可能

pthread_create()

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

```
int pthread_create(pthread_t *thread,  
                   const pthread_attr_t *attr,  
                   void *(*start_routine) (void *),  
                   void *arg);
```

pthread_t: スレッドID

pthread_attr_t: スケジューリング指定

返り値: スレッドの生成に成功 0
 スレッドの生成に失敗 0以外

gcc では、コンパイル時に `-lpthread` の指定が必要

pthread_detach()

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

```
int pthread_detach(pthread_t thread);
```

スレッドに割り当てられているリソースが終了時に回収可能であることをシステムに知らせる

返り値: 成功 0
 失敗 0以外

pthread_join()

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

```
int pthread_join(pthread_t thread, void **value);
```

メインスレッドで生成したスレッドの終了を待ち
終了状態の通知を受ける

返り値: 成功 0
 失敗 0以外

thread の使い方 (1)

```
acc = accept(soc, (struct sockaddr *)&from, &len);
```

```
if (acc == -1) {
```

```
    if (errno != EINTR) {
```

```
        perror("accept");
```

```
    }
```

```
} else {
```

```
    /* スレッド生成 */
```

```
    if (pthread_create(&thread_id, NULL, send_recv_thread, (void *) acc) != 0) {
```

```
        perror("pthread_create");
```

```
    }
```

```
}
```

thread の使い方 (2)

```
void *send_recv_thread (void *arg) {  
    pthread_detach (pthread_self());  
  
    acc = (int) arg;  
  
    for(;;) {  
        クライアントとの recv/send 処理などを実施  
    }  
    close(acc);  
    pthread_exit((void *) 0);  
}
```

SELECT

select() を用いたサーバプログラミング

- 非同期多重入出力
 - ファイルディスクリプタ (FD) の状態について
 - 読み書きができるのか？ 待たされなければいけないのか？
 - 入出力の状態は非同期
 - そのプロセスだけで決めることができない
 - 用意の出来た FD
 - 読み込み準備の整った FD からだけ読み込みたい
 - 書き込み準備の整った FD にだけ書き込みたい
- select() システムコール
 - 複数の FD が非同期に有効となる場合、どの FD が有効かを監視するシステムコール
 - FD の集合を渡して、読み出し／書き込み可能な FD を教えてもらう
 - 一定のタイムアウトも指定できる

select() を用いたサーバプログラミング

- select() は非常に重いシステムコールと言われる
- 最近では epoll() / kqueue() を利用することが多い
 - C10K 問題（クライアント1万台問題）
- プログラムの互換性という点ではまだ select()
- 一つのプロセスの中で複数のソケットを制御可能
 - プロセスが分離されない
 - データのやり取りが容易
- ソケットディスクリプタ（ファイルディスクリプタ）を監視

select() の基本的な使い方

1. FD_ZERO で初期設定
2. 監視したいファイルディスクリプタ(ソケット)を FD_SET にて指定
3. 監視ループ開始
 1. select() にて監視結果取得
 2. FD_ISSET で監視結果を評価
 3. メッセージが届いていたファイルディスクリプタ(ソケット)に応じた処理を行う
4. 監視ループ終了

select() 関数

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

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

```
int select( int nfd,
```

```
    fd_set *readfds,
```

読み込み可能な fds

```
    fd_set *writefds,
```

書き込み可能な fds

```
    fd_set *exceptfds,
```

```
    struct timeval *timeout);
```

返り値: 読み書きの準備ができているディスクリプタの個数
タイムアウトなら0, エラーの場合は -1

fd_set 構造体

```
typedef struct
```

```
{
```

```
    /* XPG4.2 requires this member name.  Otherwise avoid the name  
    from the global namespace.  */
```

```
#ifdef _USE_XOPEN
```

```
    _fd_mask fds_bits[_FD_SETSIZE / _NFDBITS];
```

```
# define __FDS_BITS(set) ((set)->fds_bits)
```

```
#else
```

```
    _fd_mask _fds_bits[_FD_SETSIZE / _NFDBITS];
```

```
# define __FDS_BITS(set) ((set)->_fds_bits)
```

```
#endif
```

```
    } fd_set;
```

標準入出力

- 標準入力 (Standard Input): キーボード
 - ファイルディスクリプタ: 0
- 標準出力 (Standard Output): ディスプレイ
 - ファイルディスクリプタ: 1
- 標準エラー (Standard Error): ディスプレイ
 - ファイルディスクリプタ: 2

- 標準入出力のファイルディスクリプタ番号

```
printf("STDIN = %d¥n", fileno(stdin));
```

```
printf("STDOUT = %d¥n", fileno(stdout));
```

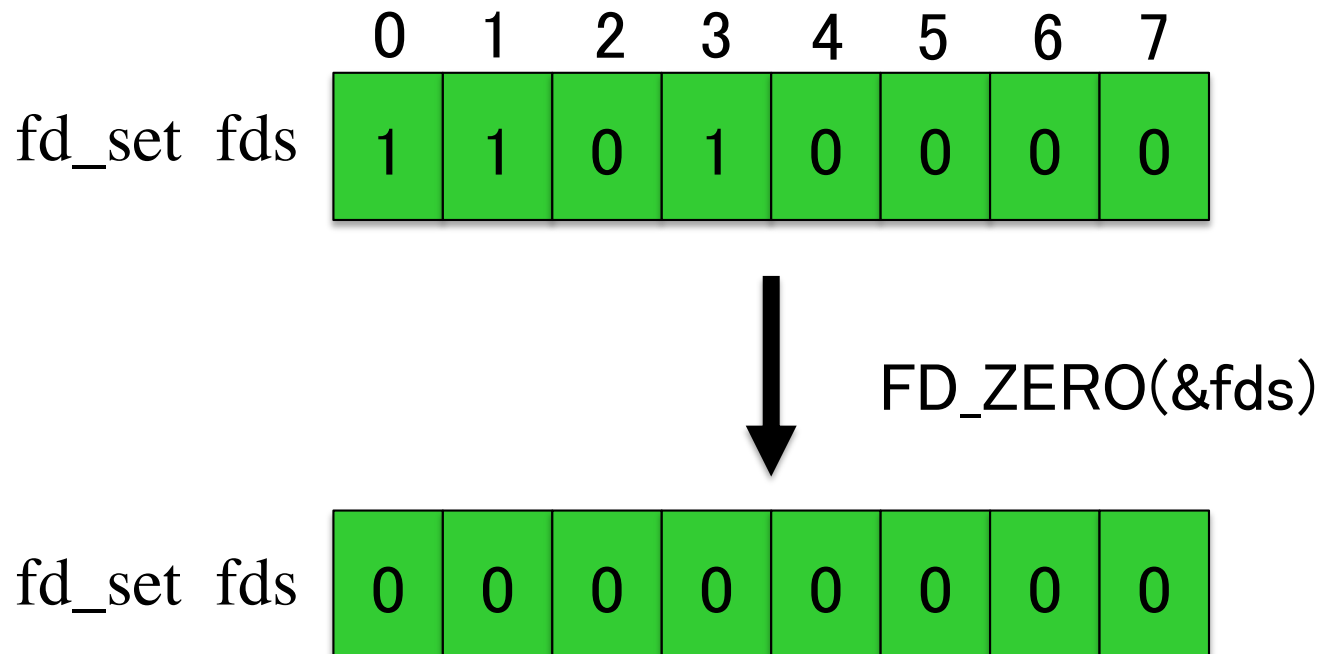
```
printf("STDERR = %d¥n", fileno(stderr));
```

fd_set を操作するためのマクロ

```
void FD_ZERO(fd_set *fds);  
void FD_SET(int fd, fd_set *fds);  
void FD_CLR(int fd, fd_set *fds);  
int FD_ISSET(int fd, fd_set *fds);
```

FD_ZERO

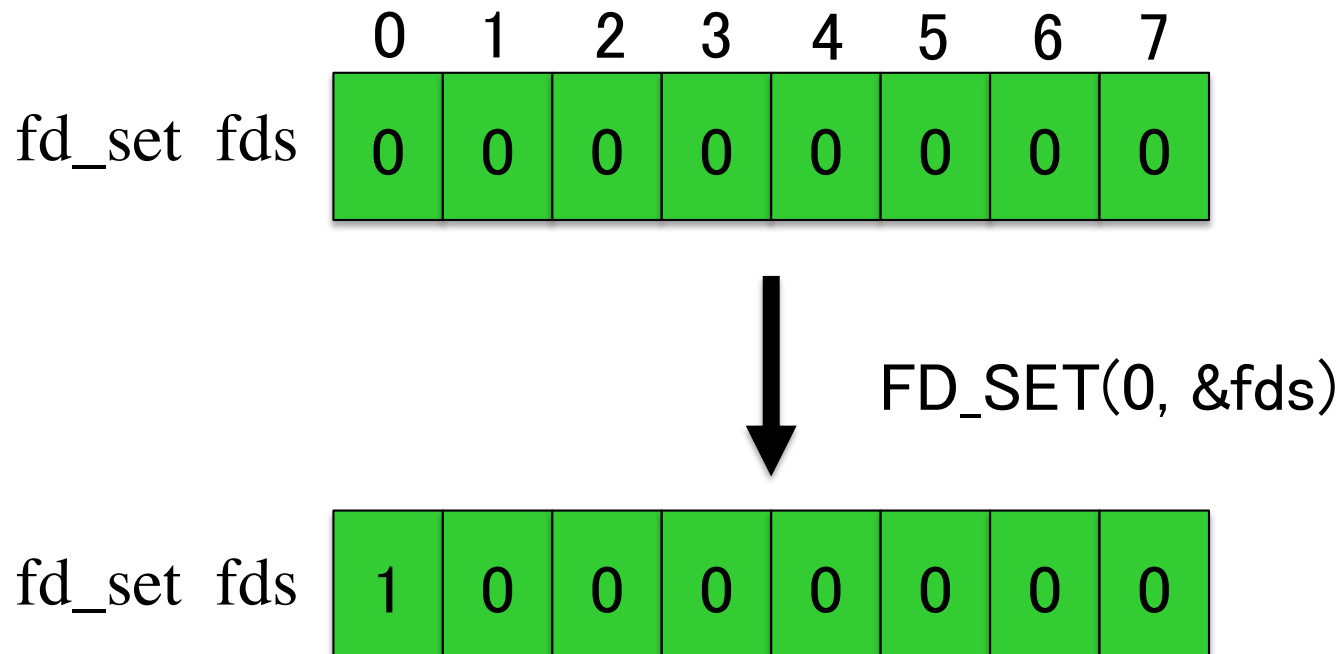
- fds 中の全ての bit をクリアする
 - fd_set 型変数の初期化に用いる
- FD_SET などを行う前に必ず行う



FD_SET

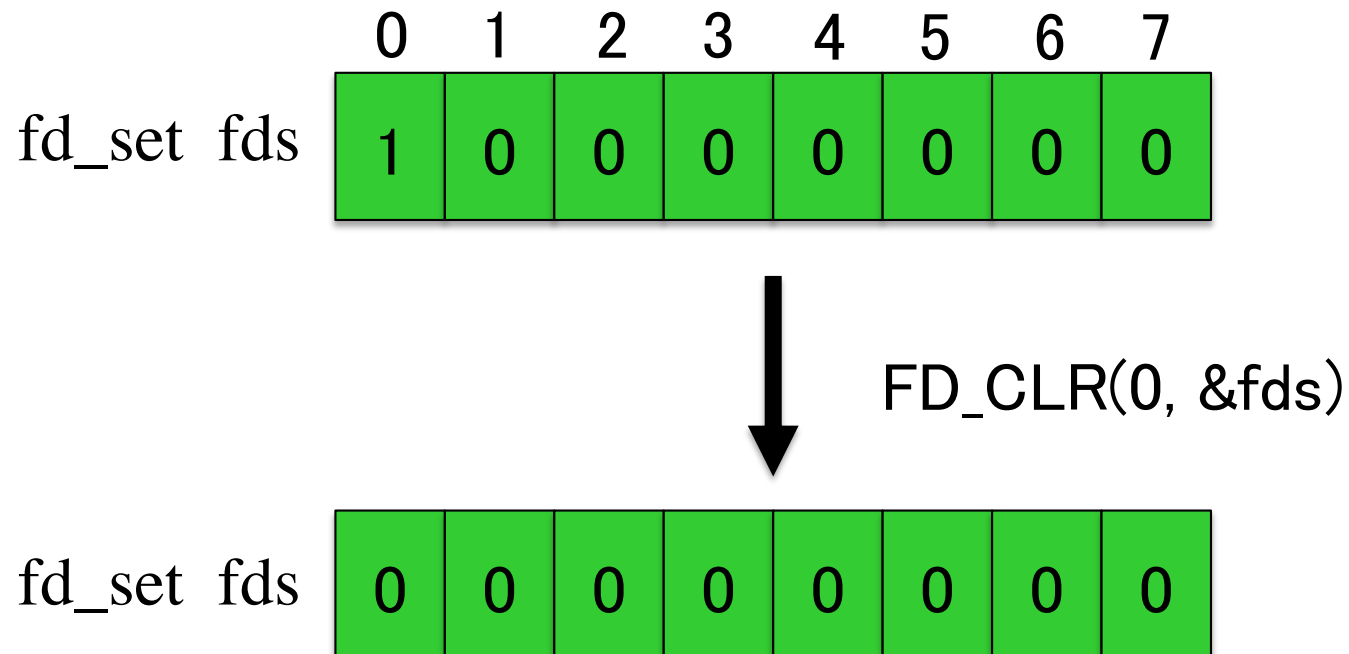
- fds 中の fd の bit をセットする
 - select() で監視すべきファイルディスクリプタを指定する

標準入力を select の監視対象に入れる場合



FD_CLR

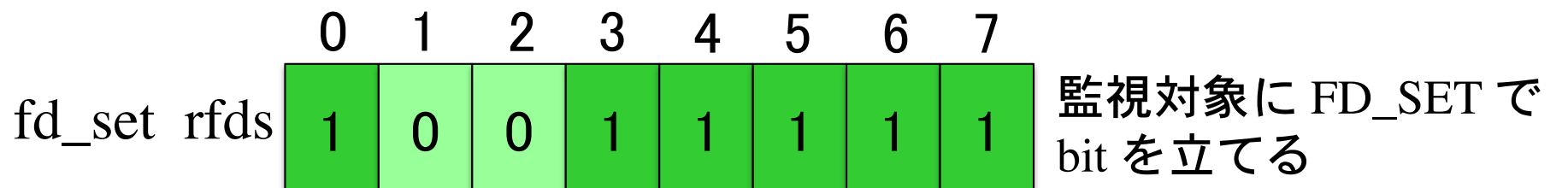
- fds 中の fd の bit をクリアする → FD_SET の反対



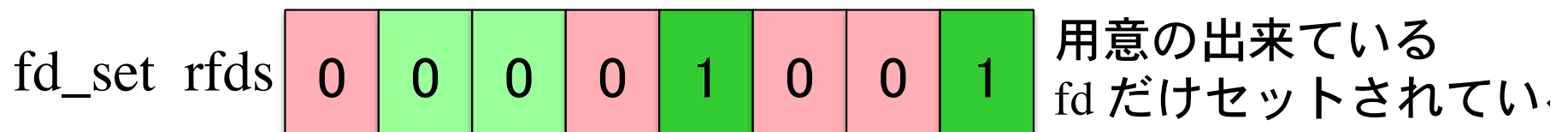
select()

- ある fds が読み込み可能か監視する場合

`select (maxfdpl, &rfd, NULL, NULL, NULL)`

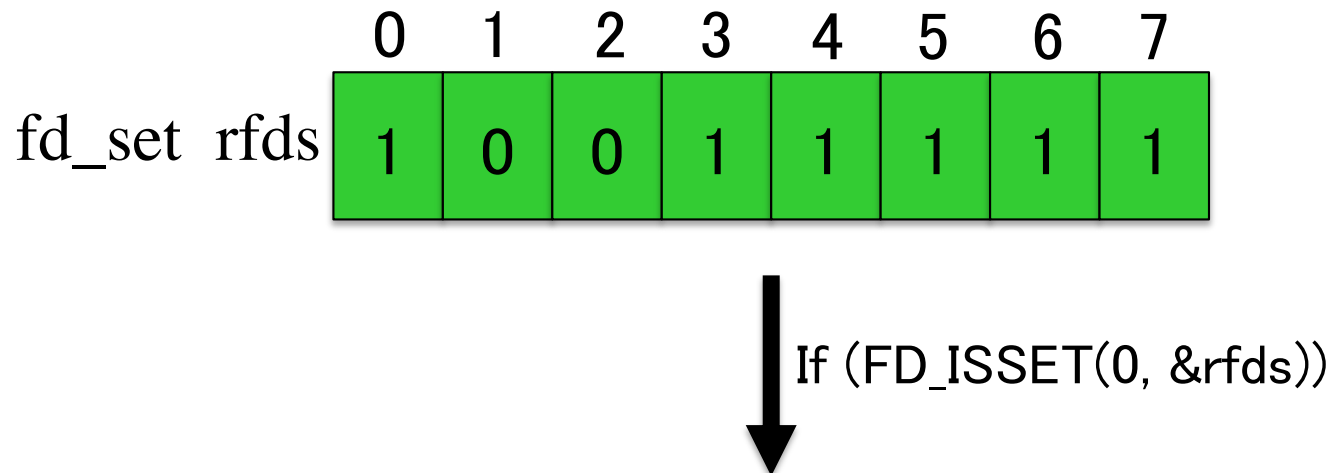


`select(maxfdpl, &rfd, NULL, NULL, NULL)`



FD_ISSET

- fds 中の fd の bit がセットされているか検査する



標準入力を読み込み可能（キー入力有り）

```
if (FD_ISSET(0, &rfd)) {  
    /* 標準入力を読み込み可能 */  
    read (0, &buf, sizeof(buf));  
}
```

select() を使ったサーバ (1)

```
int maxfd;
```

```
fd_set  fds, readfds;
```

```
struct sockaddr_in addr1, addr2;
```

```
sock1 = socket (AF_INET, SOCK_DGRAM, 0);
```

```
sock2 = socket (AF_INET, SOCK_DGRAM, 0);
```

```
addr1.sin_port = htons(10000);
```

```
addr2.sin_port = htons(20000);
```

```
bind(sock1, (struct sockaddr *)&addr1, sizeof(addr1));
```

```
bind(sock2, (struct sockaddr *)&addr2, sizeof(addr2));
```

```
FD_ZERO(&readfds);
```

```
FD_SET(sock1, &readfds);
```

```
FD_SET(sock2, &readfds);
```

select() を使ったサーバ (2)

```
struct timeval waitval;
```

```
waitval.tv_sec = 2;
```

```
waitval.tv_usec = 500;
```

```
while(1) {
```

```
    memcpy(&fds, &readfds, sizeof(readfds));
```

```
    n = select(FD_SETSIZE, &fds, NULL, NULL, &waitval);
```

```
    if (FD_ISSET(sock1, &fds)) {
```

```
        受信処理
```

```
    }
```

```
    if (FD_ISSET(sock2, &fds)) {
```

```
        受信処理
```

```
    }
```

```
}
```

select() を用いたサーバの作り方

- サーバでリスニングソケットを作成
- リスニングソケットを select() の監視に入れる
- リスニングソケットが読み出し可能 (read OK) になったら、accept() を実行
- 新しく作られた accept ソケットも select() の監視に追加
- accept ソケットが読み出し可能になったら、クライアントとの通信処理を行う
- 以上の動作を繰り返しながら通信処理を行う
 - サーバのリスニングソケット
 - クライアントの accept ソケット全て

select() を使った TCP サーバの概要

```
s = socket (AF_INET, SOCK_STREAM, 0);  
bind(s, (struct sockaddr *)&server_addr, sizeof(server_addr));  
Listen(s, 5);  
FD_SET(s, &readfd);
```

```
while(1) {  
    memcpy(&fds, &readfds, sizeof(fd_set));  
    n = select(maxfd + 1, &fds, NULL, NULL, &waitval);  
    if (FD_ISSET(s, &fds)) {  
        クライアントが accept してきた時の accept 処理  
        accept してクライアント用のソケット生成  
    }  
    for (i = 0; i < maxfd + 1; i++) {  
        if (FD_ISSET(c_sock[i], &fds)) {  
            各クライアントからデータが届いていた場合の処理  
        }  
    }  
}
```