

# ネットワーク開発実験(B1) アドバンストプログラミング演習 1 (A1) 2020 年度後期

担当教員：瀧本栄二，西村俊和

## 本科目の目標

本科目では、プログラミング演習 1・2、オペレーティングシステムなど関連科目で修得する内容を基礎として、下記に示すような一歩進んだプログラミング技法を修得する。

- ・ 基本的なシステムプログラミング（プロセス操作、シグナル 他）
- ・ ソケットプログラミング（ネットワークプログラミング）
- ・ マルチプロセスプログラミング
- ・ マルチスレッドプログラミング

## 成績評価の概要

本科目では、上記を修得するために 2 つの課題を全 11 ステップで進める。各自のスピードに応じて課題を順に完成していくこと。「全員が毎週同じ課題に取り組む形式」ではない。

内容	成績の目安（保証されたものではありません）
・ シェルの作成(a)～(f)	
・ TCP/IP を用いたチャットプログラム	- C -
・ HTTP の観測と理解	
・ Web サーバ作成 1（1 プロセス）と性能評価	- B -
・ Web サーバ作成 2（マルチプロセス）と性能評価	- A -
・ Web サーバ作成 3（マルチスレッド）と性能評価	- A+ -










## 進め方：

- ・ プログラムが完成したら、TA に対して「デモ」（プログラム実行による課題完成のアピール）をする。
  - 図（後述）を用いてプログラムの流れを説明。
  - 実際に動かしてデモンストレーションをする。
  - OK であればソースコードを manaba+R へ提出する。不可の場合は再チャレンジ。
  - デモは 1 回/授業までとします。計画的に。これは駆け込み防止用です。
- ・ 上記とは別に、定期的に教員が呼び出します。いろいろ技術的雑談をします。
- ・ 図やプログラムなど技術的なことについて簡単なプレゼンをしてもらうことがあります。
- ・ 動作していない課題プログラムは評価対象外です。

## プログラム完成のコツ&ルール

1. 「何を作るのか理解できていない」段階では絶対にプログラムを書かない。
  - ※ 「勘プログラミング」撲滅！！一行一行の意図を説明できるように！！
  - × 「こう書いたら動いたので・・・」
  - × 「こうしないと動かなくなっちゃうので・・・」
2. 処理の流れの図を作成すること。ノートと筆記用具を用いること。使用する言語は日本語等。処理の流れに確信を持てた場合にのみ次に進んでよい。
  - ※ 図はできるだけ詳細に書くこと。他人に説明できるレベルの綺麗さは重要。
  - ※ 詳細度については、理解度に応じて、大変おどろおどろしい図の作成をしてから、少しずつ詳細化していく（図を書き換える or 何個も書く）のも良い。
  - ※ 複雑なデータを扱うときはデータも図示すると良い。
  - ※ プログラムが苦手な者はここを丁寧にやると成長できる。

3. 日本語等で書いた機能を実現するのにどんな関数を使えば実現できるのかを調査する。
4. プログラムの構成（関数分割等）と作成順序について考える。
  - 何度も同じ処理が現れれば、そこを関数にする。
  - コンパクトな目的を持ったプログラムとして、関数を形成する。など
5. 以上が完了してはじめてプログラムを書くことが許される。
  - ※ 図とプログラムが矛盾することがないように、必要であれば随時図を更新すること。
  - ※ 図に間違いが見つかった場合、その内容を改善するために随時更新してよい。
  - ※ プログラムを美しく書くこと。インデント（字下げ）、空行、コメントを大切に。
  - ※ 自然なプログラミングを心がけること。
    - 関連のある行は、近くを書く。（例：演算と演算結果の条件確認 if など）
    - 同じような条件分岐はまとめる。
6. 質問をする際には、図のどの部分がうまくいかないのかについて説明し、それがプログラムだとどうなるのかを説明すること。
  - ※ 図の無い質問は一切受け付けない。
  - ※ 図のできが悪い場合はプログラミングの中断と図の再作成を指示する。
  - ※ 図が無い・図が不十分な場合はデモは NG とする。

	<b>端子</b> フローチャートの始まり及び終わりを表す		<b>ループの開始</b>
	<b>処理</b> 計算、代入などの処理を表す		<b>ループの終了</b>
	<b>サブルーチン</b> 定義済みの処理を表す		<b>入出力</b> ファイルへの入出力を表す
	<b>判断</b> 条件による分岐を表す		<b>ページ内結合子</b> フローチャートが長くなり、ページ内で二列にするとときなどに使う。
	<b>表示</b> コンソール上への結果の表示を表す		<b>ページ外結合子</b> フローチャートを次のページに続けるときに使う。

## 学習行為と不正行為

- ・ デッドコピー（ソースコードやレポートなど授業評価物の剽窃）など学習を妨害・放棄する行為、大学の教育・評価機能を妨害する行為、大学・構成員の信頼を低下させる行為は禁止（不正行為）です。
- ・ 授業評価物の全部または一部を他者に譲渡すること（評価物電子データや表示のコピー、評価物を見せながらのタイピング許可、管理不行き届きによる盗難被害）も不正行為に該当します。
  - 剽窃希望者が「あとでちゃんと自分で作るから剽窃にならない」などと騙って評価物を要求することがありますが、譲渡側も剽窃に加担したとして同様の処分ありますので、くれぐれもご注意ください。不正行為による処分は当該教科単位不認定に限らず当該セメスター全科目不認定あるいは停学などあり得ます。
  - プロジェクト型課題ではないので、本科目課題作成そのものについて共同学習をすると不正行為となります。
- ・ 不正行為でない他の人との議論や共同学習は歓迎します。
  - 新しく出てきた関数の仕様・使い方について議論する。
  - 処理の高度化・改善策について議論する。
  - プログラム動作不良部分について、デバッガなどで原因を追及する。

上述三点以外は不正行為となる可能性があるのでご注意ください。

## 0-A. eclipse

起動はコマンドラインから eclipse と入力。

初回のみ：

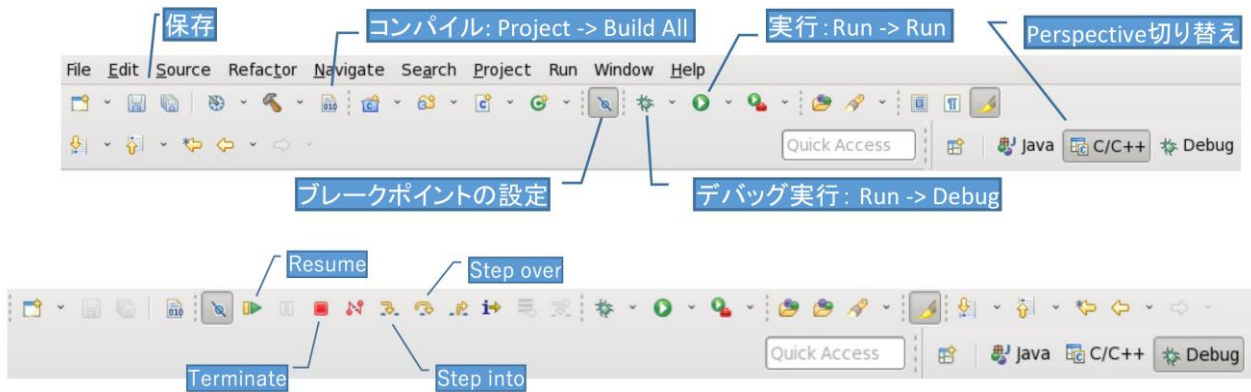
1. 「Select a workspace」と言われたら、デフォルト値のまま、Use this as the default and do not ask again にチェックして OK をクリック。
2. 「Welcome to Eclipse」では、左上にある「Welcome」タブの×をクリック。

新しいプログラム作成開始時(次の課題へ進めるとき)・・・プロジェクトの作成：

1. File→New→Project。ダイアログで「C/C++」内の「C Project」を選択し Next
2. Project name ではプログラム名を入力。「shell-a」など。
3. Project type は「Executable」内の「Hello World ANSI C Project」
4. Toolchains は Linux GCC で良い。
5. 右下 Finish をクリック。

初回のみ：「Open Associated Perspective?」では Remember my decision にチェックし Yes。

コンパイル・実行・デバッグ：



練習：

1. プロジェクト test1 を作成し、テンプレートで出てくる Hello world を動作させなさい。
2. 15 行目の puts にブレークポイントを設定し、デバッグ実行させなさい。15 行目で停止した後は、step over 実行（メニューの下にアイコンあり）しなさい。

## 0-B. デバッガの使い方 (gdb)

概略：

1. デバッグ対象のプログラムは「デバッグ情報(シンボルテーブル)入り」でコンパイルする。  
`% gcc -g -o prog prog.c` (「-g」が重要)
2. gdb (GNU Debugger) を起動する。  
`% gdb ./prog` (gdb の引数でデバッグ対象の実行ファイルを指定する)  
以降、プロンプトが「(gdb)」になる。
3. 使う。コマンド一覧参照。
4. 終わるときは、^d か quit コマンド。

コマンド一覧：

1. ソースコードの表示

1 行番号 (list の省略形。行番号も省略可)

## 2. ブレークポイント

### (a) ブレークポイントの設定

ブレークポイントとは、プログラムの実行を中断する位置のこと。

b 行番号または関数名 (break の省略形)

(例) b 8 … ソースコードの 8 行目実行前を指定

(例) b myfunc … myfunc 関数が呼び出されたとき (実行前) を指定

### (b) ブレークポイントの一覧表示 (ブレークポイント番号を調べるときも)

i b (info breakpoints の省略形)

### (c) ブレークポイントの削除

d b ブレークポイント番号 (delete breakpoints の省略形)

cl 行番号または関数名 (clear の省略形)

## 3. デバッグ対象のプログラムの実行

r (run の省略形)

r 引数 1 引数 2 … (デバッグ対象に引数を渡したい場合)

ブレークポイント指定箇所が実行されようとする、その直前にプログラムが中断され、gdb のプロンプトが表示される。

## 4. 1 行単位でのプログラム実行 (実行中断状態のときのみ使用可)

(a) プログラムを 1 行実行する。次の一行が関数呼出しの場合、呼び出して戻ってきた状態で実行中断する。

n (next の省略形)

(b) プログラムを 1 ステップ実行する。次の一行が関数呼出しの場合、呼出し先関数の中を 1 行実行した状態で実行中断する。

s (step の省略形)

## 5. 中断されたプログラムを再開する (次のブレークポイントまたは終了まで)

c (continue の省略形)

## 6. 変数の値の表示

p 変数名 (print の省略形)

(例) print pointer … 変数の内容を表示する。

(例) print \*pointer … 変数をポインタと解釈し、変数が指す先の値を表示。

## 7. ヘルプ

h (help の省略形)

h コマンド名

## 8. 終了

q (quit の省略形)

^d

## 1. 第1課題 シェルの作成

### (a) シンプルなシェルを作成しよう。

仕様の概要：

- ・ ユーザに入力を促すプロンプトを表示すること。
- ・ 標準入力からコマンド入力を受け付け、**内部コマンド**または**外部コマンド**を実行する。
- ・ 内部コマンドは、「exit」と「quit」の2つとし、これらが入力されるとシェルを終了する。
- ・ それ以外のコマンドは外部コマンドとして処理すること。  
例：「/bin/ls」と入力するとls コマンドが、「/usr/bin/emacs」なら emacs が起動すればよい。
- ・ ここでは**フォアグラウンド実行**のみとする。
- ・ コマンドの実行が終了後は、上記を繰り返し受け付けるようにすること。

※ 「emacs prog1.c」のようなコマンド起動時の引数指定は 課題(b)で実現する。

※ 「emacs &」のような「バックグラウンド実行」とその操作は課題(c)(d)で実現する。

※ ^C ([control]+[c]) が入力されるとシェル自体が終了する問題は課題(e)で解決する。

※ 「emacs」のようにファイル名を入力するだけでusr/bin/emacs が起動されるサーチパスについては課題(f)で実現する。

用語：

- ・ 内部コマンド：入力されたコマンドに対応する機能を、シェル自体が実現する形式。
- ・ 外部コマンド：入力されたコマンドをファイル名として捉え、そのファイルを実行する形式。
- ・ フォアグラウンド実行：別プロセスとして実行されるコマンドが実行終了してからプロンプトが表示される方式。シェルはそのプロセスの終了を待たなければならない。
- ・ バックグラウンド実行：シェルが、別プロセスとしてコマンドを実行させておき、その終了を待たずにプロンプトを表示させる方式。

キーとなるシステムコール・関数：

- ・ fork() … プロセスが自分（親プロセス）の複製（子プロセス）を作る
- ・ execv() … プロセスが自身を指定したプログラムに置き換える
- ・ wait() … 子プロセスが終了するまで待ち、子プロセスの終了ステータスを受け取る
- ・ exit() … プログラムを終了する。引数は終了ステータスとして親プロセスへ

ポイント：

- ・ man 等のオンラインマニュアルや参考書を見ながら、自身で関数の機能、引数、戻り値を確認しながらプログラミングするというスタイルの確立ができるか。
- ・ プロセス（動いているプログラム）が別のプロセスを起動する流れ、プロセスの終了を待ち終了ステータス（exit status）を受け取る流れが理解できるか。
- ・ 親プロセスと子プロセスの概念を理解できるか。

ヒント：

- ・ シェルは、外部コマンドを別プロセスとして起動する。このような場合、(1) 別プロセスを生成 (fork 関数) し、(2) その別プロセスを外部コマンドに変化させる (execv 関数)、という流れをとる。
- ・ 課題(a)ではコマンドラインの引数を扱わない。この場合 execv()の第一引数は外部コマンドのパス名（ここでは char cmd[サイズ];で定義されているとする）と、第2引数で「コマンドラインの引数が無い」ことを示さなければいけない。このときの第2引数は次のように定義して、execv(cmd, pargs);で良い。

```
char *pargs[2] = {cmd, NULL};
```

- ・ 親プロセスは、fork()で子プロセスを作成した後は、必ず「終了ステータス」を受けとらないとい

けない。ただ、終了を待って受けとる（同期型）か、何度も状態を確認して終了していたときのみ受けとる（非同期型）かを選択できる。今回は同期型にしたいので、`wait()`を使う。

(b) コマンドラインで引数を取れるようにしよう。

#### 仕様の概要：

課題(a)をベースに、コマンドラインに引数を取れるようにせよ。例えば、「`/usr/bin/emacs prog1.c`」のようにコマンド起動時に引数を指定できるようにせよ。具体的には次の3つの処理を追加すれば良い。

- ・ 入力されたコマンド文字列を空白で区切る処理
- ・ 区切った文字列へのポインタを配列としてまとめる
- ・ `execv` の第2引数として指定する

#### キーとなるシステムコール・関数：

- ・ `execv()`

#### ポイント：

- ・ 1つの文字列を特定の文字で区切りつつ、文字列の配列を生成できるか。  
どのような処理をすればいいのか、図示できるか。  
メモリ配置をイメージできているか。
- ・ 上記をプログラムとして表現できるか。

#### ヒント：

- ・ `execv` の第2引数はポインタの配列(`char *p[サイズ]`)である。各要素のポインタの先には文字列が格納されている。`p[0]`の先には `execv` の第1引数と同じ文字列（コマンド）、`p[1]`の先にはコマンドラインの第1引数、…、そして、最後は `p[n]=NULL` とする。

(c) バックグラウンド処理を実現しよう

#### 仕様の概要：

課題(b)をベースに、コマンドの最後に「&」が付加されたときは、「&」を引数とせず）コマンドをバックグラウンド（BG）実行するようにせよ。

- ・ BG 実行は、コマンドが実行中もシェルのプロンプトが表示され、次のコマンド入力が可能となるような実行方法である。
- ・ BG プロセスが終了したときにはその終了ステータスを取得すること。ただし、「たまに（プロンプトを表示する前くらいのタイミングで）終了した子プロセスの有無をチェックする方式」とすること。シグナルによる方法は課題(e)とする。
- ・ 最後の引数が「&」ではない時は、フォアグラウンド（FG）実行とする。

#### キーとなるシステムコール・関数：

- ・ `wait()` … どの子プロセスを待つのか意識（指定）せずに終了ステータスを受け取る。どれか一つの子プロセスが終了するまではブロックされる。
- ・ `waitpid()` … 指定した子プロセスが終了するまで待ち、子プロセスの終了ステータスを受け取る。指定していないプロセスは対象としない。オプションの指定により、全子プロセスを対象とすることもできる。また、指定子プロセスが終了していない場合にブロックされないようにすることもできる。

#### ポイント：

- ・ FG 実行の場合、従来通り子プロセスの終了を待っているか。

FG プロセスを待っている間に、別の BG プロセスが終了したような場合に、間違えてブロックが解除されないか。

- ・ いつ終了するかわからない BG プロセスの終了コードもちゃんと取得できているか。

#### (d) バックグラウンド実行しているプログラムをフォアグラウンドに変える

##### 仕様の概要：

課題(c)をベースに、内部コマンドとして「jobs」と「fg」を追加する。

jobs	…	BG 実行されているプロセス一覧を表示する (※)
fg	…	BG 実行プロセスのうち、指定した 1 個 (※) を FG に切り替える

(※) jobs コマンド時に適宜番号 (通し番号やプロセス ID など) をつけて表示させるようにする。fg コマンドではその番号を指定して FG に切り替えるようにする。

##### キーとなるシステムコール・関数：

- ・ 特になし。

##### ポイント：

- ・ BG 実行しているプロセスの情報を管理できているか。
- ・ FG 実行を指定したプロセスの終了を待つことができているか。別のプロセスが終了した場合に、ブロックが解錠されてしまうことがないか。

##### ヒント：

- ・ 「プロセスを FG に切り替える」ということは「FG のプロセスの終了を待つ」ということである。

#### (e) シグナルに対応する

##### 仕様の概要：

課題(d)までのシェルでは ^C が押されるとシェルが終了してしまう。また、BG プロセスが終了したときに即座に終了ステータスを取得できていない。次の機能を実現して、これを解決せよ。

- ・ プロンプトで入力待ちの時に ^C を押してもシェルが終了してしまわないようにする。
- ・ FG 実行している子プロセスがあるときに ^C を押すと、その子プロセスを終了させる。
- ・ BG 実行している子プロセスが終了したら、即座にその終了ステータスを取得する。

##### キーとなるシステムコール・関数：

- ・ signal … 指定したシグナルを受信したときに起動する関数を登録する
- ・ kill … 指定したプロセスへ、指定したシグナルを送る
- ・ setpgid … プロセスグループの設定

##### ポイント：

- ・ 仕様で上げた 3 つの機能が実現されているか。

##### ヒント：

- ・ ^C が押されるとシグナル SIGINT が送られてくる。
- ・ 子プロセスが終了すると、SIGCHLD が送られてくる。
- ・ 一般的に、親プロセスと子プロセスは同じプロセスグループに属する。何もしないと、子プロセスに送られた SIGINT は親プロセスにも送られるため、子プロセスで ^C を押しただけなのに親子共々終了してしまう。これを避けるには、子プロセスを別のプロセスグループにすれば良い。今回の例では setpgid(0, 0)を使えば良いが、man 参照のこと。

(f) 環境変数 PATH（サーチパス）に基づいたコマンド検索機能を付加しよう

**仕様の概要：**

課題(e)では、外部コマンドはフルパス（full path）を指定しなければならない。これでは大変不便である。そこで利用者が設定した環境変数 PATH を取得し、そこに指定されたパスを検索する機能を付加せよ。例えば、PATH に/bin や/usr/bin が設定されているとき、「ls」や「emacs」と入力するだけで外部コマンドが実行できる機能である。

**キーとなるシステムコール・関数：**

- ・ getenv … 環境変数を取得する。

**ポイント：**

- ・ 環境変数を正しく取得・解析できているか
- ・ 設定された複数のパスを全て検索できているか
- ・ 設定された順にパスを検索できているか
- ・ 環境変数を変更すれば、プログラムを再コンパイルすることなく、上記の処理ができるか。



## 2. 第2課題 Web サーバの作成

第2課題では、Web サーバの作成を通じて、ソケットプログラミング、マルチプロセス・マルチスレッドプログラミングを体得する。

### (a) 単純なサーバ・クライアントモデルのソケット通信プログラムを作成しよう

仕様の概要：

配布している参考資料のうち18章ソケットプログラミングのものを熟読し、TCP/IPを用いて簡単なチャットプログラムを作ろう。なお、TCP/IP自体は18.5節で説明されているが、これを理解するためには18章の最初から読むこと。

- ・ サーバプログラムとクライアントプログラムを作る。
- ・ サーバプログラムは4000番ポートで待ち受け、クライアントがそこに接続を試みる。
- ・ 接続後、(1) クライアントが文字列を入力・サーバ側はそれを出力、(2) サーバ側が文字列を入力・クライアント側でそれを出力、以降(1)と(2)を繰り返すような簡単なチャットプログラムを作成する。
- ・ どちらかで「quit」と入力された場合両方のプログラムが終了するようにすること。
- ・ 近く受講者と相互に接続できるか確認すること。互いに server/client を入れ替えてのテストもすること。

キーとなるシステムコール・関数：

- ・ socket()、setsockopt()、bind()、listen()、accept()、connect()
- ・ read()、write()
- ・ その他 htons()、memset()など

ポイント：

- ・ INET ドメインのソケットが作成できているか。
- ・ struct sockaddr\_in の変数がちゃんと初期化できているかどうか。
- ・ サーバ側は bind、listen、accept の手順、クライアントは connect が、正しく行われ、接続できているか。
- ・ チャット機能が実現でき quit で終了できるか。
- ・ server も client もどちらも他者との接続に耐えられるか。

ヒント：

- ・ IP アドレスは「/sbin/ifconfig eth0」で調べられる。二行目 inet addr の後方を参照。

### (b) Web ブラウザの要求を見て HTTP を理解する

仕様の概要：

この後、Web ブラウザをクライアントとしてコンテンツを提供する Web サーバを作成したい。そこで、課題(a)を元に次のような動作をするサーバプログラムを作る。

- ・ TCP/8080 で待ち受けるサーバプログラムとする。
- ・ サーバは、送信されてくるデータを表示するだけでよい。

さらに、ブラウザがどのようなリクエストをしてくるのか確認すること。

- ・ ブラウザからは http://localhost:8080/ で接続する。
- ・ 送られてくるデータの内容を Wikipedia の HTTP のページを参考に理解すること。メソッドとヘッダの意味・役割の違いも理解すること。
- ・ この後、送られてくるデータを解析するので、フォーマット(書式・区切り記号など)も確認すること。

- ・ 改行コードが CR/LF (キャラクタコード 0x0d、 0x0a) の 2 文字で表されていることを確認すること (必要であればサーバプログラムを改良せよ)。

#### キーとなるシステムコール・関数：

- ・ なし

#### ポイント：

- ・ ブラウザから送信されてくる文字列 (「GET」で始まる行、およびその後ろに続くヘッダ) が表示されているか。
- ・ クライアントから送られてくるデータを、文字列として表示するだけでなく、文字コードとしても確認できるようになっているか (特に CR/LF)。

#### ヒント：

- ・ bind 関数でエラー (Address already in use) が発生する場合は、ポート番号を 8080 以外にしてテストしてもよい。ただし、きちんと close するプログラムを書くとこのようなエラーはほぼ発生しないので、プログラムを改良しよう。

### (c) シンプルな Web サーバを作ろう & 性能評価をしよう

#### 準備：

まずは、Web コンテンツを準備する。

- ・ ホームディレクトリに移動後、コンテンツへのシンボリックリンクを作成する。

```
% ln -s ~tnt/eind/htdocs . (ピリオドに注意)
```

#### 仕様の概要：

課題(b)を元に、最も基本的なメソッドである GET に対応できる Web サーバを作成しよう。

- ・ TCP/8080 で待ち受ける。
- ・ (※) クライアントからの接続を受け付け、リクエストを受信する。
  - リクエストは「GET / HTTP/1.1」のような形式で送られてくる。重要なのは第 1 引数 (この例では「/」) で、ここがいわゆる「パス」となる。
- ・ GET メソッドで指定されたパスに基づいて送信ファイルを決定する。
  - コンテンツ用ディレクトリ htdocs のフルパスが下記だとする (適宜修正のこと)
 

```
#define SERVER_ROOT "/homer/is/00/is0000aa/htdocs"
```
  - GET メソッドで指定されたパスから実際に送信すべきファイル名生成する。  
SERVER\_ROOT の後ろにパスを付けたして、"/homer/is/00/is0000aa/htdocs/"
  - 生成したファイル名が
    - A) 通常ファイルだった場合：そのファイルを対象とする。
    - B) ディレクトリだった場合：index.html が省略されているとして、ファイル名を生成しなおす。"/homer/is/00/is0000aa/htdocs/index.html"
- ・ 生成したファイルの有無を確認し結果を送信する。
  - ファイルがあった場合
 

```
HTTP/1.0 200 OK<CR><LF>
<CR><LF>
HTML コンテンツ (ファイルの内容)
```
  - ・ ファイルがなかった場合：
 

```
HTTP/1.0 404 Not Found<CR><LF>
```
- ・ クライアントとの接続を終了後、再度クライアントからの接続を受け付ける (※に戻る)。

### キーとなるシステムコール・関数：

- ・ `stat()` … 指定したファイルの詳細情報を取得できる。関連するマクロ `S_ISREG()` と `S_ISDIR(m)` も便利である。

### ポイント：

- ・ 下記の URL は最低限試してみること。結果も確認すること。
  - `http://localhost:8080/` “A Plain Web Content” というページが表示される。
  - `http://localhost:8080/exist` It works! と表示される
  - `http://localhost:8080/exist/` It works! と表示される
  - `http://localhost:8080/empty` 404 エラー
  - `http://localhost:8080/empty/` 404 エラー
  - `http://localhost:8080/img/logo.gif` 西村・瀧本研究室ロゴが表示される
- ・ `http://localhost:8080/` は、ロゴなど正しく表示されているか。
- ・ ブラウザで何度かリロードしてもちゃんと動作するか。

### ヒント：

- ・ 困難な課題ではありません。おおよそ 240 行程度で書けました（コメント・空行込み）

## 性能評価をしよう

### 準備：

`httperf` という性能評価プログラムで自作 Web サーバの性能を計測する。次の要領でシンボリックリンクを作成する。

```
% ln -s ~tnt/eind/httperf* . (ピリオドに注意)
```

`httperf` が計測プログラム、`httperf.man` がマニュアル（テキストファイル）である。ソースコードは、`~tnt/eind/src-httperf-0.9.0.tar.gz` として置いてある。

### 手順：

- ・ 作成した Web サーバプログラムを起動する。
- ・ 基本的には次のような入力で良い。必要があれば、マニュアルを参照すること。
  - `% ./httperf --port=8080 --num-conns=1000`
- ・ スループット (`conns/s` または `req/s`) を確認（記録）する。
- ・ Reply status を参照し、`2xx` が `--num-conns` で指定した値と同じであることを確認すること。異なっている場合はバグの可能性があるので、プログラムを見直すこと。

## (d) マルチプロセスサーバを作ろう & 性能評価をしよう

### 仕様の概要：

課題(c)をベースにマルチプロセスで動作する Web サーバを作ろう。

#### <親プロセス>

- ・ TCP/8080 で待ち受ける。
- ・ (※1) クライアントからの接続を受け付ける。
- ・ クライアントからの接続があるとそれを処理するための子プロセスを生成し※1 へ戻る。

#### <子プロセス>

- ・ リクエストを受信、GET メソッドで指定されたパスに基づいて送信ファイルを決定、ファイルの有無等を確認、結果を送信。
- ・ 終了。

キーとなるシステムコール・関数：

- ・ なし

ポイント：

- ・ 課題(c)のポイントと同様のテストをクリアできるか。
- ・ `fork` のタイミングは適切か (`fork` までにすべき処理と `fork` 後にすべき処理を適切に設計できているか)
- ・ 子プロセスの終了ステータスをもれなく取得できるようになっているか。
- ・ 課題(c)同様の性能評価をし、課題(c)との比較と考察ができているか。

ヒント：

- ・ 子プロセスの終了ステータス取得にはシグナル (`SIGCHLD`) を補足して処理するのが良い。

もう一つの形、プロセスプール方式：

この課題では、クライアントからの接続がある毎にプロセスを生成・終了していた。マルチプロセスサーバでもプロセスプール方式による実装方法もある。

- ・ 親プロセスは、事前にリクエスト処理用の子プロセスを複数起動 (プール) する。
- ・ クライアントからの接続があると、子プロセスのうち一つだけがそれを受信し、リクエスト処理をする。処理が終わると再度クライアントからの接続を待つ。
- ・ 親プロセスは、子プロセスの監視をもっぱらの仕事とする (子プロセスの異常終了時の再生成、定期的な子プロセスの終了・再生成など)。

(e) マルチスレッドサーバ (スレッドプール方式) を作ろう & 性能評価をしよう

仕様の概要：

課題(d)をベースに、マルチスレッドで動作する Web サーバを作ろう。今回はスレッドとして `pthread` を使う。また、スレッドプールと呼ばれる手法を使って実装する。スレッドプールは、事前にスレッドを複数生成して待機させておき、クライアントから接続が来たらスレッドの一つに処理をさせる手法である。課題(d)ではクライアント接続後に `fork` したがそれとは異なる流れとなる。

<親スレッド>

- ・ ソケットを生成し、接続受付の直前 (`listen`) まで準備する。
- ・ 子スレッドを生成する。個数は 4~16 個程度。
- ・ 生成後は、子スレッドと同様な処理をすることとする (下記の※へ)。

<子スレッド>

- ・ (※) 接続を受け付ける。
- ・ リクエストを受信、`GET` メソッドで指定されたパスに基づいて送信ファイルを決定、ファイルの有無等を確認、結果を送信。
- ・ 再度※に戻る。

キーとなるシステムコール・関数：

- ・ `pthread_create()`、`pthread_detach()`、`pthread_self()`

ポイント：

- ・ 課題(c)のポイントと同様のテストをクリアできるか。
- ・ `pthread_create` のタイミングは適切か。スレッドプールが実現できているか。
- ・ 課題(c)同様の性能評価をし、課題(c)との比較と考察ができているか。

ヒント：

- ・ 子スレッドが **accept** で使用するソケットディスクリプタ(**sd**)は、親スレッドが **listen** 等で使用した **sd** で良い。複数の子スレッドが一斉に同じ **sd** で **accept** 待ちをすることになる。これによって OS がその中のどれかに接続を割り当ててくれる（どれか 1 個の **accept** 要求に対してのみ結果を返してくれる）ため、**accept** 待ちしていたスレッドの一つだけが接続を受け付けることができる。
- ・ **pthread** では、親スレッドが子スレッドの終了ステータスを得る手法のバリエーションが乏しい。そのため、今回は子スレッドを **detach** された状態で実行し、終了ステータスの受信はしなくてもよいこととする。
- ・ マルチプロセスでは、**fork** 後は、親と子間、兄弟間で変数の値は共有されない。しかし、マルチスレッドでは、スタック領域に格納される自動変数以外はスレッド間で共有される。グローバル変数や **static** 変数を使うときは、他のスレッドがその変数を使う（競合してしまい、意図せず互いに影響を与えてしまう）可能性を十分考慮すること。

## さらに高度な内容にむけて

### 性能評価：

これまでは単純に性能を計測しただけでしたが、次のようなことをやってみるとどのような結果になるでしょうか？

- ・ 転送されるファイルの大きさを変えたとき。
- ・ 転送を **localhost** ではなく、別のコンピュータからネットワーク越しに行ったとき。

性能評価は、単なる動作テストではありません。そのプログラムが設計思想を満たしているか、どのような条件でどれくらいのパフォーマンスがでるのかを客観的に数値で示すためのものです。数値で示すことで、技術的・学術的には手法がすぐれていること明確に示すことができ、論文等の実績を得ることができます。商品であればパンフレットに掲載して顧客に売り込みができるようになります。新サービスならば投資家の説得にも使えるでしょう。社長に対して予算要求するための説得材料にもなります。

### プログラム技術：

#### <シェル>

- ・ パイプ機能を実現せよ。ls -al | grep b | sort | uniq などができるように。

#### <Web サーバ>

- ・ マルチスレッドで、**epoll** を使った実装をしてみましょう。
- ・ HTTP/1.1 では、リクエスト毎にコネクションを切らないようにできます。すなわち、各コネクションで、複数のリクエストを連続的に処理できるように拡張してみましょう。
- ・ 上記の **epoll**、複数リクエスト対応を実装したとき、性能はどうなるでしょう。

以上