I441 2013/07/30
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

前回

- システムコールの流れ
- 並行サーバ
  - マルチプロセス
  - 非同期 I/O

今回

- 非同期コネクション
- エラーハンドリング
- 共有メモリ、等々

性能向上のヒント

- コンテキストスイッチを削減
  - 繰り返しサーバと並行サーバの混合
    fork/pthread_create の発生を軽減する
- 潜在的な待ち時間を削減 — ブロック回避・軽減
  - 非同期 I/O
    ブロックするシステムコールを呼ぶ前に受信・送信可能かどうか確認
  - 非同期コネクション（次ページ）
    システムコールでブロックしないようにする

非同期コネクション、ノンブロッキング

- ブロックしない、即座に返ってくる技法
- ノンブロッキングでは戻り値で成否確認できない
  - ◇ 成否の確定を待たないのだから当然
  - ◇ errno に EAGAIN や EINPROGRESS が入る
- 具体的な切り替えは fcntl で

```
fd = socket(...);
opt = O_NONBLOCK;
ik = fcntl(fd, F_SETFL, opt);
```

非同期コネクション、ノンブロッキング *(cont.)*

現在値を引き継いで切り替えると無難
他の属性を破壊しない

```
fd = socket(...);
opt = fcntl(fd, F_GETFL, 0);
opt |= O_NONBLOCK;
ik = fcntl(fd, F_SETFL, opt);
```

XOR で戻す

```
opt = fcntl(fd, F_GETFL, 0);
opt ^= O_NONBLOCK;
ik = fcntl(fd, F_SETFL, opt);
```

空回り防止・回避

システムコールでブロックしなくなると...

- 無闇に空回りするプログラムになることも

```
while(1) {
  nr = read(fd, buf, BUFSIZ);
}
```

通称*busy loop*、CPU を浪費するので極力避ける

対策

- スリープを入れる
- select/poll で読み込めるか確認する（前回）
- タイムアウトつき select/poll が有効

空回り防止・回避 *(cont.)*

スリープ例

```
while(1) {
   nr = read(fd, buf, BUFSIZ);
   sleep(1);
}
```

- **複数の単位が用意されている**
  - sleep **は秒**[s]
  - usleep **はマイクロ秒**[us]
  - nanosleep **はナノ秒**[ns]

※ 精度は OS **やライブラリによる、盲信・過信しない**

タイムアウト付き select

タイムアウト(1.5秒）つき select 例

```
FD_ZERO(&fds)
FD_SET(fd, &fds)
tw.tv_sec = 1;
tw.tc_usec = 500*1000;
while(1) {
  memset(&rfd, &fds, sizeof(fds));
  ik = select(fd+1, &rfds, NULL,
              NULL, &tw);
  if(ik>0) {
    nr = read(fd, BUF, BUFSIZ);
  }
}
```

読込監視を1.5秒続ける

タイムアウト付き select *(cont.)*

read ブロックの select ブロック置換ともみなせる

- 受信あれば読み込み可能を検出して即座に返る
- 受信なければ指定時間経過後に返る

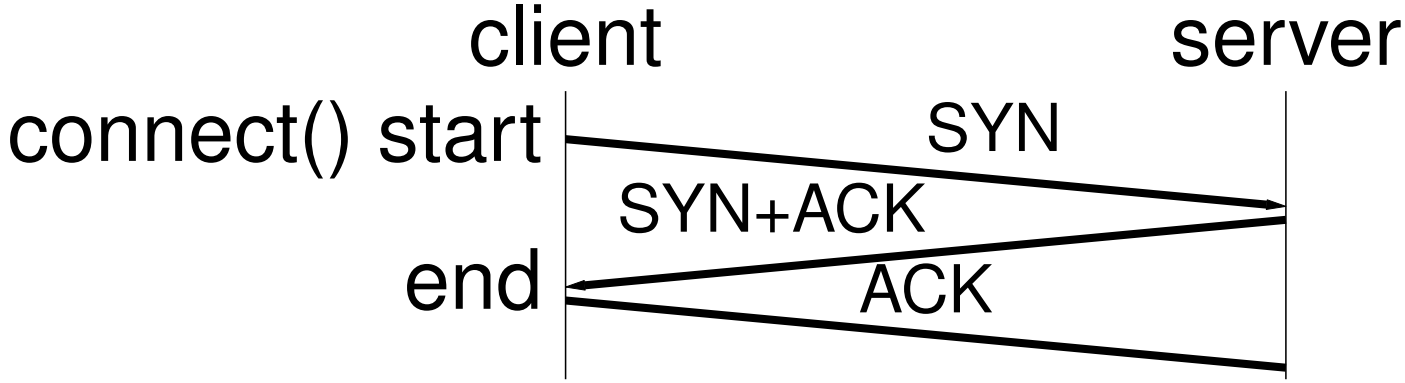
したがって、ループ全体的には

- 受信があれば即座に受信を処理
- 受信が続く場合も即座に次の受信を処理
  スリープの方は必ず待つ、レート制御に近い
- 受信が無ければスリープにほぼ等価

※ レート制御は setitimer を参照

connect の待ち時間削減

- 非同期コネクションの主な用途の一つ
- connect の待ち時間は比較的長い
  - TCP のレベルで考える



  - この時間（数十ms〜数百ms）も節約したい
  - 待っている間に他の処理がしたい

connect の待ち時間削減 *(cont.)*

```
fd = socket(...)
ik = enable_nonblock(fd)
ik = connect(fd,...)
if(ik<0 && errno!=EINPROGESS) {
   ＜＜connect開始失敗時の処理＞＞
}
＜＜他の処理＞＞
if(iswritable(fd)) {
   ik = scanerror(fd, &err)
   if(err) {
      ＜＜connect 失敗時の処理＞＞
   }
   disable_nonblock(fd)
}
```

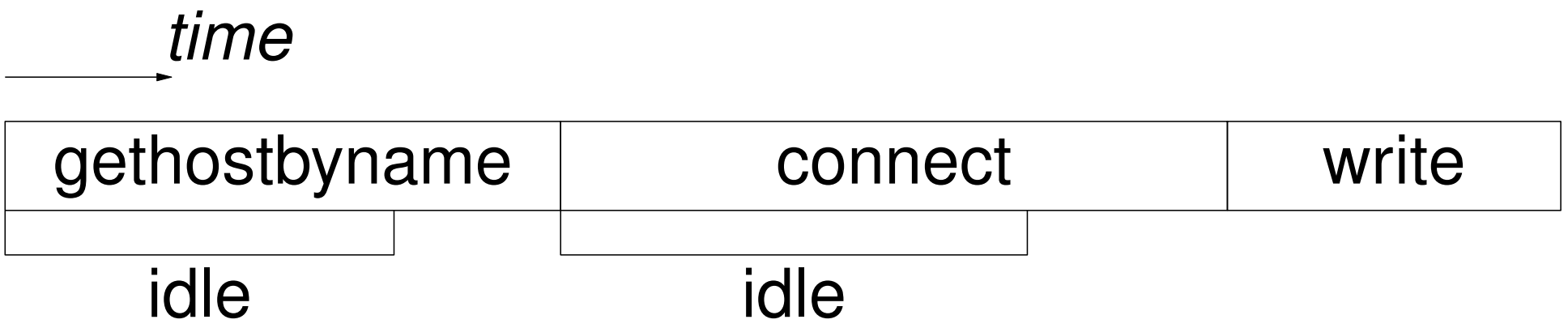
connect の待ち時間削減 *(cont.)*

- enable_noblock: **ノンブロッキングに変更**
- disable_noblock: **ブロッキングに変更**
- iswritable: **書き込み可能か確認**
  非同期コネクションの connect **は成否確定すると書き込み可能になる　書き込み可能のみ検出例**

```
ik = select(nfd,NULL,wfds,NULL,NULL)
```

- scanerror: getsockopt **でエラーを確認（前回）**
  通常の errno **と同様な値を得る**

クライアント connect 前後

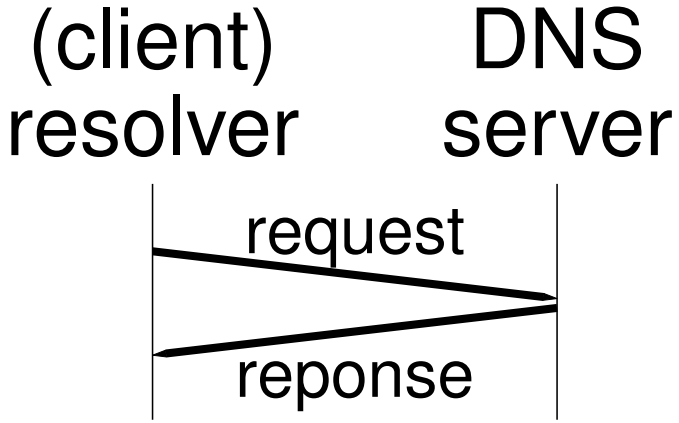


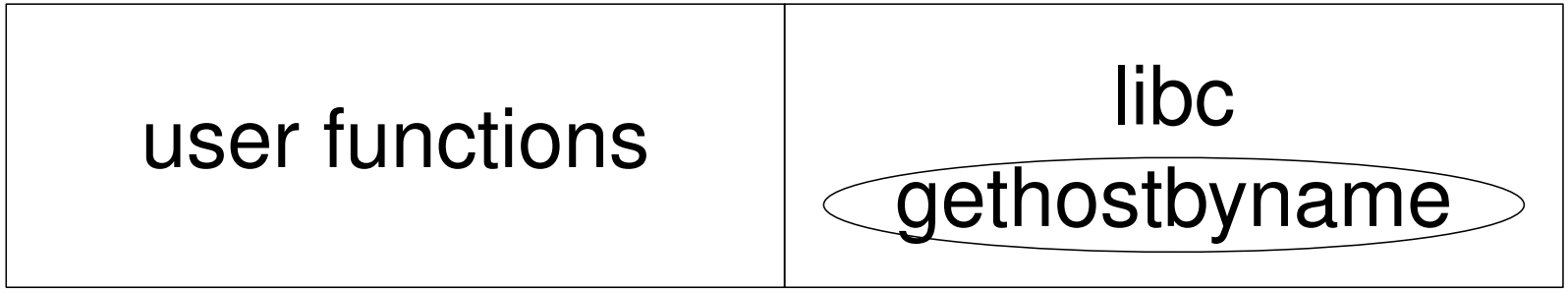
- **実は** connect **の前にも処理がある**
  - gethostbyname/getaddrinfo **等の問い合わせ（次ページに詳細）**
- write **は待ち時間ほぼなし**
  - **バッファリングされているから（前回登場）**

gethostbyname 類の待ち時間

ホスト名から IP アドレスへ変換

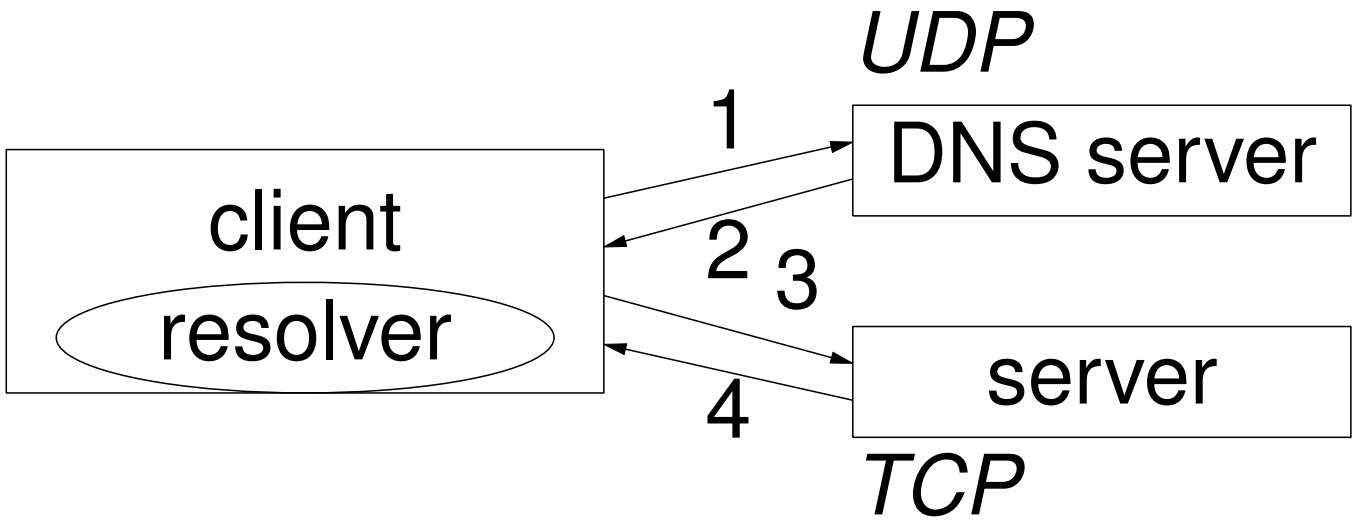
- **主に** DNS **の通信**
  主に UDP **で実現**
  ここでも待ち時間が存在
- resolver **は** DNS client **を指す**
- **ライブラリ内の処理なので容易には短縮できない**





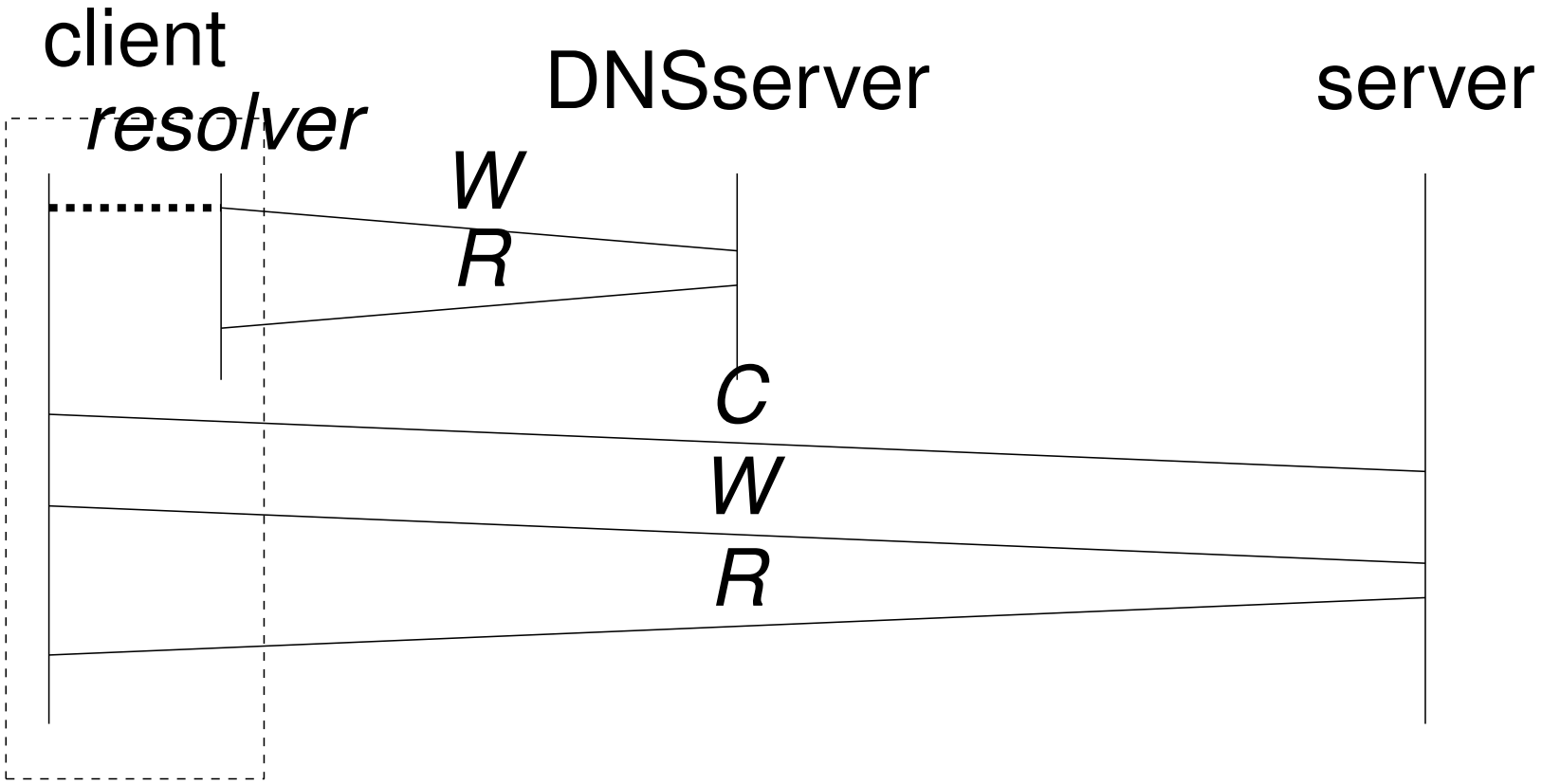
gethostbyname 類の待ち時間 *(cont.)*

基本形のプログラム関係図



gethostbyname 類の待ち時間 *(cont.)*

基本形のタイミング図



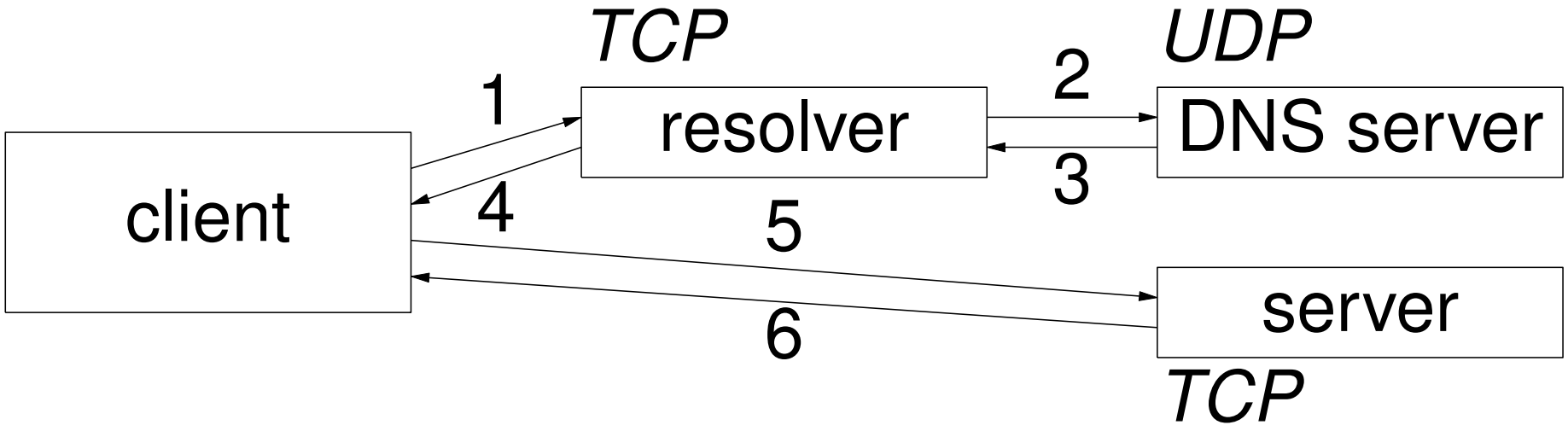
gethostbyname 類の待ち時間 *(cont.)*

対策

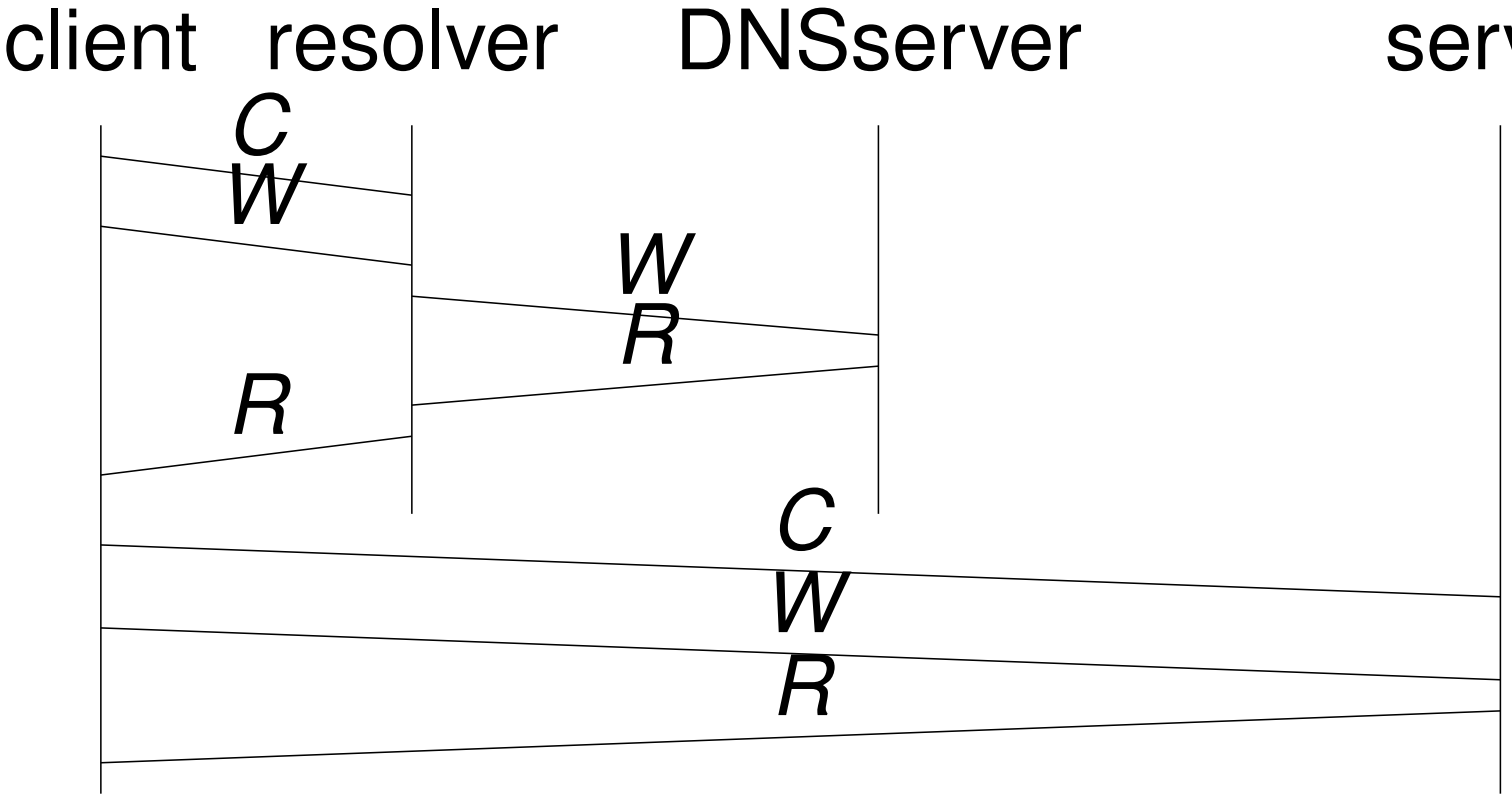
1) resolver を別プロセスにする
   resolver 作業を TCP で応答するサーバに
   隠蔽された resolver の待ち時間を
   待ち時間制御可能な connect, read に読み代え
2) キャッシュを設ける
   問い合わせ回数を減らす
3) 別 resolver 関数
   待ち時間のない関数があれば良い

対策1: resolver 別プロセス

- resolver が gethostbyname 呼び出し
- クライアントは TCP のみ扱う
- TCP なら待ち時間制御可能に



対策1: resolver 別プロセス *(cont.)*



対策2: キャッシュ

gethostbyname **呼び出し回数軽減**

自作

- gethostbyname **結果を記録**
- gethostbyname **呼出前に記録を走査**

他実装

- unbound **等**

繰り返しが多ければ効果大きい

対策3: 別 resolver 関数

待ち時間のない／短い関数を使う
待ち時間中に別処理できる関数／機構を使う

自作

- UDP で DNS 問い合わせ
- 一定時間でタイムアウト

他実装

- UDNS, adns, libdns 等
- light weight resolver 等

他実装注意点

- 他人のバグに引きずられる可能性

  ※ libc内蔵 gethostbyname のバグ可能性極小
- 望みのプラットホームで動かない可能性
  - ◇ 組み込み環境
  - ◇ 古い OS
- 各種内蔵・関連機構への影響
  - ◇ resolv.conf
  - ◇ NetManager

エラーハンドリング

- システムコール返り値を確認する（再掲！）
- エラーでも、一時的な場合がある

考慮しなければならないエラー値（代表的）

EINTR	割り込み発生
EAGAIN	もう一回
EINPROGRESS	進行中
ECONNRESET	相手が解放・切断した
EPIPE	相手が切断した

エラーハンドリング: EINTR

システムコール処理中に各種割り込みが発生すると、処理が中断して EINTR を返す

```
try_accept:
    sfd = accept(lfd,...)
    if(sfd<0) {
        if(errno==EINTER) {
            goto try_accept:
        }..
    }
```

EINTR が出てきたら、繰り返す
EAGAIN も同様

※ 発生度合いは OS に依存する

エラーハンドリング: EPIPE

書き込み時に通信が切断すると EPIPE が発生

```
nw = write(sfd,...)
if(nw<0) {
     if(errno==EPIPE) {
          return -1
     }...
}
```

EPIPE を見つけたらそれ以降の処理は無理

- return か exit で対処

PIPE 設定

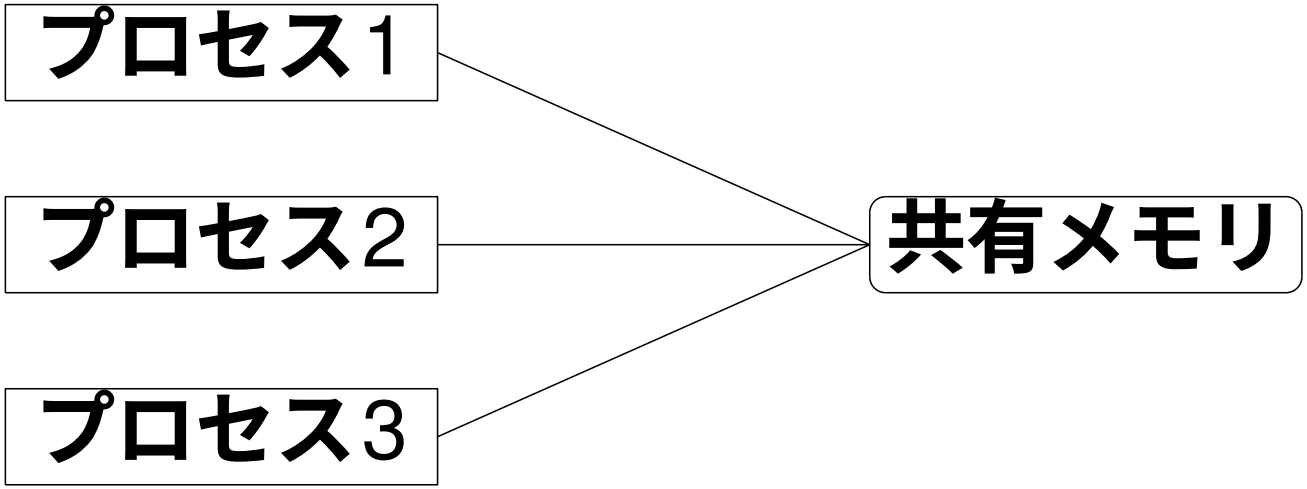
- EPIPE 発生処置は原則プロセス停止となる
  - ◇ 小さなプログラムはその方が無難
- プロセス停止の無効化は signal 使う

```
signal(SIGPIPE, SIG_IGN);
```

  POSIX に合わせるなら sigaction がおすすめ
- 無効にしないと簡単に終了するプログラムになってしまう
  サーバでは使い物にならない

共有メモリ

複数のプロセスでメモリを共有する



親子関係にないプロセス間で同じデータ扱える
プロセスが終了しても保持される点に注意

共有メモリ *(cont.)*

システムコールやライブラリ関数は 2 系統存在

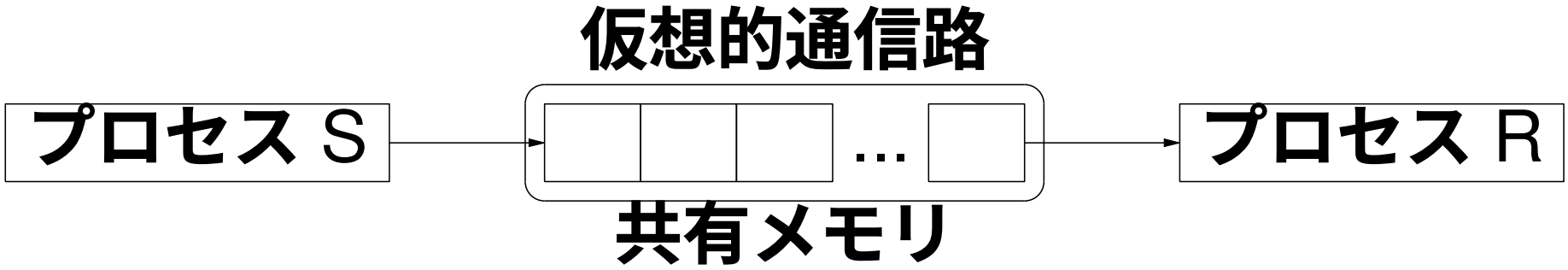
- shm: shmget, shmat, shmdt
  - ◇ key **や** id **で管理したメモリ領域を扱う**
- mmap
  - ◇ **ファイルをメモリ上に投影する**

注意点

- **サイズ上限等** OS **によって異なる**
  - ◇ shm **は古い** OS **では領域が小さい**
- semaphore **等で同時書き込みを回避する**

共有メモリを使った通信

- 一方からデータを共有メモリに書き込む
- 他方でデータを共有メモリから読み込む



- 固定長なら実装容易
- 可変長は長さに十分注意

ネットワークスタックを介さないので高速

ソケットその他

アドレス再利用

- 同じ IP アドレスを複数のコネクションで使う
- サーバの多くではこの設定が必要

```
flag = 1;
setsockopt(fd, SOL_SOCKET, SO_REUSEADDR,
     &flag, sizeof(flag));
```

ソケットその他 *(cont.)*

fd パッシング

- sendmsg, recvmsg で fd を交換できる

- msghdr, iovec 等データ構造が少し複雑
- 親子関係ないプロセス間でも可能

応用システム: 代理サーバ

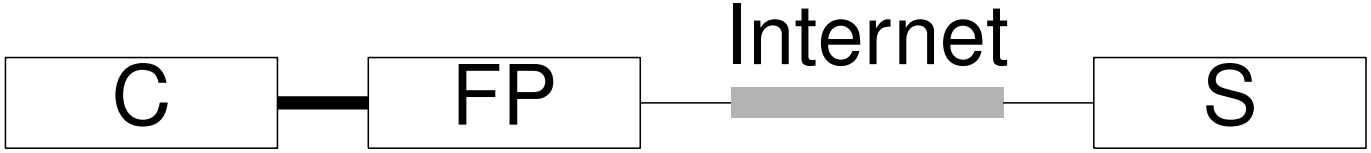
代理サーバ (proxy server)

- **クライアントとサーバの間で通信を中継**

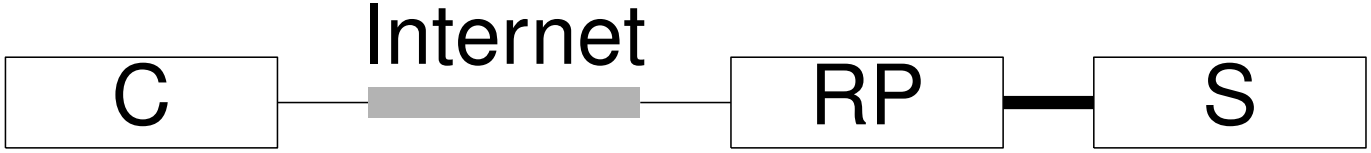
- **キャシングやデータ変換等を装備する場合も**
  - ◇ **キャッシュ持つと応答が短くなる**
  - ◇ **キャッシュ持つとサーバへのトラヒック減る**
- **クライアントとサーバの両方の機能を持つ**
  - ◇ **プログラムの難易度高い**

応用システム: 代理サーバ *(cont.)*

- forward proxy // **特に指定なければこれ**
  - **クライアント側に設置**
  - **キャッシュがあるといわば読み込みキャッシュ**



- reverse proxy
  - **サーバ側に設置**
  - **キャッシュがあるといわば書き込みキャッシュ**

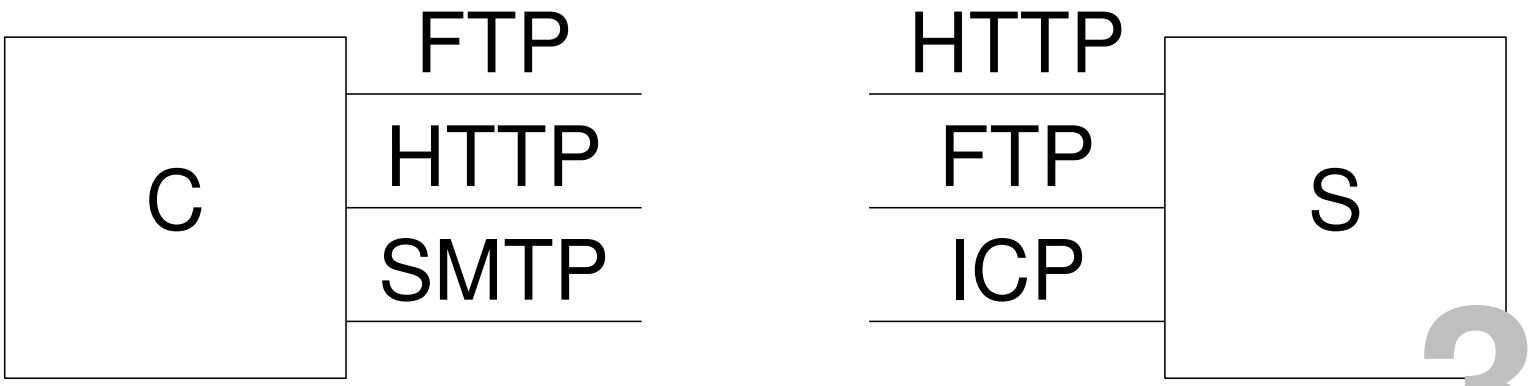


応用システム: マルチサービス

プログラムは一つのサービス（アプリケーションプロトコル）に限定されない

※ ここまでは簡単にするために触れなかった

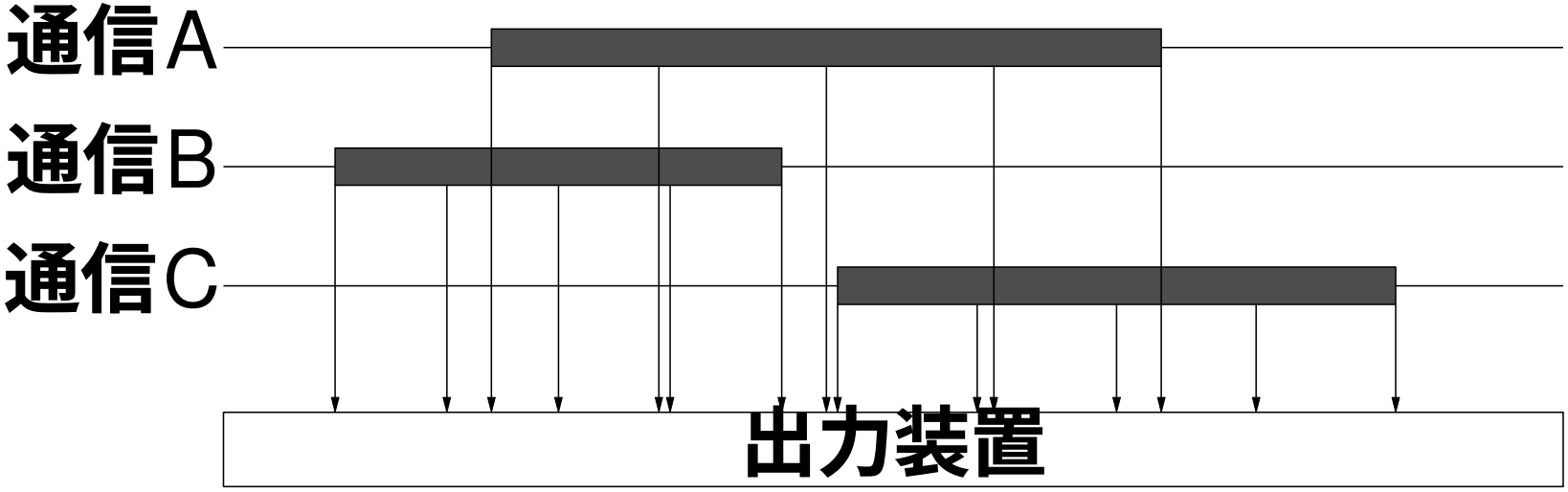
- 複数サービス利用するクライアント
- 複数サービス提供するサーバ



応用システム: マルチサービス *(cont.)*

クライアントでは出力が混ざらないよう注意

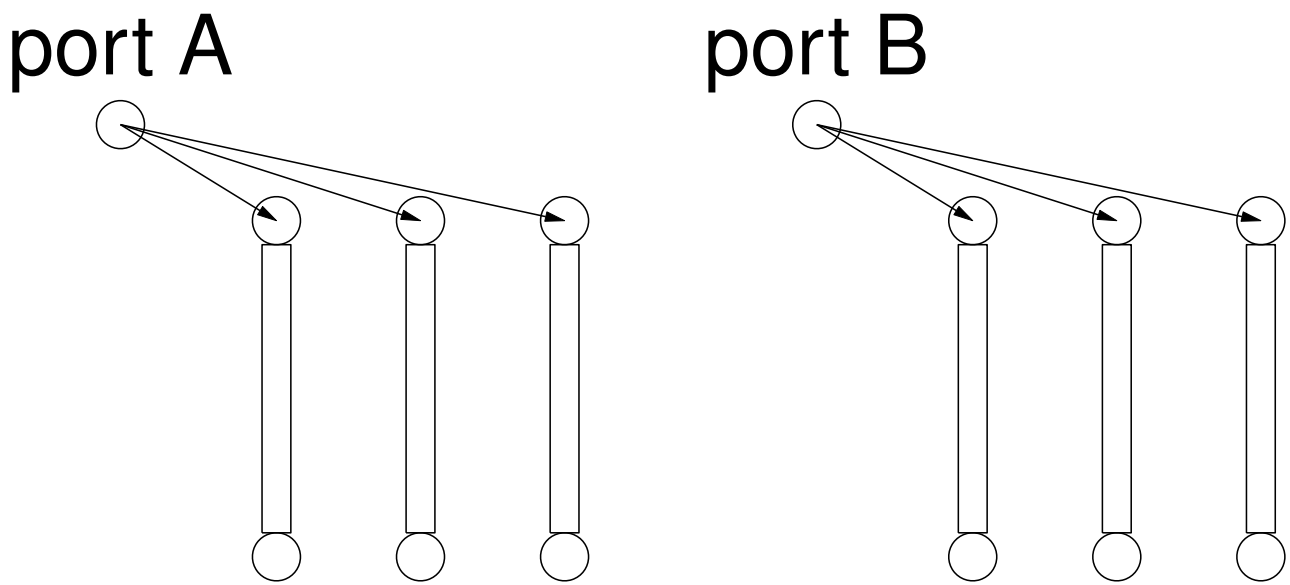
- CUI ならコンソール
- GUI ならウィンドウ



このまま出力すると、BBABABBACCACACC

応用システム: マルチサービス *(cont.)*

サーバは複数 listen fd を持つ



内容に応じてコンテキスト等を決める

参考実装

- **サーバ**
  - Apache, nginx
- **代理サーバ**
  - Squid
- **ベンチマーク**
  - ab(Apache**内蔵**), httperf, JMeter
  - WebPolygraph

プラットホームの差異と対策

差異

- ヘッダファイルやライブラリの有無
- 型定義有無や内容、関数定義の有無や引数

対策

- 小さな差異は同等品を用意する
- 大きな差異
  - 足りないだけならインストールを促す
  - 使わずに済む回避策をとる
  - 諦める

configure

configure と make だけでコンパイルを可能に

```
% gtar ztvf some.tar.gz
% cd some
% ./configure
% make
```

様々なプラットホームで、これだけでコンパイル

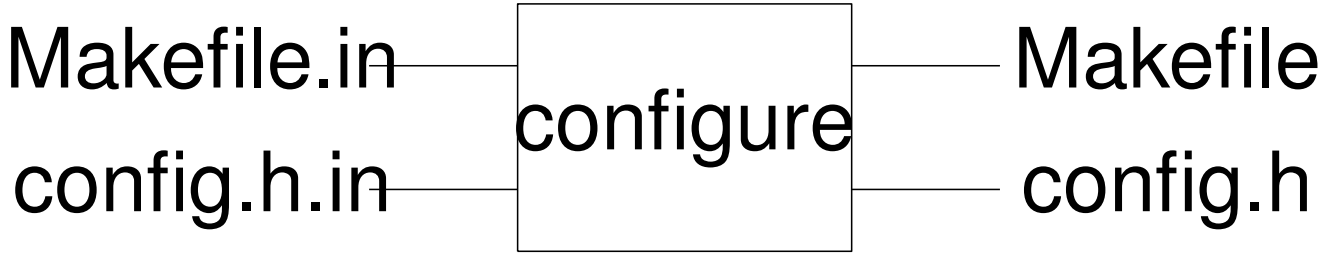
- OS 由来の差異を吸収 ※ UNIX 中心
  - BSD, Linux, Solaris, MacOS
  - Linux の様々ディストリビューション
- ヘッダ・ライブラリの差異を吸収

configure *(cont.)*

configure は

- autoconf で生成されるシェルスクリプト

- Makefile や config.h 等を生成する



- config.h に差異や処理方針、その内容を記録

configure *(cont.)*

- ファイルシステムからヘッダやライブラリを把握
- config.h 上では把握結果はマクロとして残る

```
#define HAVE_SELECT
```

- .c や .h ではマクロに応じたコードを書く

```
#ifdef HAVE_SELECT
#define isreadable isreadable_select
#else
#define isreadable isreadable_poll
#endif
```

※ Makefile を作る関連プログラム automake もある