ネットワークプログラミング

第４回
ソケットプログラミング（４）
－再びLinuxです－

ソケットオプション

- TCP/IPプロトコルの開発では、できるだけ多くのアプリケーションにとって都合が良いようにデフォルトの動作が決められたようだ。
  - いわば、「フリーサイズ」の服。しかし、それがぴったりあうことはめったにない。
  - 受信バッファのサイズはどれくらいあれば足りるのか？

→ソケットの動作を決めるさまざまな特性は、「ソケットオプション」で調整できる。

ソケットオプション
値の取得と設定

```
int getsockopt(int s, int level, int optName,
               void *optVal, unsigned int *optLen)

int setsockopt(int s, int level, int optName,
               const void *optVal, unsigned int optLen)
```

s: socket()で割り当てられたソケットディスクリプタ
level: オプションのレベル(プロトコルスタックの層(レイヤ))
　　ソケットレイヤ自体で処理するならば、SOL_SOCKET
　　トランスポート層固有ならば、IPPROTO_TCP
　　ネットワーク層固有ならば、IPPROTO_IP
optName: オプションの種類
optVal: バッファへのポインタ(オプションの設定値が格納されている場所)
optLen: バッファのサイズ (getの場合は、サイズを格納するための変数へのポインタ)

ソケットオプションの変更
例）受信バッファサイズを2倍にする

```
int rcvBufferSize;
int sockOptSize;

/* デフォルトのバッファサイズを取得し、表示する */
sockOptSize = sizeof(rcvBufferSize);
if (getsockopt(sock, SOL_SOCKET, SO_RCVBUF,
                &rcvBufferSize, &sockOptSize) < 0)
  DieWithError("getsockopt() failed");
printf("Initial Receive Buffer Size: %d¥n", rcvBufferSize);

/* バッファサイズを2倍にする */
rcvBufferSize *= 2;
if (setsockopt(sock, SOL_SOCKET, SO_RCVBUF,
                &rcvBufferSize, sizeof(rcvBufferSize)) < 0)
  DieWithError("setsockopt() failed");
```

ソケットオプション（optNameに記述する定数）

オプション	データ型	値	説明
■SOL_SOCKETレベル			
SO_BROADCAST	int	0,1	ブロードキャストを有効にする
SO_KEEPALIVE	int	0,1	キープアライブメッセージを有効にする
SO_LINGER	linger {}	時間	close()が確認を待つための遅延時間
SO_RCVBUF	int	バイト数	ソケットの受信バッファサイズ
SO_RCVLOWAT	int	バイト数	recv()が処理を戻すまでに受信するバイト数の最小値
SO_REUSEADDR	int	0,1	すでに使用中のアドレス/ポートへのバインドを有効にする
SO_SNDLOWAT	int	バイト数	送信パケットのバイト数の最小値
SO_SNDBUF	int	バイト数	ソケットの送信バッファサイズ

ソケットオプション（2）（optNameに記述する定数）

オプション	データ型	値	説明
■IPPROTO_TCPレベル			
TCP_MAX	int	秒数	キープアライブメッセージの間隔
TCP_NODELAY	int	0,1	データをまとめるための遅延
■IPPROTO_IPレベル			
IP_TTL	int	0～255	IPパケットのユニキャストの存続時間
IP_MULTICAST_TTL	unsigned char	0～255	IPパケットのマルチキャストの存続時間
IP_MULTICAST_LOOP	int	0,1	自身がマルチキャストソケットで送信したパケットの受信を有効にする
IP_ADD_MEMBERSHIP	ip_mreq{}	グループアドレス	指定マルチキャストグループに宛てられたパケットの受信を有効にする
IP_DROP_MEMBERSHIP	ip_mreq{}	グループアドレス	指定マルチキャストグループに宛てられたパケットの受信を無効にする

シグナル

- ユーザによる割り込み文字入力や、タイマーの期限切れのような、特定のイベントの発生をプログラムに伝えるための仕組み。
- 発生したイベントは、そのときコードのどの部分が実行中であるかにかかわらず、シグナルによって非同期でプログラムに通知される。

シグナルの処理

- ４つのいずれかの方法で処理

  無視
  - プロセスはシグナルが送信されたことを感知しない。

  強制終了
  - OSによって強制終了される。

  処理関数実行
  - プログラムで指定されたシグナル処理関数を実行

  ブロック
  - プログラムがシグナルの受信を可能にするためのアクションを起こすまで、何の影響も受けないようにする。

算術エラーや、メモリ関連のエラーなども
イベントが発生するようになっている。

- UNIXでは、各種イベントの発生を知らせることができるように、さまざまなシグナルが用意されている。
  - 処理のデフォルトは、無視か、強制終了。

		デフォルト
• SIGALRM	アラームタイマーの終了	終了
• SIGCHLD	子プロセスの終了	無視
• SIGINT	割り込み文字の入力（CTRL-C）	終了
• SIGIO	ソケットのI／O準備完了	無視
• SIGPIPE	クローズしたソケットへの書き込み	終了

参考：kurohimeの　/usr/include/bits/signum.h

sigaction()

- シグナルのデフォルト動作を変更する

```
int sigaction(int whichSignal,
              const struct sigaction *newAction,
              struct sigaction *oldAction)
```

成功時0、失敗時-1を返す。

int型の引数を1つとる関数へのポインタ

```
struct sigaction {
  void     (*sa_handler)(int); /* シグナルハンドラ */
  sigset_t sa_mask;    /* ハンドラ実行中にブロックされるシグナル */
  int      sa_flags;   /* デフォルト動作を変更するためのフラグ */
};
```

sa_maskを設定する関数

- **int sigemptyset(sigset_t *set)**
  - **set**の全フラグをリセットする
- **int sigfillset(sigset_t *set)**
  - **set**の全フラグをセットする
- **int sigaddset(sigset_t *set, int whichSignal)**
  - **whichSignal**で指定したシグナル番号のフラグをセット
- **int sigdelset(sigset_t *set, int whichSignal)**
  - **whichSignal**で指定したシグナル番号のフラグをリセット

詳しいことは、オンラインマニュアルコマンドの **man** で
関数リファレンスを参照してください。

例題： SigAction.c

- yahikoで SigAction.c をコンパイル・実行してみよう。
- Ctrl-C を入力すると、停止する。
  - 停止の際、これまでと違った挙動が起こる。
  - Ctrl-Cで、SIGINTイベントが発生、その処理をInterruptSignalHandler()が行っている。

ちょっと改造

- InterruputSignalHandlerを
  以下のように変更してみるとどうなるか？

```
void InterruputSignalHandler(int ignored) {
  printf("Interruput Received.¥n");
  sleep(3);
}
```

実行を止められない！？

- Ctrl-C を入力すると、メッセージを表示して、3秒間スリープ（休止）。その後、元に戻る…
  - 停止できない？
- 止め方：　少なくとも二通りある。
  1. 別のターミナルを開いて、killコマンドで、該当するプロセス番号のプロセスを停止する。

     ps aux|grep ユーザ名 → kill プロセス番号
  2. 実行しているターミナルで、Ctrl-Z を入力して、実行を中断し、jobs でジョブ番号を確認して、kill %ジョブ番号　で停止する。

シグナルの処理中に複数のシグナルが発生したら？

- 改造したSigAction では、
  - SIGINTが発生すると、処理関数内で3秒休む。
  - その間に、3回Ctrl-Cを押すと、
    - 最初の1つがブロックされ（取っておかれる）
    - 残りの2つは破棄される
    - 関数の実行が終わると、システムは同じハンドラ関数をもう1回だけ実行する（ブロックしたシグナルの分）

  シグナルの動作を正しく理解して、プログラムを作らないと、アプリケーションの動作に不都合が生じる。

シグナルとソケット

サーバ（あるいはクライアント）から、
TCPソケットを介して接続が確立されているとき、

- 接続先のマシンが（クラッシュなどの理由で）予期せず接続をクローズしたら…
- プログラムは、
  - そのソケットから送信を試みてエラーが返されるか、
  - SIGPIPEが送信される

  までの間、接続が切れたことを知ることができない。

その接続に使っているリソースを開放する

SIGPIPEによるデフォルト動作は、「終了」
サーバの場合は特に、SIGPIPEの動作を変更すべき
（さもないと、クライアントの異常で、サーバがダウン）

ノンブロッキング I/O

- これまでの例は、要求された処理が完了するまで、プログラムの実行がブロックされる。
  - recv()は受信するデータが無ければブロックされ、send()は送信に必要なバッファ領域が足りなければブロックされる。
  - 接続関連の関数も、接続が確立されるまでブロックされる。
- 関数の完了を待っている間でも、ユーザの要求に対する応答のタスクは実行できるようにしたい

  → ノンブロッキング I/O

- ブロッキング動作を制御する方法
  - ノンブロッキングソケット
  - 非同期 I/O
  - タイムアウト

このへんの話は、ややこしいので、
ここでは表面的なことだけ紹介します。
詳しく知りたい人は、勉強してみてください。

ノンブロッキングソケット

- すべての関数呼び出しをブロックしないように、ソケットの設定を変更するのが最も簡単。
  - ノンブロッキングによるエラーは、errnoがEWOULDBLOCKに設定される。
- ブロッキングのデフォルト動作は、fcntl()で変更。
  - fcntl (file control) : あらゆるファイルの制御ができる

connect()は例外

```
int fcntl(int socket, int command, long argument)
```

フラグの設定	F_SETFL	フラグ名
フラグの確認	F_GETFL	0

ノンブロッキング
O_NONBLOCK

非同期 I/O

- ノンブロッキングソケットは、処理が成功したかどうかを定期的に確認する（ポーリング）ことが必要。
- 非同期 I/O では、SIGIOシグナルを利用して、ソケットで発生した I/O関連のイベントをプロセスに通知する。
  - イベントが発生するまでの間、別の処理を行うことができる。

タイムアウト

- 「特定のイベントが一定期間一度も発生しなかった」という情報が必要になるときには、
  タイムアウトを実装する。
  - 例えば、サーバからの応答が2秒間なかったら、タイムアウトしたものとして受信をあきらめるか、あらためて要求を送信する。といったアクションをとるようにプログラムする場合に必要。
  - タイマー関数 alarm() を利用する。
    - タイマーが切れると、SIGALRMシグナルが発生、ハンドラ関数が実行される。
    - **`unsigned int alarm(unsigned int secs)`**

マルチタスク

- これまでのサーバプログラムでは、一度に1つのクライアントしか処理しない。
  - あるクライアントが通信をしている間に、別のクライアントが接続した場合、接続が確立され、要求を送信できるようになるが、最初のクライアントの通信が終わるまでは、処理が始まらない。
  - このようなサーバは、「反復サーバ」と呼ばれる。
    - 処理が短時間で終わるような場合は、効果的
    - 負荷が大きくなると、待ち時間の許容限度超える
- マルチタスクOS
  - プロセスやスレッドの仕組で、解決。「平行サーバ」

クライアントごとにプロセス生成

- 同じホスト上で独立に動作するプログラムの一つひとつがプロセス。
- １つのクライアントからの接続要求に対して、新しいプロセスを1つ生成して通信に使用することができる。
- 新しいプロセスの生成には、fork() を使う。
  - fork は、スプーンとフォークの「フォーク」の意。
    - フォークの先端のように、プロセスが二つに分かれるようなイメージ

fork

- 呼び出し元とまったく同じプロセスを新しく生成
  - プロセスIDとforkから受け取る戻り値が異なる。
    - 親プロセスは、子プロセスのプロセスID、
      子は0を受け取る。
  - その後、2つのプロセスは独立して動作する。
- 子プロセスは、終了しても自動的には消滅しない。
  →「ゾンビ」
  - 親プロセスが、waitpid()を呼び出して、
    回収(harvest)するまでシステムリソースを消費する。

forkを使ったプログラム例

- TCPEchoServer-Fork.c
  - forkで子プロセスを生成して、子プロセスがクライアントとの通信を担当。
  - 親プロセスは、接続要求があるたびに、子プロセスを生成する。
  - 第1引数に待ち受けポート番号を指定。
- TCPEchoClient.c
  - 動作確認用、クライアント
  - 引数は3つ： サーバIPアドレス、送信メッセージ、接続ポート番号

課題

- TCPEchoClient.c を Windowsで使えるように修正してください。
  - Linuxでも使用できるように、ソース統合の形式で修正してください。
  - Teamsで課題を割り当てるので、ソースファイルをアップロードして、提出すること。

（Serverの方はLinux専用。課題の対象外です。）

プログラムの動作テスト

- yahikoは、ファイアウォールの設定により、TCPEchoServer-Forkを動作させておいてもyahiko以外からのTCPEchoClientのアクセスは受け付けません。
- Windows端末等からTCPEchoClientの動作テストをする場合は、テスト用サーバ IPアドレス：202.209.3.88　ポート：20000 にアクセスしてください。