

ネットワークコンピューティング 第6回

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

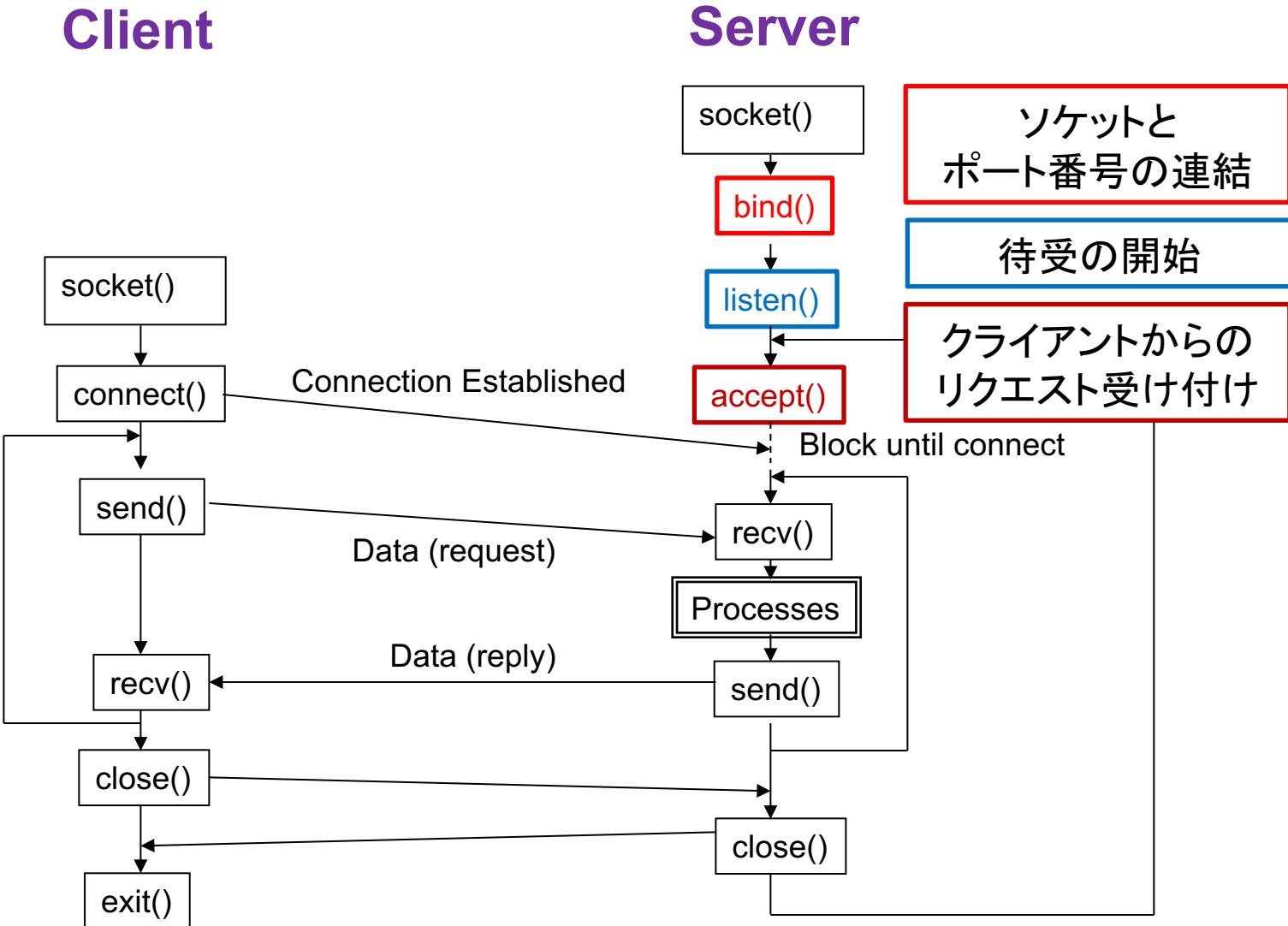
授業に関する情報

- 授業スライド、連絡事項、課題等に関する連絡
 - <https://lecture.sekiya-lab.info/>
- 授業用ログインノード
 - login1.sekiya-lab.info
 - login2.sekiya-lab.info

本日の流れ

- 質問への回答
 - Listen の役割
- 演習環境について
 - なぜ前回中山先生はサーバを上げるのに失敗したのか
- 前回の復習
 - 並行サーバの作り方
- 本日のテーマ
 - 名前解決
 - IPv4/IPv6 デュアルスタックなサーバ

Listen の役割



サーバサンプルプログラム (抜粋 - エラー処理なし)

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

bind(sock, (struct sockaddr *)&addr, sizeof(addr));
listen(sock, 5);

while (1) {
    len = sizeof(client);
    cli = accept(sock, (struct sockaddr *)&client, &len);
    write(cli, "HELLO\n", 7);
    close(cli);
}
```

AWS (Amazon Web Service)

- 世界最大のクラウドカンパニー
 - 計算資源とかストレージとかデータベースとか
 - 時間単位の課金で使うことができます
- ここで lecture, login1, login2 を立ち上げています
 - EC2 というサービス

The screenshot shows the AWS EC2 Instances page. On the left, there's a sidebar with navigation links like EC2 Dashboard, Events, Tags, Reports, Limits, Instances, Launch Templates, Spot Requests, Reserved Instances, and Dedicated Hosts. The 'Instances' link is currently selected. The main area has a search bar at the top. Below it is a table with columns: Name, インスタンス ID, インスタンス状態, アベイラビリティ, インスタンスの状態, ステータスチェック, アラームのステータス, パブリック DNS (IPv4), IPv4 パブリック IP, IPv6 IP, and キー名. There are three rows in the table, each representing an instance: 'login2' (i-0042d7d3e206e792f), 'login1' (i-02501f5a0826cbd88), and 'Web' (i-0eff842378382a22). All instances are listed as 'running'.

Name	インスタンス ID	インスタンス状態	アベイラビリティ	インスタンスの状態	ステータスチェック	アラームのステータス	パブリック DNS (IPv4)	IPv4 パブリック IP	IPv6 IP	キー名
login2	i-0042d7d3e206e792f	t2.micro	ap-northeast-1a	running	2/2 のチェックに合格...	なし	ec2-54-249-36-78.ap...	54.249.36.78	2406:da14:941:3c...	LoginNode1-K
login1	i-02501f5a0826cbd88	t2.micro	ap-northeast-1a	running	2/2 のチェックに合格...	なし	ec2-13-115-209-199.ap...	13.115.209.199	2406:da14:941:3c...	LoginNode1-K
Web	i-0eff842378382a22	t2.micro	ap-northeast-1a	running	2/2 のチェックに合格...	なし	ec2-13-231-128-203.ap...	13.231.128.203	2406:da14:941:3c...	LoginNode1-K

AWS のサービス

コンピューティング	EC2 Lightsail ▾ Elastic Container Service Lambda Batch Elastic Beanstalk	管理ツール	CloudWatch AWS Auto Scaling CloudFormation CloudTrail Config OpsWorks Service Catalog Systems Manager Trusted Advisor Managed Services	分析	Athena EMR CloudSearch Elasticsearch Service Kinesis QuickSight ▾ Data Pipeline AWS Glue	カスタマーエンゲージメント	Amazon Connect Pinpoint Simple Email Service
ストレージ	S3 EFS Glacier Storage Gateway	メディアサービス	Elastic Transcoder Kinesis Video Streams MediaConvert MediaLive MediaPackage MediaStore MediaTailor	セキュリティ、アイデンティティ、コンプライアンス	IAM Cognito Secrets Manager GuardDuty Inspector Amazon Macie ▾ AWS Single Sign-On Certificate Manager CloudHSM Directory Service WAF & Shield Artifact	ビジネスの生産性	Alexa for Business Amazon Chime ▾ WorkDocs WorkMail
データベース	RDS DynamoDB ElastiCache Amazon Redshift	機械学習	Amazon SageMaker Amazon Comprehend AWS DeepLens Amazon Lex Machine Learning Amazon Polly Rekognition Amazon Transcribe Amazon Translate	モバイルサービス	Mobile Hub AWS AppSync Device Farm Mobile Analytics	デスクトップとアプリケーションのストリーミング	WorkSpaces AppStream 2.0
移行	AWS Migration Hub Application Discovery Service Database Migration Service Server Migration Service Snowball	ゲーム開発	Amazon GameLift	IoT	IoT Core IoT 1-Click IoT Device Management IoT Analytics Greengrass Amazon FreeRTOS	拡張現実 (AR) とバーチャルリアリティ (VR)	Amazon Sumerian
ネットワーキング & コンテンツ配信	VPC CloudFront Route 53 API Gateway Direct Connect	開発者用ツール	CodeStar CodeCommit CodeBuild CodeDeploy CodePipeline Cloud9 X-Ray	アプリケーション統合	Step Functions Amazon MQ Simple Notification Service Simple Queue Service SWF		

VM (インスタンス) の設定

インスタンス: i-0042d7d3e206e792f (login2) パブリック DNS: ec2-54-249-36-78.ap-northeast-1.compute.amazonaws.com

説明 ステータスチェック モニタリング タグ

インスタンス ID	i-0042d7d3e206e792f
インスタンスの状態	running
インスタンスタイプ	t2.micro
Elastic IP	アベイラビリティゾーン: ap-northeast-1a
セキュリティグループ	launch-wizard-1, ルールの表示
予定されているアドレス	予定されていません。アドレスはめりません
AMI ID	ubuntu/images/hvm-ssd/ubuntu-xenial-16.04-amd64-server-20180306 (ami-0d74386b)
プラットフォーム	-
IAM ロール	-
キーペア名	LoginNode1-Key
EBS 最適化	False
ルートデバイスタイプ	ebs
ルートデバイス	/dev/sda1
ブロックデバイス	/dev/sda1
Elastic GPU	-
Elastic GPU タイプ	-
Elastic GPU ステータス	-

パブリック DNS (IPv4)	ec2-54-249-36-78.ap-northeast-1.compute.amazonaws.com
IPv4 パブリック IP	54.249.36.78
IPv6 IP	2406:da14:941:3c03::20
プライベート DNS	ip-172-31-40-245.ap-northeast-1.compute.internal
プライベート IP	172.31.40.245
セカンダリプライベート IP	
VPC ID	vpc-15bb2972
サブネット ID	subnet-9a7658d3
ネットワークインターフェイス	eth0
送信元/送信先チェック	True
T2 Unlimited	無効
所有者	833647524345
起動時刻	2018年4月15日 1:03:34 UTC+9 (926 時間)
削除保護	False
ライフサイクル	normal
モニタリング	基本
アラームステータス	なし
カーネル ID	-
RAM ディスク ID	-
配置グループ	-
仮想化	hvm
予約	r-09cf8061d8d48b8b0
AMI 作成インデックス	0
テナント	default
ホスト ID	-
アフィニティ	-
状態遷移の理由	-
状態遷移の理由メッセージ	-

VM (インスタンス) の設定

インスタンス: i-0042d7d3e206e792f (login2) パブリック DNS: ec2-54-249-36-78.ap-northeast-1.compute.amazonaws.com

説明 ステータスチェック モニタリング タグ

インスタンス ID	i-0042d7d3e206e792f
インスタンスの状態	running
インスタンスタイプ	t2.micro
Elastic IP	アベイラビリティゾーン: ap-northeast-1a
セキュリティグループ	launch-wizard-1, ルールの表示
予定されているアドレス	予定されていません。アドレスはありません
AMI ID	ubuntu/images/hvm-ssd/ubuntu-xenial-16.04-amd64-server-20180306 (ami-0d74386b)
プラットフォーム	-
IAM ロール	-
キーペア名	LoginNode1-Key
EBS 最適化	False
ルートデバイスタイプ	ebs
ルートデバイス	/dev/sda1
ブロックデバイス	/dev/sda1
Elastic GPU	-

パブリック DNS (IPv4)	ec2-54-249-36-78.ap-northeast-1.compute.amazonaws.com
IPv4 パブリック IP	54.249.36.78
IPv6 IP	2406:da14:941:3c03::20
プライベート DNS	ip-172-31-40-245.ap-northeast-1.compute.internal
プライベート IP	172.31.40.245
セカンダリプライベート IP	
VPC ID	vpc-15bb2972
サブネット ID	subnet-9a7658d3
ネットワークインターフェイス	eth0
送信元/送信先チェック	True
T2 Unlimited	無効
所有者	833647524345
起動時刻	2018年4月15日 1:03:34 UTC+9 (926 時間)
削除保護	False
ライフサイクル	normal
モニタリング	基本
アラームステータス	なし

セキュリティグループ: sg-02e3eec021bfbed3f



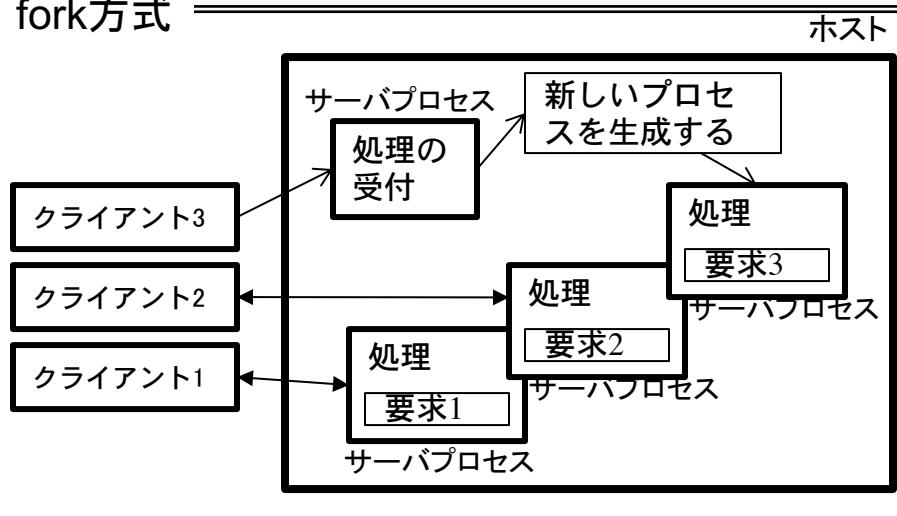
説明 インバウンド アウトバウンド タグ

編集

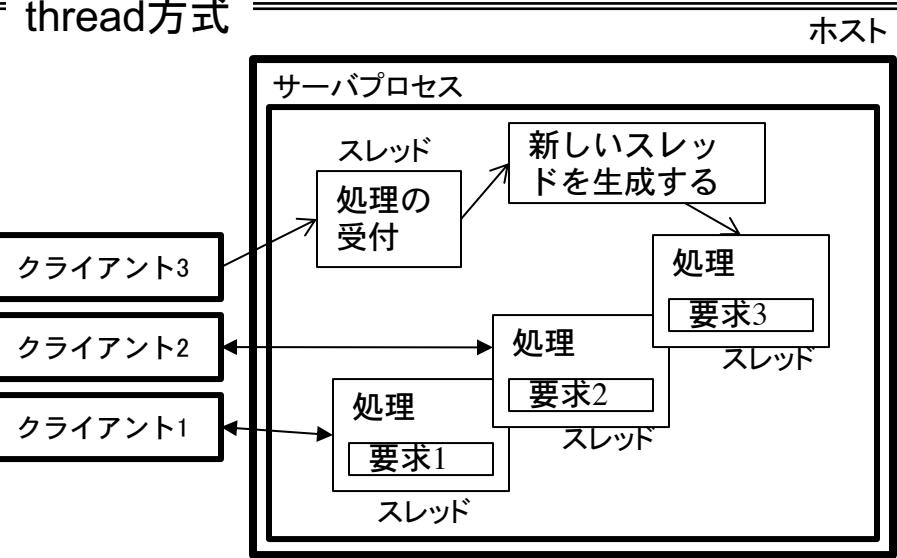
タイプ	プロトコル	ポート範囲	ソース	説明
すべてのトラフィック	すべて	すべて	0.0.0.0/0	
すべてのトラフィック	すべて	すべて	::/0	
カスタム ICMP ルール - IPv6	すべて	該当なし	::/0	
すべての ICMP - IPv4	すべて	該当なし	0.0.0.0/0	

並行サーバの構成方法

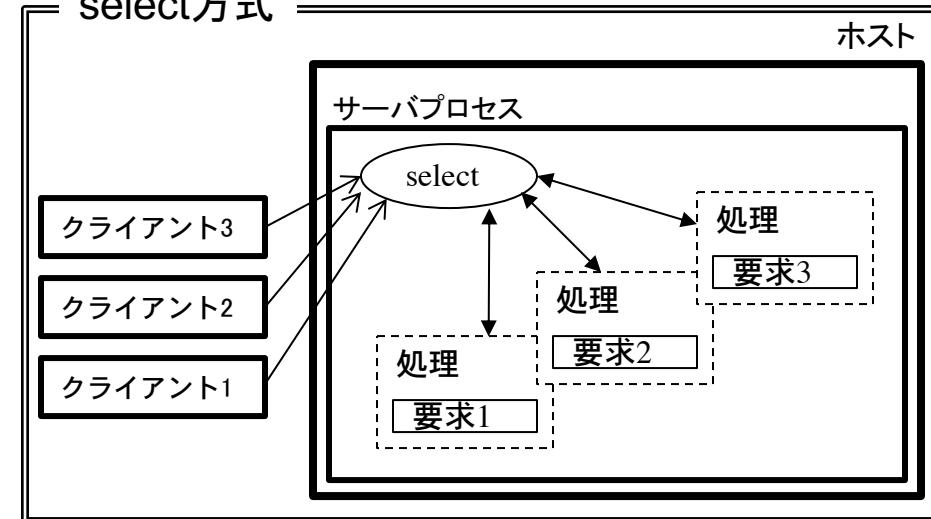
fork方式



thread方式



select方式



利点・欠点

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

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

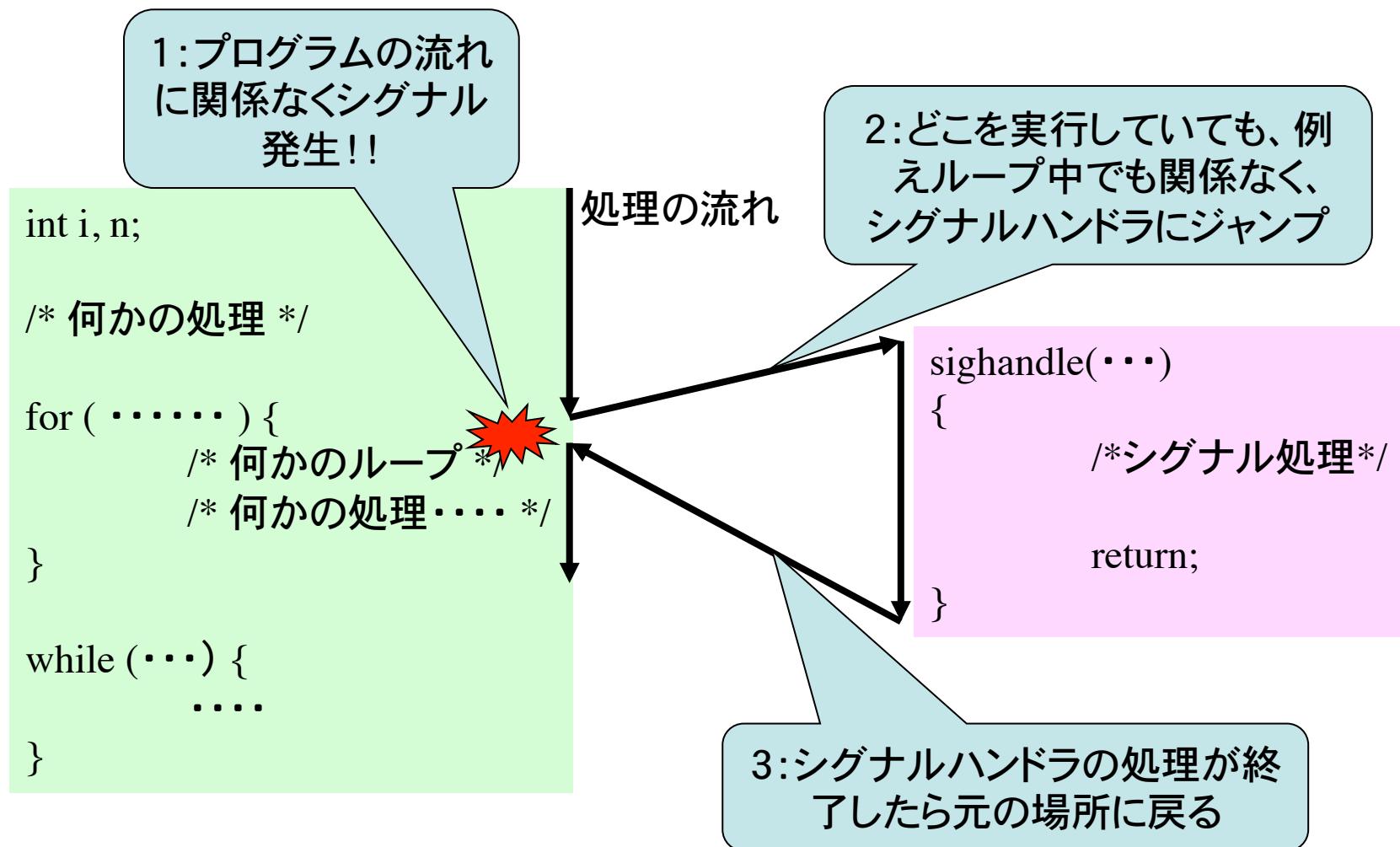
- プロセスの確認方法

```
% ps -l
```

fork() を用いたサーバプログラミング

- 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、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



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);
    }
}
```

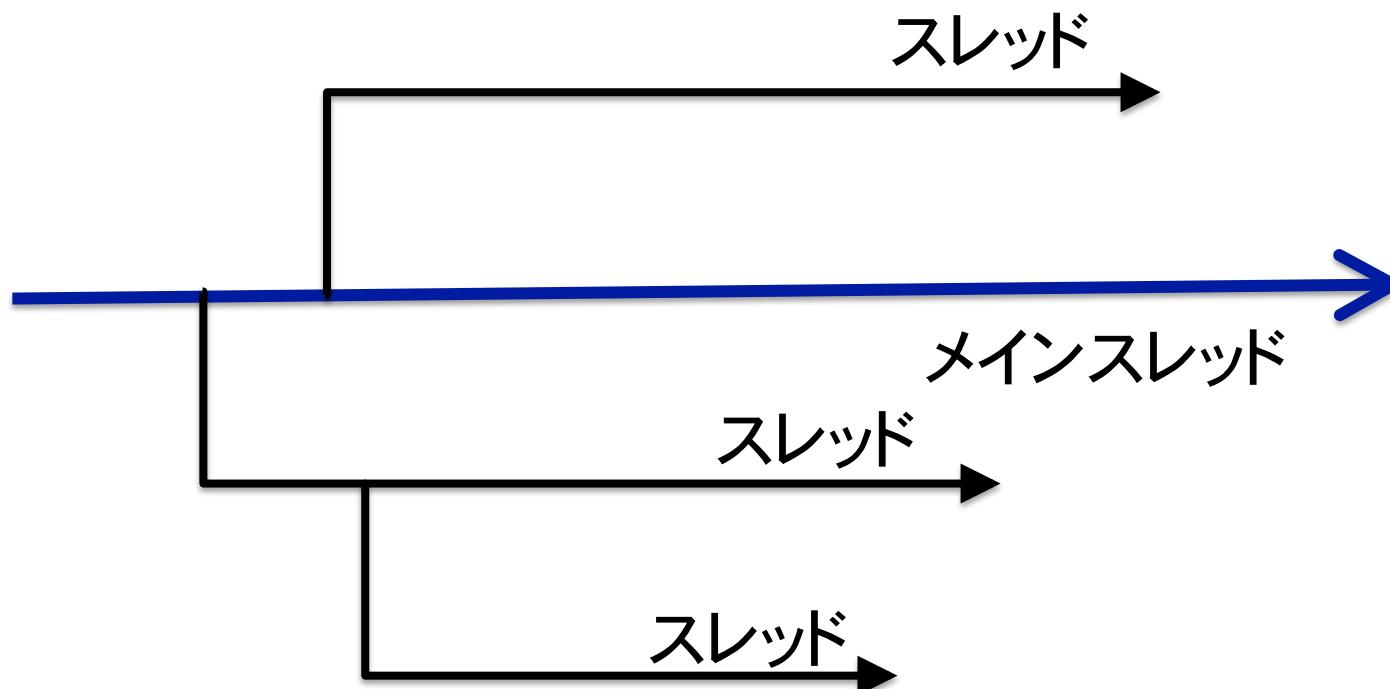
- `pthread_create()`について

```
#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

Thread の概念



多段に派生させることが可能

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

select() を利用し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() を利用したサーバプログラミング (cont.)

- select() は非常に重いシステムコールと言われる
- 最近は 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スクpri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スクpriタ（ソケット）に応じた処理を行う
4. 監視ループ終了

Python の Thread

```
from threading import Thread
import time

class SampleThread(Thread):

    def __init__(self, name, interval):
        super(SampleThread, self).__init__()
        self.daemon = True
        self.count = 0
        self.interval = interval
        self.name = name

    def run(self):
        for _ in range(5):
            time.sleep(self.interval)
            self.count += 1
            print '%s: %d' % (self.name, self.count, )

if __name__ == '__main__':
    t1 = SampleThread('t1', 1)
    t2 = SampleThread('t2', 2)
    t1.start()
    t2.start()
    t1.join()
    t2.join()
```

Ruby の Thread

```
list = ["A", "B", "C", "D"]
io    = File.open("result.log", "w")
threads = []

list.each do |name|
  threads << Thread.fork(name) do |name|
    for count in 10000.times
      io.puts name
    end
    puts "Thread#{name} completed!";
  end
end

threads.each do |thread|
  thread.join
end
```

Go Lang の goroutine

```
func say(s string, ch chan int)
    fmt.Println(s)
    ch <- 1
}

func main() {
    var ch chan int
    ch = make(chan int)
    go say("Hello world", ch)
    <-ch
}
```

名前解決

ホスト名とIPアドレス

- インターネットでは、Domain Name System(DNS)を使ってホスト名とIPアドレスの相互変換をしている

www.u-tokyo.ac.jp

ホスト名

133.11.114.194

IPアドレス

```
% nslookup www.u-tokyo.ac.jp  
Server:      45.0.0.100  
Address:     45.0.0.100#53
```

```
Non-authoritative answer:  
Name:  www.u-tokyo.ac.jp  
Address: 133.11.114.194
```

gethostbyname()

- ホスト名(文字列)を関数に渡すと、hostent 構造体を返す

関数

```
struct hostent * gethostbyname(const char *name);
```

hostent構造体

```
struct hostent {  
    char          *h_name;           /* ホストの正式名称 */  
    char          **h_aliases;        /* 別名のリスト */  
    int           h_addrtype;        /* ホストのアドレスタイプ */  
    int           h_length;          /* アドレスの長さ */  
    char          **h_addr_list;      /* アドレスの入った配列へのポインタ */  
#define h_addr  h_addr_list[0] /* address, for backward compatibility */  
};
```

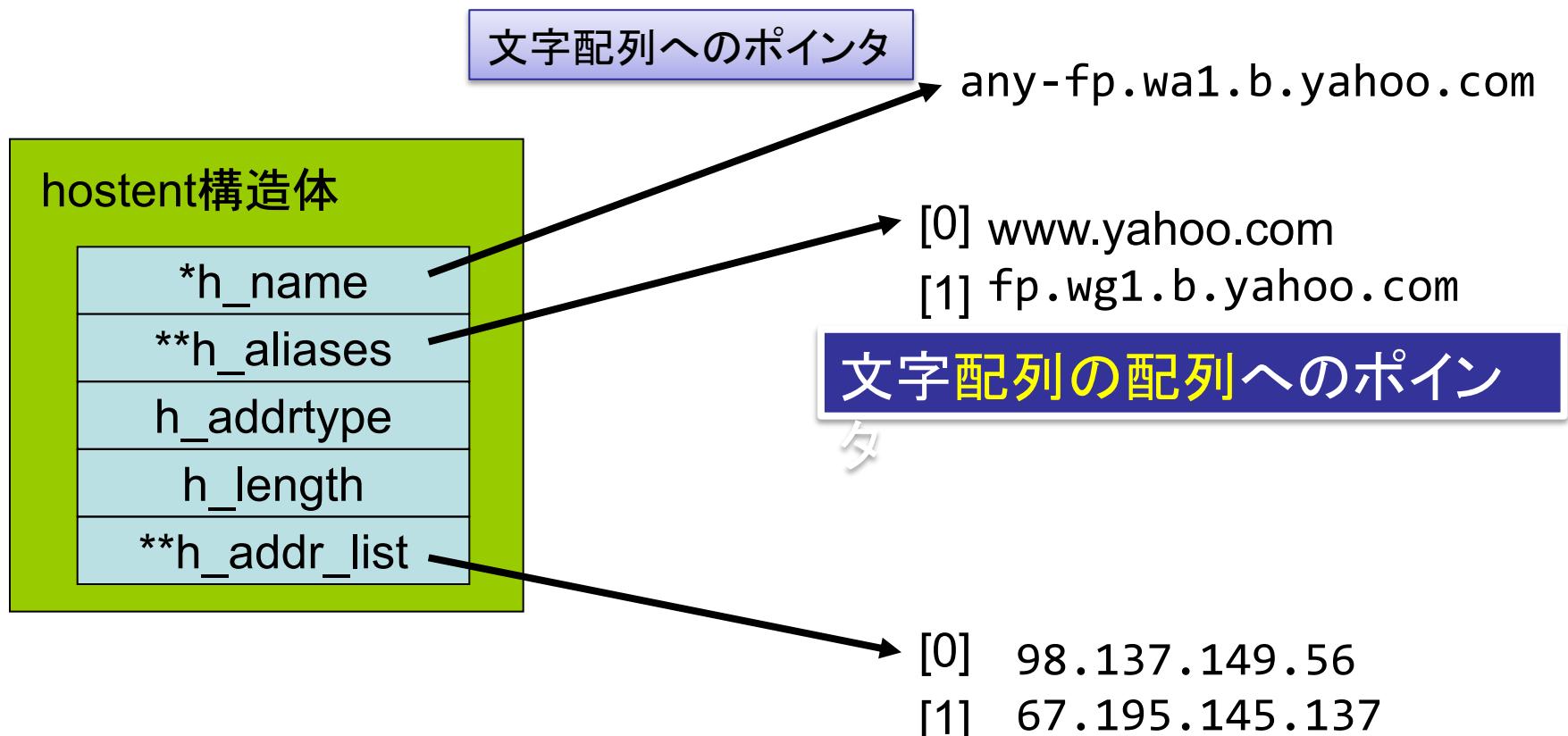
hostent構造体

- 変換した情報が入った構造体
- IPアドレスの配列を持つ
- なぜ配列か?
 - 一つの名前に複数のアドレスが付いている場合がある

```
sekiya@LECTURE-CLIENT:~$ host www.yahoo.com
www.yahoo.com is an alias for fp.wg1.b.yahoo.com.
fp.wg1.b.yahoo.com is an alias for any-fp.wa1.b.yahoo.com.
any-fp.wa1.b.yahoo.com has address 69.147.125.65
any-fp.wa1.b.yahoo.com has address 67.195.160.76
```

```
sekiya@LECTURE-CLIENT[~]% host www.kddi.com
www.kddi.com has address 61.200.220.160
www.kddi.com has IPv6 address 2001:268:fd02::1
www.kddi.com has IPv6 address 2001:268:fd01::1
```

hostent構造体とアドレスの配列



アドレスの入った文字配列の配列へのポインタ

旧来の名前解決のサンプルコード

```
if( (hp = gethostbyname( argv[1] )) == NULL ){
    perror("gethostbyname");
    return 0;
}

printf("h_name = %s\n", hp->h_name );
printf("h_addrtype = %d\n", hp->h_addrtype );
printf("h_length = %d\n", hp->h_length );

for( i=0; hp->h_aliases[i]; i++ ){
    printf("h_aliases[%d] = %s\n", i, hp->h_aliases[i] );
}

for( i=0; hp->h_addr_list[i]; i++ ){
    bcopy( hp->h_addr_list[i], &sin_addr, hp->h_length );
    printf("h_addr_list[%d] = %s\n", i, (char
*)inet_ntoa(sin_addr) );
}
```

実行例

```
sekiya@LECTURE-CLIENT:~/EXAMPLE$ ./name_res www.yahoo.com
h_name = any-fp.wa1.b.yahoo.com
h_addrtype = 2
h_length = 4
h_aliases[0] = www.yahoo.com
h_aliases[1] = fp.wg1.b.yahoo.com
h_addr_list[0] = 98.137.149.56
h_addr_list[1] = 67.195.145.137
h_addr_list[2] = 67.195.145.138
h_addr_list[3] = 72.30.2.43
```

DNS を用いたプログラミング（再掲）

- ドメイン名 ⇄ IPアドレス変換のライブラリ利用
 - gethostbyname(), gethostbyname2()
 - (基本的な)ドメイン名 → IPアドレス変換関数
 - getaddrinfo()
 - gethostbyname() を汎用的に拡張した関数
 - マルチプロトコル対応

さまざまなプロトコルで動くようにするには？

- 現在のインターネットは(主に)IPv4で構成されているが、将来的にはIPv6に移行することになる
 - IPv4アドレス枯渇 (<https://www.nic.ad.jp/ja/ip/ipv4pool/>)
- IPv4 と IPv6 では単純にアドレスの大きさが違う
 - IPv4: 4 Octets (4バイト)
 - IPv6: 16 Octets (16バイト)
- 通信するソフトウェアでいちいち場合分けをするのは手間がかかる
- 将来的に IPv7～ などが登場したときにどうするか？

プロトコル非依存プログラム

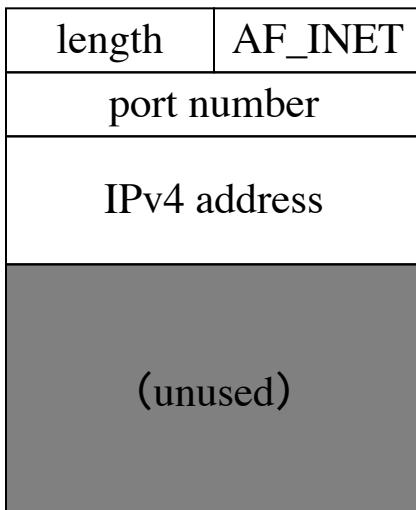
- 汎用的な構造体・関数を使う(ようにする)
- プロトコルに特化した構造体・関数は極力使わない(ようにする)
 - プロトコル依存な関数の例
 - in_addr, in6_addr, sockaddr_in, sockaddr_in6
 - gethostbyname(), gethostbyname2(), inet_ntoa(), inet_ntop()
 - プロトコル非依存(を目指している)関数の例
 - sockaddr, sockaddr_storage
 - getaddrinfo(), getnameinfo()

sockaddr 構造体について

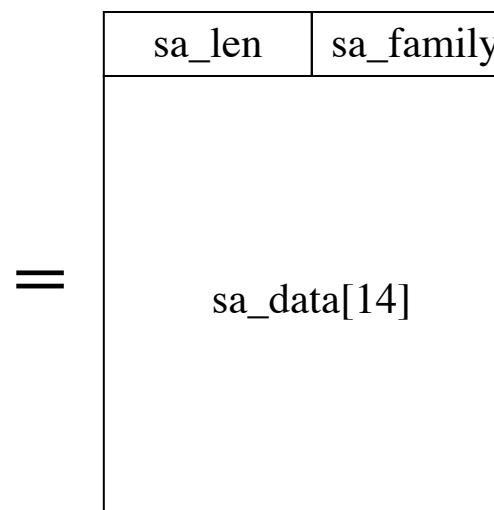
- sockaddr_in 構造体はプロトコル依存(IPv4用)
- 汎用的に扱う場合は sockaddr 構造体が使われるが、
sockaddr 構造体は IPv6 アドレスを入れるにはサイズが不足

```
struct sockaddr {
    unsigned char           sa_len;          /* total length */
    sa_family;              /* address family */
    char                   sa_data[14];       /* address value */
};
```

sockaddr_in()



sockaddr()



= <

IPv6 address

各構造体のサイズ比較

sockaddr()

sa_len	sa_family
sa_data[14]	

16bytes

sockaddr_in()

length	AF_INET
port number	
IPv4 address	
(unused)	

16bytes

sockaddr_in6()

length	AF_INET6
port number	
Flow label	
IPv6 address	
scope id	

28bytes

sockaddr_storage()

ss_len	ss_family
ss_pad1[]	
__ss_align	
__ss_pad2[]	
ss_len	ss_family

128bytes

ポインタによるキャストを利用

- それぞれの型は、ポインタをキャストすることで変換することができる
- 領域には `sockaddr_storage` を用い、関数に渡す場合は `sockaddr` を用いる(ようにする)

例：

```
struct sockaddr_storage ss;           // 領域を確保
struct sockaddr *sa;                 // ポインタを確保

sa = &ss;                           // ポインタに領域を指定

/* アドレス等のネットワーク情報を sa (ss内) に格納 */

bind(sockfd, sa, sa->sa_len);    // ネットワーク関数呼び出し
```

addrinfo 構造体について(再掲)

```
struct addrinfo {  
    int                 ai_flags;  
    int                 ai_family;  
    int                 ai_socktype;  
    int                 ai_protocol;  
    size_t              ai_addrlen;  
    char                *ai_canonname;  
    struct sockaddr    *ai_addr;  
    struct addrinfo    *ai_next;  
}
```

sockaddr 構造体のサイズでは
IPv6 アドレスが収まらないはずでは？

addrinfo 構造体の利用例(再掲)

example:

```
struct addrinfo hints, *res;
memset(&hints, 0, sizeof(hints));
hints.ai_socktype = SOCK_DGRAM;
hints.ai_family = AF_UNSPEC;
getaddrinfo("dns1.nc.u-tokyo.ac.jp", NULL, &hints, &res);
```

*res

ai_flags	
ai_family	AF_INET6
ai_socktype	
ai_protocol	0
ai_addrlen	24
ai_canonname	NULL
ai_addr	2001:200:180::35:1
ai_next	

ai_flags	AF_INET
ai_family	
ai_socktype	
ai_protocol	0
ai_addrlen	16
ai_canonname	NULL
ai_addr	133.11.0.1
ai_next	NULL

getaddrinfo() を使ったサンプルコード

```
struct addrinfo hints, *res0, *res;  
  
memset(&hints, 0, sizeof(hints));  
hints.ai_socktype = SOCK_DGRAM;  
hints.ai_fally = AF_UNSPEC;  
getaddrinfo(hostname, NULL, &hints, &res0);  
  
res = res0;  
while ( res != NULL) {  
    if ( res->ai_family == AF_INET) {  
        printf("ipv4 address : %s\n", inet_ntop(res->ai_family,  
            &((struct sockaddr_in *)(res->ai_addr))->sin_addr, buf, sizeof(buf));  
    } else if ( res->ai_family == AF_INET6) {  
        printf("ipv6 address : %s\n", inet_ntop(res->ai_family,  
            &((struct sockaddr_in6 *)(res->ai_addr))->sin6_addr, buf, sizeof(buf));  
    }  
    res = res->ai_next;  
}  
freeaddrinfo(res0);
```

ポインタによるキャストが
利用されている

freeaddrinfo()

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

```
void freeaddrinfo(struct addrinfo *ai);
```

- getaddrinfo() で動的に割り当てられた領域を解放する
- これを忘れるとなメモリ使用量が増えていってしまう

getaddrinfo() を使ったサンプルコード(再掲)

```
struct addrinfo hints, *res0, *res;

memset(&hints, 0, sizeof(hints));
hints.ai_socktype = SOCK_DGRAM;
hints.ai_fally = AF_UNSPEC;
getaddrinfo(hostname, NULL, &hints, &res0);

res = res0;
while ( res != NULL) {
    if ( res->ai_family == AF_INET) {
        printf("ipv4 address : %s\n", inet_ntop(res->ai_family,
            &((struct sockaddr_in *) (res->ai_addr))->sin_addr, buf, sizeof(buf)));
    } else if ( res->ai_family == AF_INET6) {
        printf("ipv6 address : %s\n", inet_ntop(res->ai_family,
            &((struct sockaddr_in6 *) (res->ai_addr))->sin6_addr, buf, sizeof(buf)));
    }
    res = res->ai_next;
}
freeaddrinfo(res0);
```

前述のプログラムは一部プロトコル依存

- プロトコルに応じて引数の変更が必要
 - sockaddr_in と sockaddr_in6 の構造が異なる
- inet_ntop(int af, const void *src, …)
 - inet_ntop では、in_addr もしくは in6_addr を第2引数に取る
 - addrinfo は sockaddr 構造体を持つ
- 結局、ai_family が AF_INET か AF_INET6 かを区別して sockaddr_in または sockaddr_in6 に cast する必要あり

getaddrinfo() を使ってもプロトコル非依存じゃない
⇒ では、どうすればよい？

getnameinfo()

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

```
int getnameinfo(const struct sockaddr *sa, socklen_t salen, char *host,
                size_t hostlen, char *serv, size_t servlen, int flags);
```

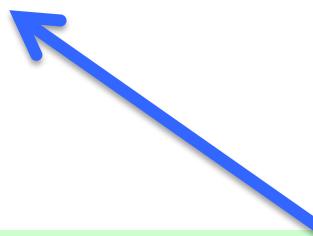
- `getnameinfo()` では `sockaddr` 構造体を引数として渡せるので、`AF_INET` か `AF_INET6` かを区別する必要がない
- 返り値
 - 成功： 0
 - 失敗： それ以外の値

getnameinfo() を用いたサンプルコード

```
struct addrinfo hints, *res0, *res;
```

```
memset(&hints, 0, sizeof(hints));  
hints.ai_socktype = SOCK_DGRAM;  
hints.ai_faily = AF_UNSPEC;  
getaddrinfo(hostname, NULL, &hints, &res0);
```

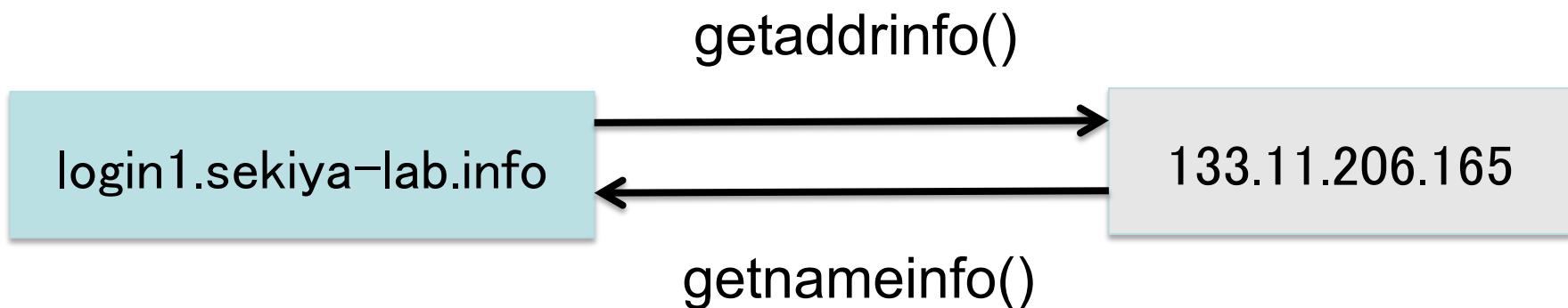
```
res = res0;  
while ( res != NULL ) {  
    getnameinfo(res->ai_addr, res->ai_addrlen, buff, sizeof(buff),  
               NULL, 0, NI_NUMERICHOST);  
    printf("addr: %s\n", buff);  
    res = res->ai_next;  
}  
freeaddrinfo(res0);
```



プロトコル非依存になった

getaddrinfo() / getnameinfo()

- 実際には DNS を用いた以下のような変換ライブラリとして利用される（ことが多い）



IPv4 プログラミングと IPv6 プログラミングの違い

- IPv4 プログラミングでは

- struct hostent *hp;
 - struct **sockaddr_in** servaddr;
 - hp = **gethostbyname**(hostname);

左記の構造体のメンバ情報を

上記の関数を用いて集めて利用

- connect(sockfd, (struct sockaddr *) &servaddr, sizeof(servaddr));

- IPv6 プログラミングでは

- struct hostent6 *hp;
 - struct **sockaddr_in6** servaddr;
 - hp = **gethostbyname2**(hostname, **AF_INET6**);

左記の構造体のメンバ情報を

上記の関数を用いて集めて利用

- connect(sockfd, (struct sockaddr *) &servaddr, sizeof(servaddr));

IPv4 にも IPv6 にも両方対応するためには

- 以下の構造体のメンバ情報を
 - struct addrinfo hints, *ai;
- 以下の関数を用いて集め
 - getaddrinfo(hostname, port, &hints, &ai);
- 以下のように用いればよい
 - socket(ai->ai_family, ai->ai_socktype, ai->ai_protocol);
 - connect(sockfd, ai->ai_addr, ai->ai_addrlen);

では、サーバプログラミングはどうすればよい？

サーバプログラミング(プロトコル依存)

- IPv4 プログラミングでは

```
struct sockaddr_in addr;
sockfd = socket(AF_INET, SOCK_STREAM, 0);
addr.sin_family = AF_INET;
addr.sin_port = htons(80);
addr.sin_addr.s_addr = INADDR_ANY;
bind(sockfd, (struct sockaddr *) &addr, sizeof(addr));
listen(sockfd, 5);
```

- IPv6 プログラミングでは

```
struct sockaddr_in6 addr;
sockfd = socket(AF_INET6, SOCK_STREAM, 0);
addr.sin6_family = AF_INET6;
addr.sin6_port = htons(80);
addr.sin6_addr = in6addr_any;
bind(sockfd, (struct sockaddr *) &addr, sizeof(addr));
listen(sockfd, 5);
```

getaddrinfo() を用いたサーバプログラミング (1)

- IPv4 プログラミングでは

```
hints.ai_family = AF_INET;  
hints.ai_flags = AI_PASSIVE;  
hints.ai_socktype = SOCK_STREAM;  
getaddrinfo(NULL, "80", &hints, &res);  
sockfd = socket(res->ai_family, res->ai_socktype, 0);  
bind(sockfd, res->ai_addr, res->ai_addrlen);  
freeaddrinfo(res);
```

- IPv6 プログラミングでは

```
hints.ai_family = AF_INET6;  
hints.ai_flags = AI_PASSIVE;  
hints.ai_socktype = SOCK_STREAM;  
getaddrinfo(NULL, "80", &hints, &res);  
sockfd = socket(res->ai_family, res->ai_socktype, 0);  
bind(sockfd, res->ai_addr, res->ai_addrlen);  
freeaddrinfo(res);
```

getaddrinfo() を用いたサーバプログラミング (2)

- プロトコル非依存??

```
hints.ai_family = AF_UNSPEC;  
hints.ai_flags = AI_PASSIVE;  
hints.ai_socktype = SOCK_STREAM;  
getaddrinfo(NULL, "80", &hints, &res0);
```

```
n = 0;  
res = res0;  
while (res != NULL) {  
    sock[n] = socket(res->ai_family, res->ai_socktype, 0);  
    bind(sock[n], res->ai_addr, res->ai_addrlen);  
    res = res->ai_next;  
    n++;  
}  
freeaddrinfo(res0);
```

このプログラムには問題点がある。。。

AF_UNSPEC / AI_PASSIVE の実行結果

- INADDR_ANY (0.0.0.0) / in6addr_any (::)
- hints.ai_flags に AI_PASSIVE フラグが指定され、かつ node が NULL の場合、返されるソケットアドレスは、サーバソケットを bind() するのに適したものとなる

```
hints.ai_family = AF_UNSPEC;
hints.ai_flags = AI_PASSIVE;
hints.ai_socktype = SOCK_STREAM;
getaddrinfo(NULL, "80", &hints, &res0);
res = res0;
while (res != NULL) {
    getnameinfo(res->ai_addr, res->ai_addrlen, buff, sizeof(buff), NULL, 0, NI_NUMERICHOST);
    printf("IP address : %s\n", buff);
    res = res->ai_next;
}
freeaddrinfo(res0);
```

```
$ ./getaddr_server
IP address : 0.0.0.0
IP address : ::
```

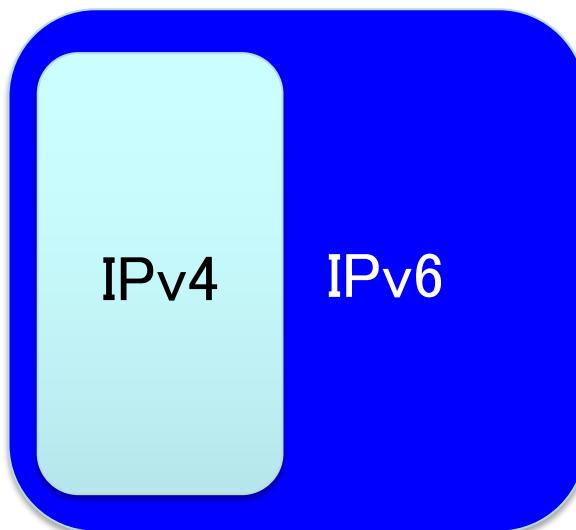
サーバプログラムでの getaddrinfo() (1/2)

- getaddrinfo にて AF_UNSPEC / AI_PASSIVE
 - addrinfo 構造体のリスト (0.0.0.0 / ::) を取得
- hints.ai_family = AF_UNSPEC
 - IPv4, IPv6 の両方のアドレスが返ることになる
- for ループで回して bind を実施
 - 最初に bind が成功したものだけで loop を終了すると
⇒ IPv6 アドレスのみ bind することになる
 - 成功しても失敗しても全ての addrinfo に bind すると
⇒ IPv4, IPv6 のどちらにも bind することになる

サーバプログラムでの getaddrinfo() (2/2)

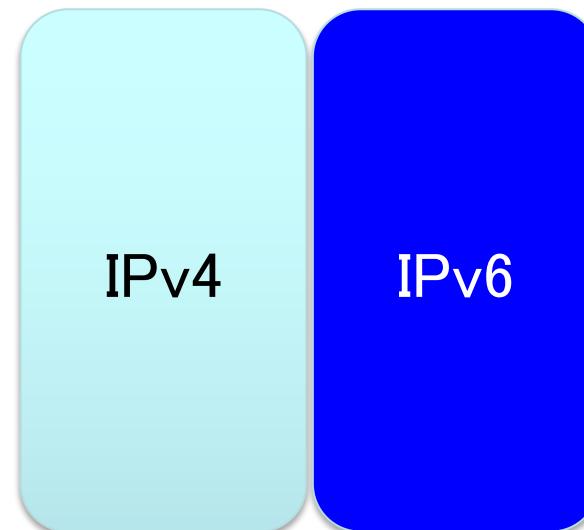
- for ループで回して bind する場合
 - OS によって挙動が異なる。
 - Linux / MacOS の場合 0.0.0.0 が先、:: が次
 - FreeBSD / NetBSD の場合 :: が先、0.0.0.0 が次
 - (よって) 最初に bind に成功したものだけで終了すると
⇒ IPv4 only サーバとなる可能性もある
 - IPv4/IPv6 両方に対応するサーバを作成するには、
IPv6 → IPv4 の順に bind() を実施するのがよい
↳ なぜ？

OS による IPv6 アドレスの設計思想の差



IPv6 は IPv4 を包括する
ものであるという思想

同一ポート番号で IPv4 と IPv6 を
重ねて bind() できない



IPv6 と IPv4 は、それぞれ
別ものであるという思想

同一ポート番号で IPv4 と IPv6 を
重ねて bind() できる

IPV6_V6ONLY

- `setsockopt` にて `IPV6_V6ONLY` というオプションが定義されている
- これを `socket` に対して設定することで、同一ポート番号にて、IPv4 と IPv6 を重ねて `bind()` することが可能となる

```
setsockopt(sock, IPPROTO_IP, IPV6_V6ONLY, &yes, sizeof(yes));
```

多種の OS で動作するサーバプログラム

```
hints.ai_family = AF_UNSPEC;
hints.ai_flags = AI_PASSIVE;
hints.ai_socktype = SOCK_STREAM;
getaddrinfo(NULL, "80", &hints, &res0);

n = 0;
res = res0;
while (res != NULL) {
    sock[n] = socket(res->ai_family, res->ai_socktype, 0);
    setsockopt(sock[n], IPPROTO_IPV6, IPV6_V6ONLY, &yes, sizeof(yes));
    bind(sock[n], res->ai_addr, res->ai_addrlen);
    res = res->ai_next;
    n++;
}
freeaddrinfo(res0);
```

getifaddrs()

- サーバプログラミングをしている場合
 - サーバがどんな IP address を持っているのかが必要となる場合がある
 - 複数 IP address を持っている場合がある
- getifaddrs() を利用すると、IP address のリストを取得することが可能

ifaddrs 構造体

```
struct ifaddrs
{
    struct ifaddrs *ifa_next;          /* Pointer to the next structure */
    char *ifa_name;                  /* Name of this network I/F */
    unsigned int ifa_flags;           /* Flags as from SIOCGIFFLAGS ioctl */
    struct sockaddr *ifa_addr;        /* Network address of this I/F */
    struct sockaddr *ifa_netmask;     /* Netmask of this interface */
    union
    {
        struct sockaddr *ifu_broadaddr; /* Broadcast address of this I/F */
        struct sockaddr *ifu_dstaddr;   /* Point-to-point dest. address */
    } ifa_ifu;
    void *ifa_data;                  /* Address-specific data (may be unused) */
};
```

サンプルコード: getifaddrs()

```
struct ifaddrs *ifa_list, *ifa;
char addrstr[256];

getifaddrs(&ifa_list);
for (ifa = ifa_list; ifa != NULL ; ifa = ifa->ifa_next) {
    printf("%s\n", ifa->ifa_name);
    memset(addrstr, 0, sizeof(addrstr));

    if (ifa->ifa_addr->sa_family == AF_INET) {
        inet_ntop(AF_INET, &((struct sockaddr_in *)ifa->ifa_addr)->sin_addr, addrstr, sizeof(addrstr));
        printf("IPv4 : %s\n", addrstr);
    } else if (ifa->ifa_addr->sa_family == AF_INET6) {
        inet_ntop(AF_INET6, &((struct sockaddr_in6 *)ifa->ifa_addr)->sin6_addr, addrstr, sizeof(addrstr));
        printf("IPv6 : %s\n", addrstr);
    }
    printf("\n");
}
freeifaddrs(ifa_list);
```

実行結果 : getifaddrs()

```
sekiya@LECTURE-CLIENT:~/EXAMPLE$ ./ifaddrs
```

```
lo
```

```
    IPv4: 127.0.0.1
```

```
eth0
```

```
    IPv4: 133.11.205.162
```

```
lo
```

```
    IPv6: ::1
```

```
eth0
```

```
    IPv6: 2001:200:180:452:216:3eff:fe91:5a19
```

```
eth0
```

```
    IPv6: fe80::216:3eff:fe91:5a19
```