I441 2013/07/26
ネットワークプログラミング
— システムコール中心に —

知念

北陸先端科学技術大学院大学 高信頼ネットワークイノベーションセンター
Dependable Network Innovation Center,
Japan Advanced Institute of Science and Technology

アウトライン

篠田先生不在のため、代理講義

1) **おさらい // 初級レベル**
   クライアントやサーバ
2) **中級レベル**
   並行サーバ技術・技法
   - **マルチプロセス、マルチスレッド**
   - **非同期** I/O
3) **高度な話題**
   - **共有メモリ、ノンブロッキング**

参考文献

1) Comer: Internetworking with TCP/IP Volume 3 Linux/POSIX Socket Version
2) Stevens: Unix Network Programming (3rd Edition)
3) Stevens: Advanced Programming in the UNIX Environment (3rd Edition)

初心者は 1) **がおすすめ。半分くらいやればそれなりの腕になれる。**
2) **や** 3) **は内容が濃いで初心者にはつらい。**

システムコールプログラミング

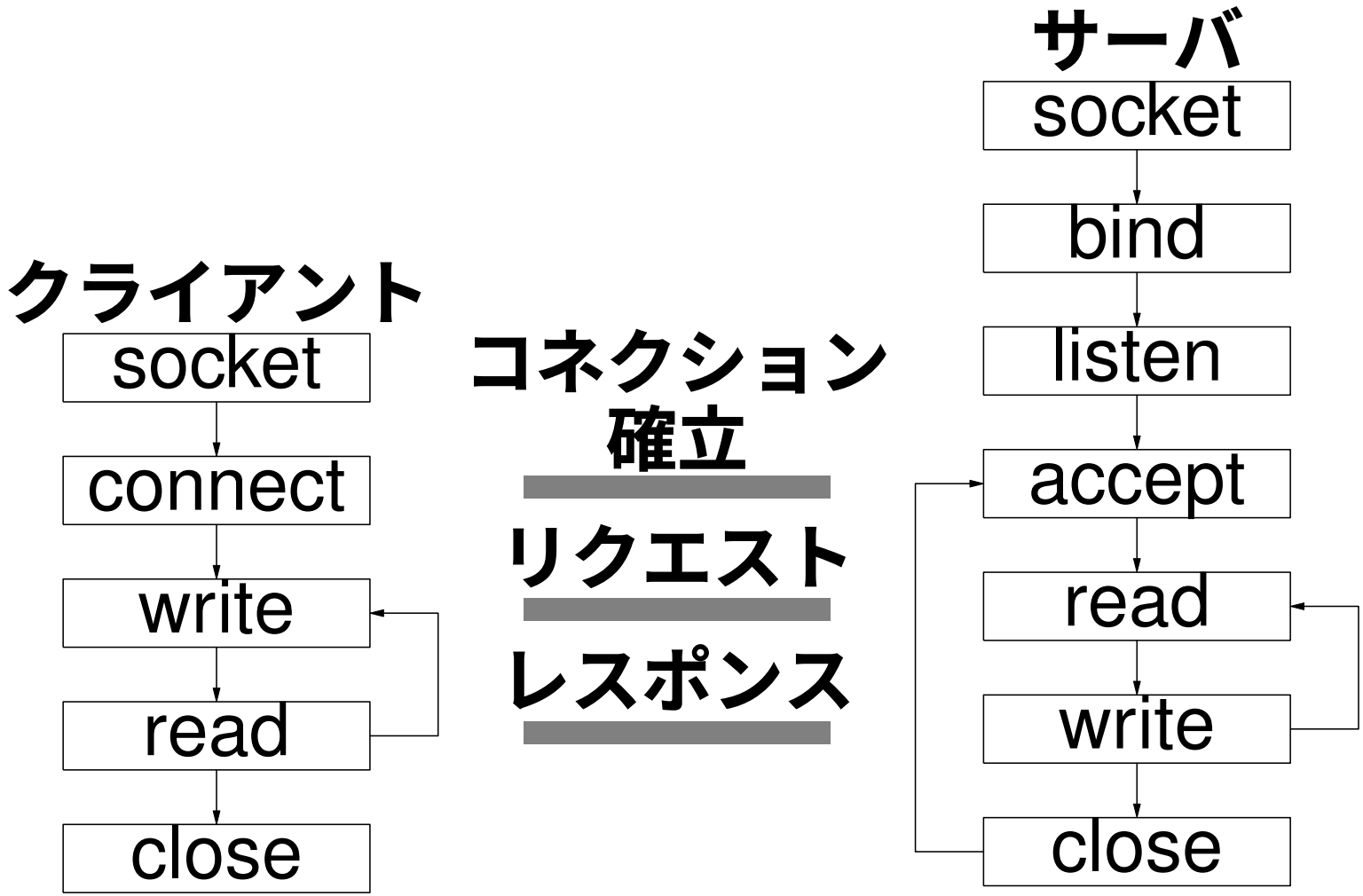
- ネットワークプログラミングの大部分はシステムコール、特にソケット操作
- システムコールは OS に処理を依頼する機構
  C 言語の関数の形態になっている
  - ◇ ソケットは BSD でシステムコールとして登場
  - ◇ それ以外 OS ではライブラリ関数の場合もあり
- 自分のプラットフォームの man を読んで確認
  安易にWWW検索して、別 OS の記述を読んで誤解する事故が多発

man を読もう

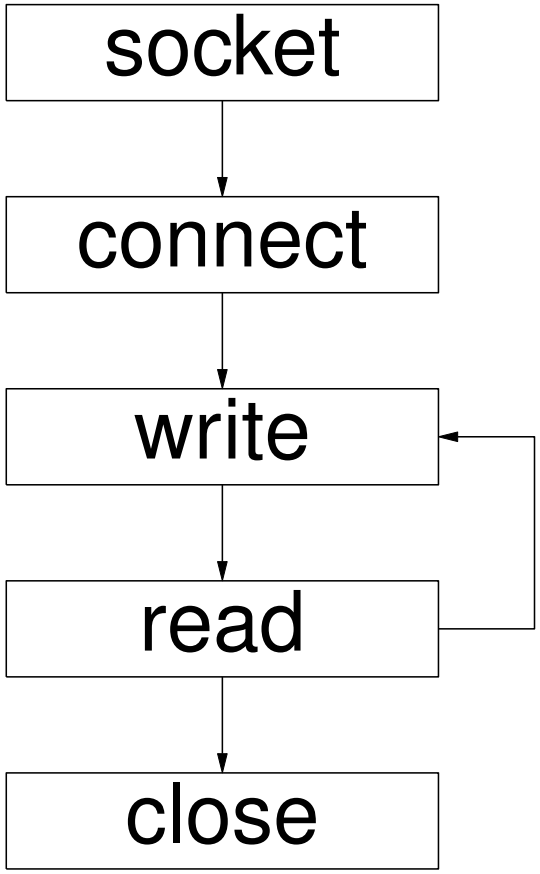
- 引数
  - 型に注意、似たようなもの多数あり
- 戻り値 // RETURN VALUE(S)
  処理の成否、程度等を返す
- errno // ERRORS
  - エラーの理由を格納している
  - 伝統的には int型だが、最近はそうとは限らない
  - errno.h をインクルードすること

```
#include <errno.h>
```

システムコールの流れ


サーバ
socket
bind
listen
accept
read
write
close
クライアント
socket
connect
write
read
close
コネクション
確立
リクエスト
レスポンス

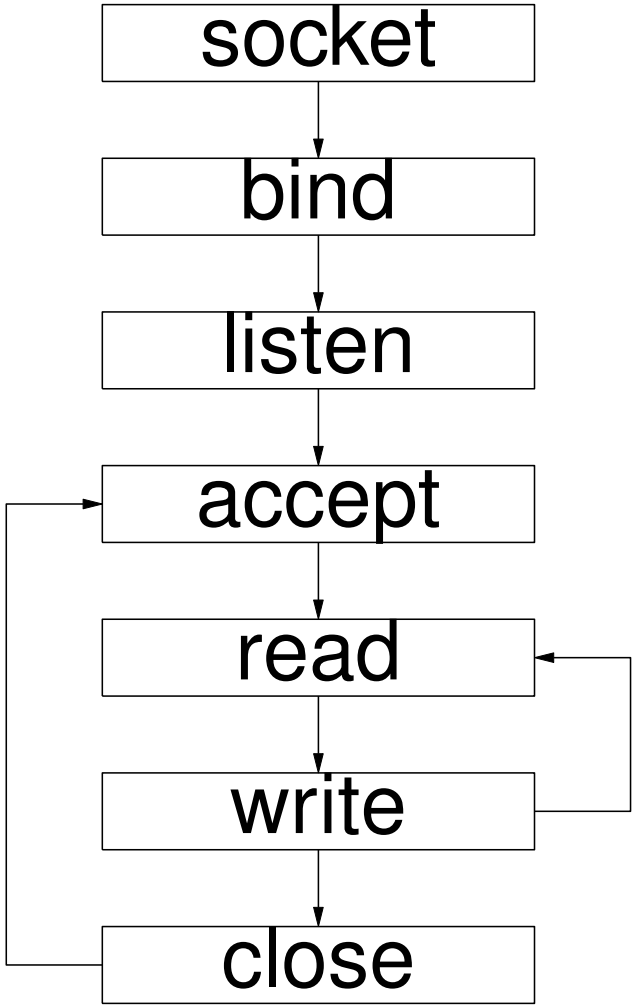
システムコールの流れ *(cont.)*



クライアント側

- アクティブコネクション
- 典型的クライアントの例
  - ◇ HTTP 等
- write と read はプロトコルによって入れ替わる

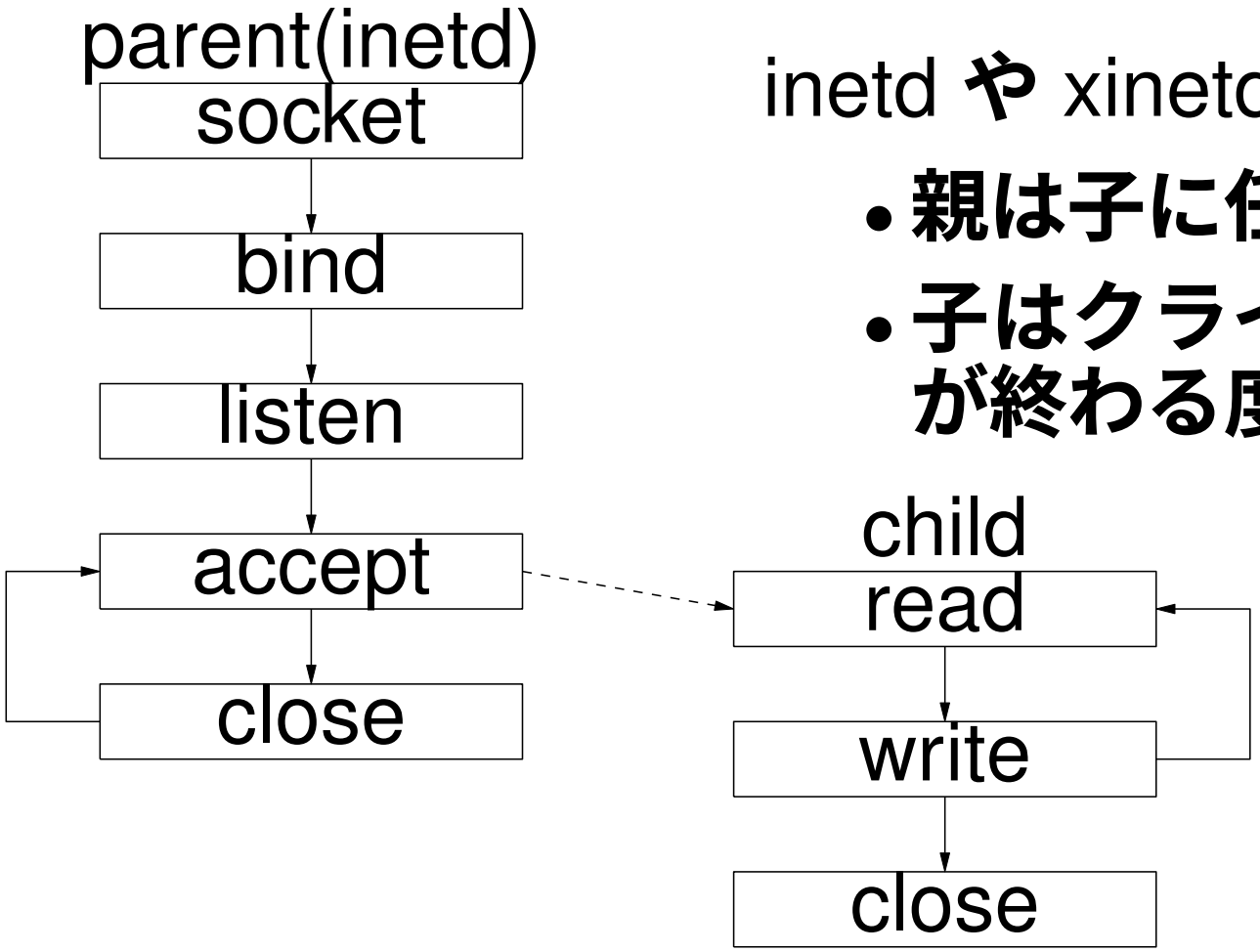
システムコールの流れ *(cont.)*



サーバ側

- パッシブコネクション
- 典型的サーバの例
  - ◇ HTTP 等
- write と read はプロトコルによって入れ替わる
- クライアントとの通信が終わると accept に戻る

システムコールの流れ *(cont.)*



inetd や xinetd の場合

- 親は子に任せて戻る
- 子はクライアントとの通信が終わる度に終了

システムコール補足

- 通信はファイル操作に合わせて実現されている
  - ファイルディスクリプタ (file descriptor; fd) で識別
- socket は open に相当
- read, recv, recvmsg はほぼ等価
- write, send, sendmsg はほぼ等価
- 終了は close 以外に shutdown がある

その他のシステムコール、ライブラリ関数

- setsockopt, getsockopt
  ソケットへの各種オプション設定、取得
  - ◇ **アドレス再利用、ノンブロッキング等で登場**
- gethostbyname, getservbyname, getprotobyname
  ホスト名、サービス名、（トランスポート）プロトコル名から各種番号やID取得
- fork
  プロセス生成、サーバの並行動作で利用
- wait
  子プロセスの終了待機

プログラム例示

- 行数にひるまない
- システムコールの流れを追いかける

IPv6 補足

現在は IPv6 の普及期

- **古い資料は IPv4 向けが多い**
- **IPv4 はアドレスが尽き、今は懸命に延命中**
- **今後は極力 IPv6 向けを作る方が良い**
  古いシステムはいたしかたない

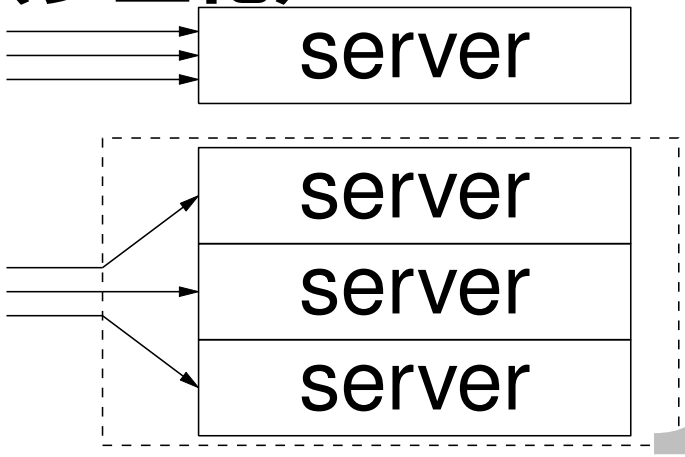
注意点

- sockaddr_in **避けて** sockaddr_storage
- gethostbyname **避けて** getaddrinfo/getnameinfo
- **等々**

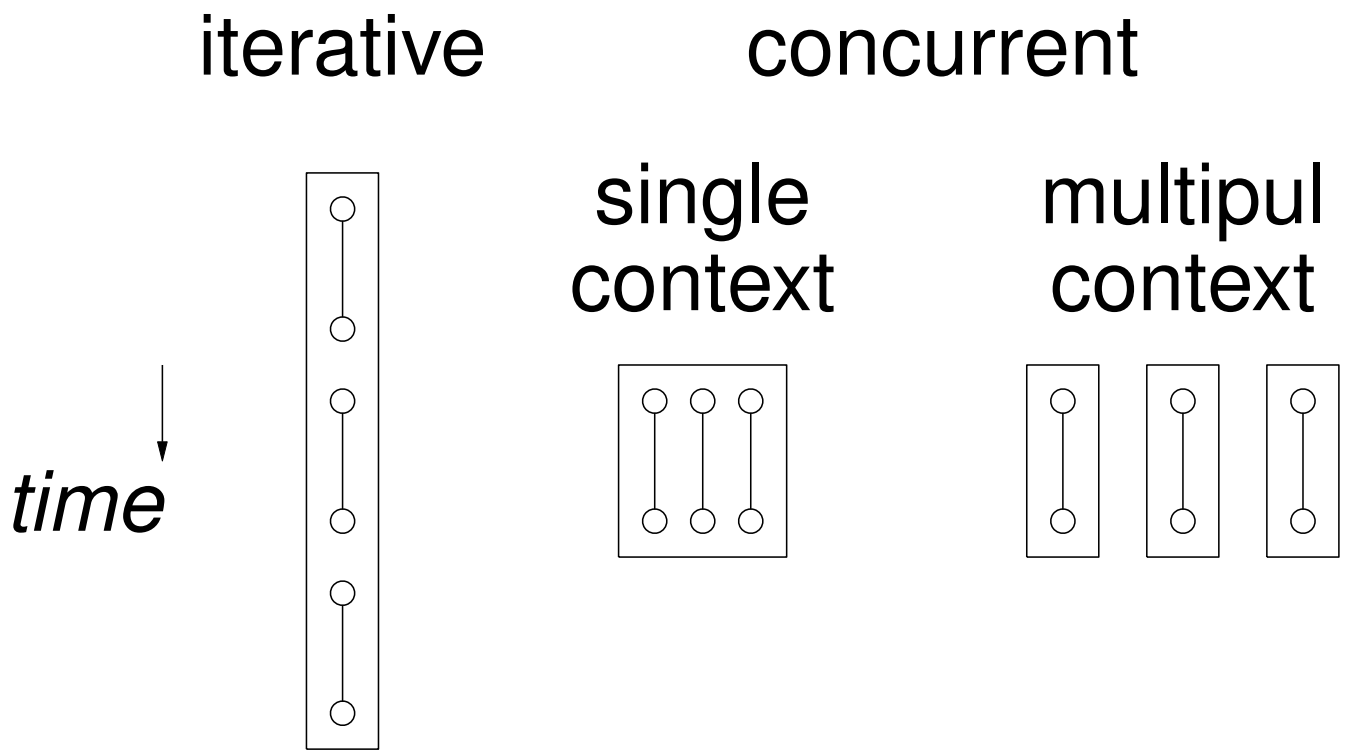
サーバ設計

- iterative server; **繰り返しサーバ**
  アクセスが集中すると捌けない

- concurrent server; **並行サーバ**
  アクセスの集中に対応（多重化）
  - ◇ **単コンテキスト**
  - ◇ **複コンテキスト**
    - ⋆ **マルチプロセス**
    - ⋆ **マルチスレッド**



多重化—タイミングチャートで説明



- コンテキストと処理の組み合わせの議論
- 繰り返しサーバは非常に時間がかかる

繰り返しサーバ概要

一件づつ処理 — 並行コネクション 1

$M/M/1$

```
lfd = socket(...)
bind(lfd,...)
listen(lfd,...)
while(1) {
    sfd = accept(lfd, ...)
    read(sfd)
    write(sfd)
    close(sfd)
}
```

※ 通信数制限のために、あえて採用する事もある

並行サーバ、マルチコンテキスト概要

リクエストの度に fork でプロセス生成

$M/M/n$

```
前半省略
while(1) {
    sfd = accept(lfd, ...)
    pid = fork()
    if(pid>0) {
        close(sfd);
        continue;
    }
    read(sfd)
    write(sfd)
    close(sfd)
    exit(0)
}
```

マルチコンテキスト、マルチスレッド化

大雑把にはプロセス操作をスレッド操作に置換

- fork を pthread_creare に
- exit を pthread_exit に

注意点

- **どれかのスレッドが異常終了すると全滅**
- **ある場所のメモリを同時に書き込むと動作不定**
  同時書き込み防止技法を使う
  - ◇ mutex, condition variable 等

コンテキスト生成負荷

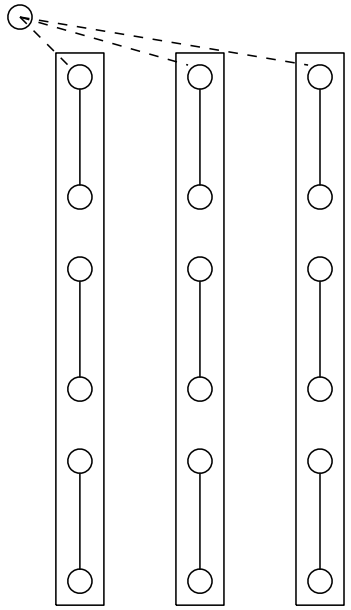
並行サーバのトレードオフ

- 処理とコンテキストを 1対1 対応させると理解しやすい、開発しやすい
- しかし fork や pthread_create は高い負荷が発生
  - ◇ 実は fork は accept より重い処理
  - ◇ コンテキスト生成回数を減らす工夫が欲しい
- 一方、単体コンテキストは非常に複雑プログラム
  - ◇ バグが増える可能性が高い

コンテキスト生成回数減らす工夫

並列と繰り返しの折衷

- 全体は並列
- 親は子に任せる
- 子は繰り返し、回数限定



⇒ 生成回数が効果的に減少する

*　　*　　*

- accept とは独立にコンテキスト生成

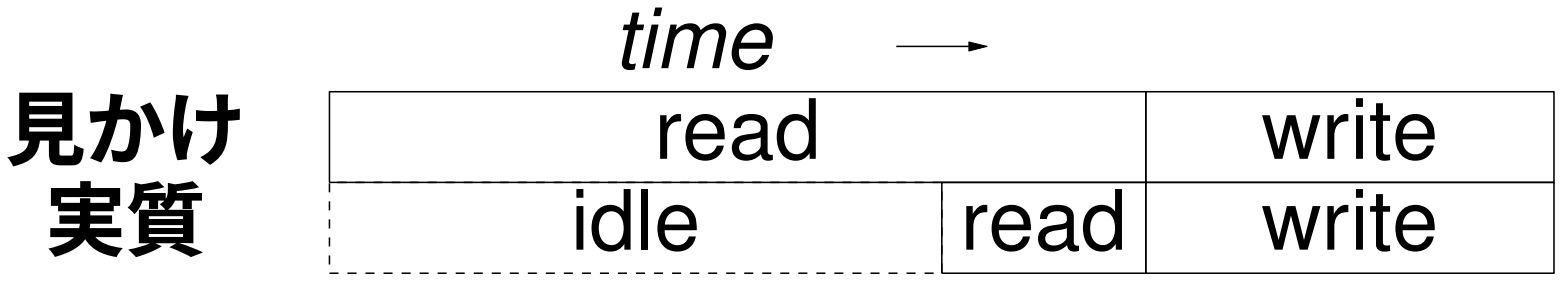
⇒ アクセス集中に依存しなくなる

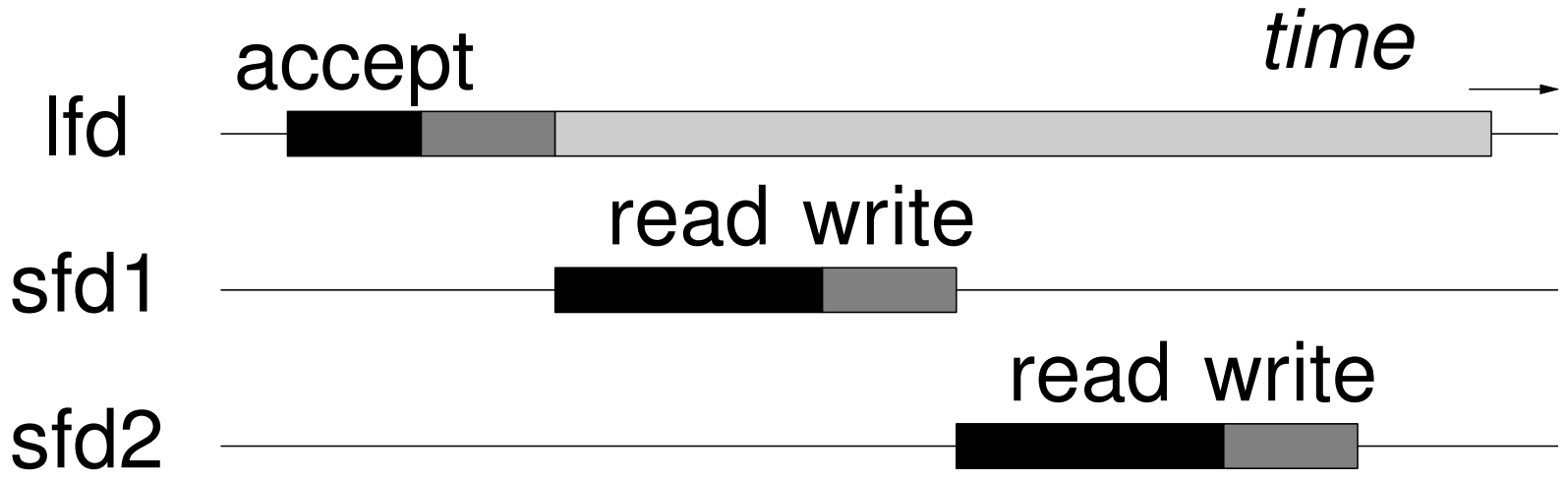
シングルコンテキスト並行サーバ

- 一つのコンテキストで大量コネクション扱う
  - ◇ 数万コネクションに達する事もある
- 非同期 I/O が必須（次ページ）
  - ◇ 一つの accept と多数の read/write を処理
- コンテキスト周りのオーバヘッド削減
  - ◇ それ以外、I/O 待ち等は削減できない
- プログラムが非常に難しい

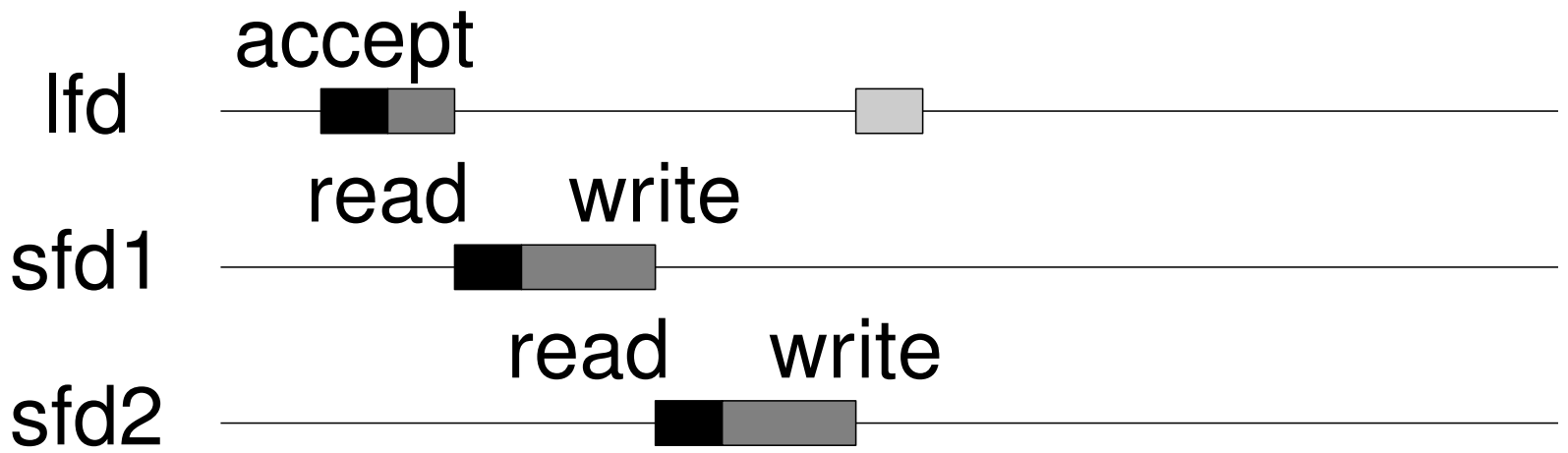
非同期 I/O

- **一つの** accept **と多数の** read/write **を処理**
  - ◇ accept **だけ処理する訳には行かない**
  - ◇ read/write **だけ処理する訳にも行かない**
- **特定のシステムコールの成否を待つのではなく、処理できるコネクションから処理する**
- **成否決定まで待っている場合が多い**: read **の例**





非同期 I/O で待ち時間短縮すると...



その他の待ち時間

ブロックするシステムコールに待ち時間が存在する

- accept: **待ち時間多い**
- read: **待ち時間多い（前述）**
- write: **待ち時間なし**
  原則バッファリングしているので待ち時間はない
- close: **待ち時間なし**
- bind,listen: **待ち時間なし**
- connect: **待ち時間多い**
  複雑なので、非同期コネクションの箇所で紹介

公平性

- 前提はコネクション多数
- なんらかの順序で処理を進めることになる

- 固定順序では若い位置のコネクションが優先に
  - ◇ accept 受理順、fd の数字若い順、等
- 何らかの工夫が必要
  - ◇ 先頭から処理するのやめる
  - ◇ 前回の位置を覚える
  - ◇ ランダム性を導入する

非同期 I/O 向けシステムコール

- select
  連続した fd の配列で対象を指定
  - ◇ fd の数が多いと処理時間が長い
  - ◇ 処理の偏りが出やすい
  - ◇ 前準備が複雑
- poll
  fd を含む構造体の配列で対象を指定
  fd が連続している事は期待していない

select

fd の集合(fd_set)に監視対象 fd を登録する
以下は lfd を監視してaccept結果 sfd を登録

```
FD_ZERO(&fds);
FD_SET(lfd, &fds);
while(1) {
   memcpy(&rfds, &fds, sizeof(fds));
   ik = select(nfd, &rfds, NULL,NULL,NULL);
   if(FD_ISSET(lfd, &rfds)) {
      sfd = accept(lfd, ...);
      FD_SET(sfd, &fds);
      if(sfd>nfd) nfd = sfd;
   }
}
```

fd_set 操作マクロ

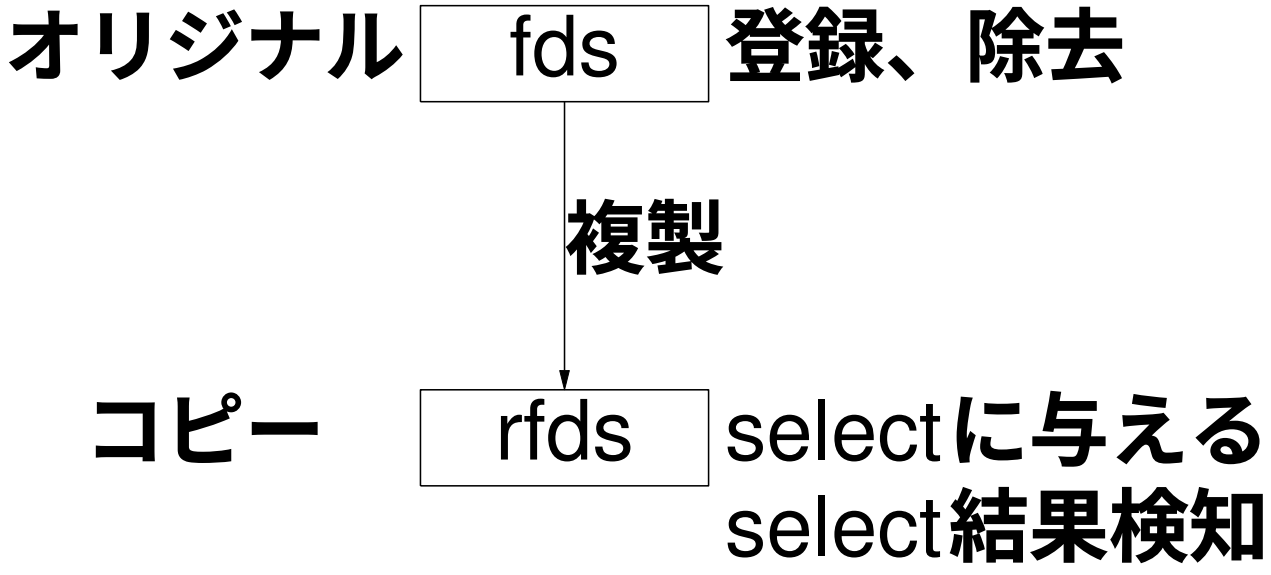
- fd_set の内部はビット操作
- 簡単のためマクロが用意されている

FD_ZERO	全体初期化
FD_SET	fd 一つを登録
FD_ISSET	fd 一つが変化したか確認
FD_CLEAR	fd 一つを除去

typedef で型として fd_set を定義している場合も

fd_set 操作マクロ *(cont.)*

- select は fd_set の内容書き換えるので、複製を作って select に与える
- select 結果は複製した方に格納される



select 補足

```
int select(int nfd, fd_set *rfds,
   fd_set *wfds, fd_set *efds,
   struct timeval *tout)
```

`nfd` は管理対象fd集合中の管理対象の長さ
`rfds` は read 監視対象の fd集合
`wfds` は write 監視対象の fd集合
`efds` は例外監視対象の fd集合
`tout` はタイムアウト（打ち切り）時間

先の例は read だけ監視して、ずっと待つ

```
ik = select(nfd, &rfds,NULL,NULL,NULL);
```

コネクション数

- Linux は数百万までチューニングなしで使える
  CentOS 6.4 メモリ 48GB で 500万弱
- 昔の OS は制約が多かった
  - ◇ カーネルリコンパイル
  - ◇ さまざまな設定をチューニング
- 一度に扱えるコネクションは実質上、数百・数千
  コネクションの変化に select/poll がついて行けなくなる

コネクション管理機構

本格的単コンテキスト並行サーバでは必要
一般的には多数の情報を含む構造体の配列

- ユーザ、相手の IP アドレス
- コネクションの fd
- リクエスト
- その他

操作

- accept 結果から新規登録
- select に与える fd_set 作る、select 結果走査

コネクション管理機構の例

```
typedef struct {
   char *user;
   struct sockaddr_storage *addr;
   int    fd
   char *req;
   struct timeval start;
} conn_cntl_blk;

int nconns=0;
conn_cntl_blk *conns=NULL;

int addnewconn(int nfd);
int extracefdset(struct fd_set *fds);
conn_cntl_blk* findconn(
         struct fd_set *fds);
```

poll

select とほぼ同じ位置づけ、SYSV 由来

```
struct pollfd fds;
fds.fd     = lfd;
fds.events = POLLIN;
ik = poll(&fds, 1, wtime);
if(fds.revents & POLLIN) {
    sfd = accept(lfd, ...);
}
```

監視対象の fd を fds、内容を events に登録
結果は revents に格納される

第一引数は配列なので対象が多くてもよい

poll *(cont.)*

直接 fd を使っていないので、監視順序変更が容易

```
struct pollfd *fds;
fds[0].fd      = 7;
fds[0].events = POLLIN;
fds[1].fd      = 5;
fds[1].events = POLLIN;
ik = poll(fds, 2, wtime);
```

この例、select では 8個の fd 与える必要がある

```
FD_ZEERO(fds)
FD_SET(5, &fds)
FD_SET(7, &fds)
memcpy(&rfd, &fds, sizeof(fds));
ik = select(8, &rfd, NULL, NULL, NULL);
```

非同期 I/O エラーハンドリング

- 他と違い select/poll は複数コネクション扱う
  各コネクションのエラーは返せない
- 各コネクションのエラーは getsockopt で拾う

```
elen = sizeof(err);
chk = getsockopt(xfd, SOL_SOCKET,
        SO_ERROR, &err, &elen);
if(err) {
  if(err==ECONNREFUSED) {
    ...
  if(err==EHOSTUNREACH) {
    ...
}
```

先進的話題

- 高速なシステムコールや機構が作られている
  - epoll
  - kqueue
  - /dev/poll

  カーネル側での対応が必要
- これらに対応したライブラリも作られている
  - libevents 等