# 今ならきっともっとよく分かる，「問題追求 ( デバッグ ) の仕方」

# よくある光景

- サーバーを立てた
- クライアントを立てた
- 通信して結果が表示されると思ったのに表示されない
- まてどくらせどプログラムが終わらない
- 「すいません , 動かないんですが ... 」

# これまでに見かけた「落ち」

- /dev/dsp から 100 万バイト読もうとして異様に時間がかかる (100 万 /8000 = 2 分 )
- 接続先がないときに connect に異様に時間がかかる ( 数分後 timeout)
- EOF の判定方法を間違っており , 無限ループ (while (EOF == read(...)) …)

# 重要なこと

- 最終的な落ちが重要なのではない！
  - それらを「知らなかったこと」を問題にしてもキリがない
  - これからもたくさん知らないことは出てくる
  - 知ってても忘れる , うっかりする (if (a = b) ...)
  - API の挙動についてすべて理解してからプログラムを書くわけでもない
- 大事な事は「おかしな挙動をするプログラム」を直す過程・方法

# プログラムが間違っている時の現象

- エラーで終了 (segmentation fault など )
- 出力が間違っている
- 終わらない ...
- 
- 基本的にはこれ「しかない」

# 大事な事

- この状態では「落ち」を知らない ( 当たり前 !)
- 「落ち」に応じて取るべき手段が変わるような方法は , 方法とは言えない
- ( 誤解を恐れず言えば ) この状態から「常に同じ方法」で原因を求めることが問題追求 ( デバッグ ) の正しい方法

# 常に取るべき方法とは？

- 現在のプログラムの「どこで何が起きているか」をよく把握する ( 調べる )
  - 自分： 医者
  - プログラム： 患者
- 標語： 直すより「調べる」

# まずい心理状態：「なぜ動かないんだっ！」

- 特にまずい状態
  - さっきまで動いてたのに！
  - このファイルなら動くのに (/dev/dsp にすると動かない )!
  - 家では動いたのに！
  - 俺は悪くない !!

# そうは言っても具体的には？

- もちろん「調べる手段」は何通りもあるのだが，現在の実験レベルでは以下の二つを使えれば十分
- printf 文を「適所に」はさむ
- デバッガで実行がどこまで行っているか調べる

# 例題

- 例えば以下のプログラムがどういうわけか終了しなかったとする
- s = API_X( … );
  API_Y(s, ...);
  while (1) {
    n = API_Z(s, data, N);
    if (n == 0) break;
  }

# これが 3 分間待っても終わらないとしたらどんな可能性が考えられるか

- 当たり前だが処理系にバグがあって全く上のコードと関係ないことをやっている場合をのぞき , 以下の 3 とおりの可能性「しかない」
  - API_X に異様に時間がかかっている
  - API_Y に異様に時間がかかっている
  - API_Z に異様に時間がかっている
  - while 文が異様にたくさん回っている
- 「このうちのどれが起きているのだろう」と探るのがデバッグ

# printf 挿入

- API の前後をサンドイッチ
  - API_X(x); →
    ```
    printf(”-> API_X(%d);\n”, x);
    r = API_X(x);
    printf(”%d <- API_X(%d);\n”, r, x);
    ```
- ループの先頭で進捗を表示
  - while (…) { … } →
    ```
    c = 0;
    while (…) {
      printf(”%d th iteration\n”, c++);
    }
    ```

# よい心がけ

- たかが printf されど printf → 表示される内容を丁寧に , 意味のあるものに
  - 関数の引数や返り値を表示するのはよい心がけ
- ループでは毎回異なる文字列を
  - カウンタで何回目の繰り返しかを表示するのはよい心がけ
- 要するに ,

× printf(”aaaaaaaaa\n”);

# 注意：

- printf には改行を
- 念をおすなら
  - printf(”....\n”);
    fflush(stdout);
- 標準出力が端末の (>  を使っていない ) 場合 :
  - 改行をしないとその場で表示されない . 改行すればされる
- 標準出力がファイルの (> file) 場合
  - fflush するか , ある程度 ( 例えば 4KB) たまるまで書かれない

# さらによい心がけ

- if(dbg) { printf(...); }
- dbg=0/1 だけで表示を ON/OFF できる
- 一旦動き出しても printf を消さずに必要に応じてすぐに復活できる

# (リーダ's howto風に) 最後にこれだけはもう一度言っておく

- 後から printf を差し込むくらいなら以下くらいは最初からやっておく
  - s = API_Z(); →
    s = API_Z();
    if (s == -1) { perror(”Z”); exit(1); }
- これだけで「 API_Z() がおかしな値を返していないことの確認」になっている
- この一言で救われるバグがどれだけ多いか

# デバッガを使うなら (1)

- プログラムが短ければ , ステップ実行 (next コマンド ) を叩き続けるだけで , どの API_?() が異様に時間がかかっているか一目瞭然
- ある一連の文 ( 例えばループ ) が終了していることを確かめたければブレークポイント
  - s = API_X( … );
    API_Y(s, ...);
    while (1) {
      n = API_Z(s, data, N);
      if (n == 0) break;
    }

# デバッガを使うなら (2)

- while 文の無限ループは ,5000 回回ってちゃんと終了するものと , 無限ループしているものをステップ実行で追うのは辛いかもしれない
  - printf でよい
- デバッガを使った乱暴な方法
  - 「走らせて ctrl-C で止める」を何回か繰り返す
  - いつもある API 中で止まっていたらそれが怪しい
  - いつもあるループ中のどこか ( 自分のコード ) で止まっていたらそのループが無限ループしているかもしれない

# 個別の「落ち」に対する解説

- connect(...) に異様に時間がかかる
- 「存在しないアドレス + ポート」に接続しようとしたときの挙動に何通りかある．決めつけてはいけないが大雑把には
  - そのアドレスに IP パケットはとどいたがそのポートでは誰も待っていない → すぐにエラー (connection refused)
  - 宛先が同一サブネット内で , その IP アドレスのマシンが LAN 内に存在しない → 割とすぐにエラー (no route to host)
  - 宛先が遠くでそこまで届かない → 異様に時間がかかったのちにエラー (connection timeout)

# read に異様に時間がかかる

- /dev/dsp を 100 万バイト読もうとしたら ...
- 100 万 /8000 = 125 秒
- 厄介なのは「普通のファイルだとうまく行くのに /dev/dsp にしたら動かなくなった！」という怪現象としてこれが映ること ( 悪いのは俺じゃないシンドローム )

# EOF に関する誤解

- データの終わりに達したら read は 0 を返す．より具体的には，
  - ファイルを最後まで読みきった後の read
  - 接続相手が close を発行し，かつ彼が close 以前に送ったデータをすべて自分が受け取った後の recv(=read)
- これは単に「もうあなたが受けとるべきデータはない」ということを示しているのであって，EOF というデータが実際に読み込まれているわけではない（ファイル中に EOF というデータ（文字）が入っているわけではない）

# 誤解の原因

- 「 EOF を返す」という比喩的な言い方が誤解を招く
- それだけではなく , ある種の API, fgetc(), getchar() などは実際に , データの終わりに達したことを示すのに EOF( 実は -1) という特別な文字を返り値として返す
  - この場合も EOF などというデータが読まれているわけではない
  - 単に ,API によって「終わりに達した」ことの伝え方に色々ある , ということに過ぎない

# 余談：なぜ getchar() は EOF を返す？

- getchar() は「次の一バイト」を返す関数
- 終端に達した場合も「何か」を返さなくてはしょうがない
- 当然その「何か」は普通の一バイトとかぶってはいけない
  - 0 もファイル中に現れる文字の一種だから 0 を返すわけには行かない
  - 要するに 0, ..., 255 以外の値でなくてはいけない
  - そこで -1 が選ばれたというだけの話

# getchar() は int を返す

- getchar() の響きから , これが char ( または unsigned char) を返すと誤解しがち
- しかし , 考えてみると「普通の一バイト + もしくは -1 」というのを表すには 8 ビットでは足りない
- そこで int にしましょうということになった

# 以下はかわいそうなことになります

- char c; /*  もしくは unsigned char */
  while ((c = getchar()) != EOF) { … }