セキュリティキャンプ２０１４
セキュアなシステムを作ろうクラス
ＯＳの見える化を考えるゼミ
テーマ#01「Linuxシステムの運用監視プログラムを作ろう」
システムプログラミングのススメ

熊猫さくら

2014/8/12-2014/8/16

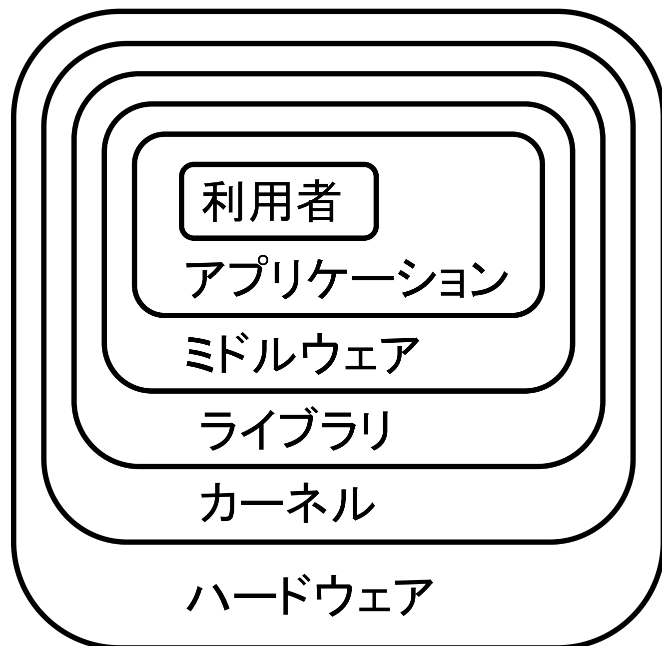
この資料を作った理由

- ＯＳＳを採用した業務システムのトラブル対応業務に従事してきて、ＯＳ（カーネル）の動作が関係する問題を切り分けるにあたり、事象を再現したり情報を取得するためのプログラムを作成できる人（あるいは、作成できるスキルを持っていて実際に作成しようとする人）が私以外に居ないことが問題だと思ったから。
- コンピュータシステムの仕組みを知らずに利用していることで、非効率的な使い方をしていたり無茶な要求をしていたりする人が多いことが問題だと思ったから。

コンピュータシステムについて

- プログラムの役割としては、多くの階層が存在している。

（プログラムの役割を基準にした分類）



利用者・・・コンピュータシステムを利用する人

アプリケーション・・・利用者に直接見えている
プログラム

ミドルウェア・・・アプリケーションから使われる
プログラム

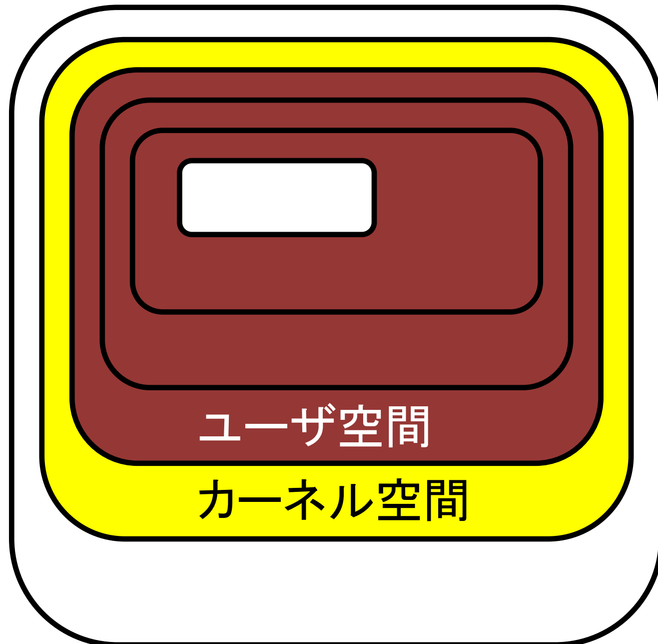
ライブラリ・・・アプリケーション／ミドルウェア
から使われるプログラム

カーネル・・・アプリケーション／ミドルウェア／
ライブラリから使われるプログラム

ハードウェア・・・プログラムを動作させる環境
（CPUやメモリ）および各種装置

でも、プログラムの動作する環境としては２種類しかない。

（プログラムの動作する環境を基準にした分類）



ユーザ空間・・・アプリケーション／ミドルウェア／ライブラリが動作している環境

カーネル空間・・・カーネルが動作している環境

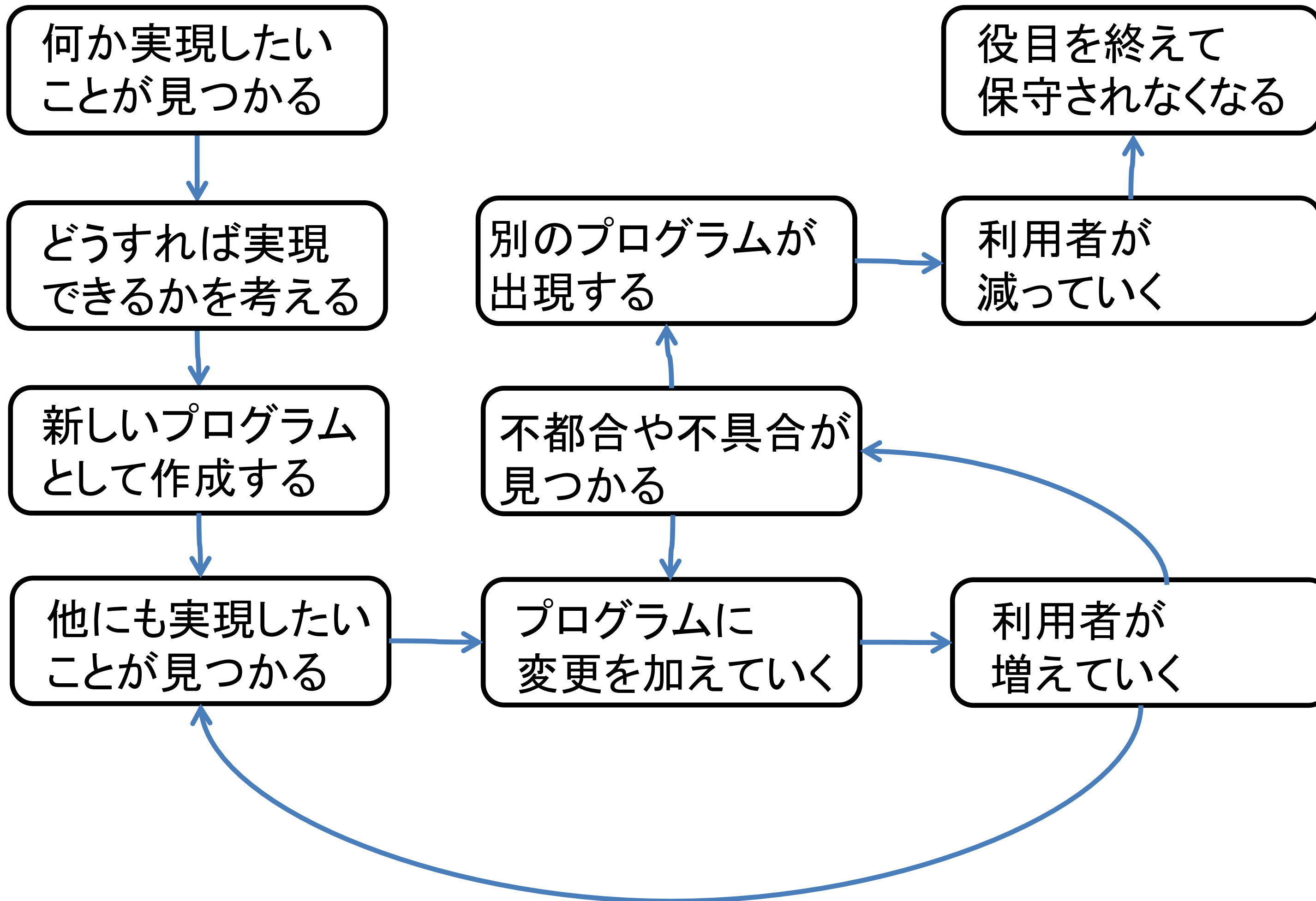
- ユーザ空間とカーネル空間とで役割の違いはあっても、プログラミングを行う上での基本は同じ。それは、プログラムというものが、何らかの入力を受け取り、何らかの処理を行い、何らかの結果を出力するための道具だから。
  - プログラムを作成するというのは、法律を定めるようなもの。
  - プログラムを利用するというのは、定められた法律に従って生活するようなもの。
- プログラムとは何かを理解すると、自然とプログラムの処理内容やセキュリティにも関心が向く筈。
  - マルウェア・・・利用者に有害な法律
  - セキュリティホール・・・悪用や乱用が可能な法律

- 「システムプログラミング」を経験してみませんか？
  - カーネルの機能を直接呼び出すこと（システムコール）やライブラリ関数を意識したプログラムを書くことで、

普段は意識していない／知らない
コンピュータシステムの内幕を
知ることができる。
内幕を知っていると、効率的な
活用ができるし、トラブル発生
時にも適切に対処できる。

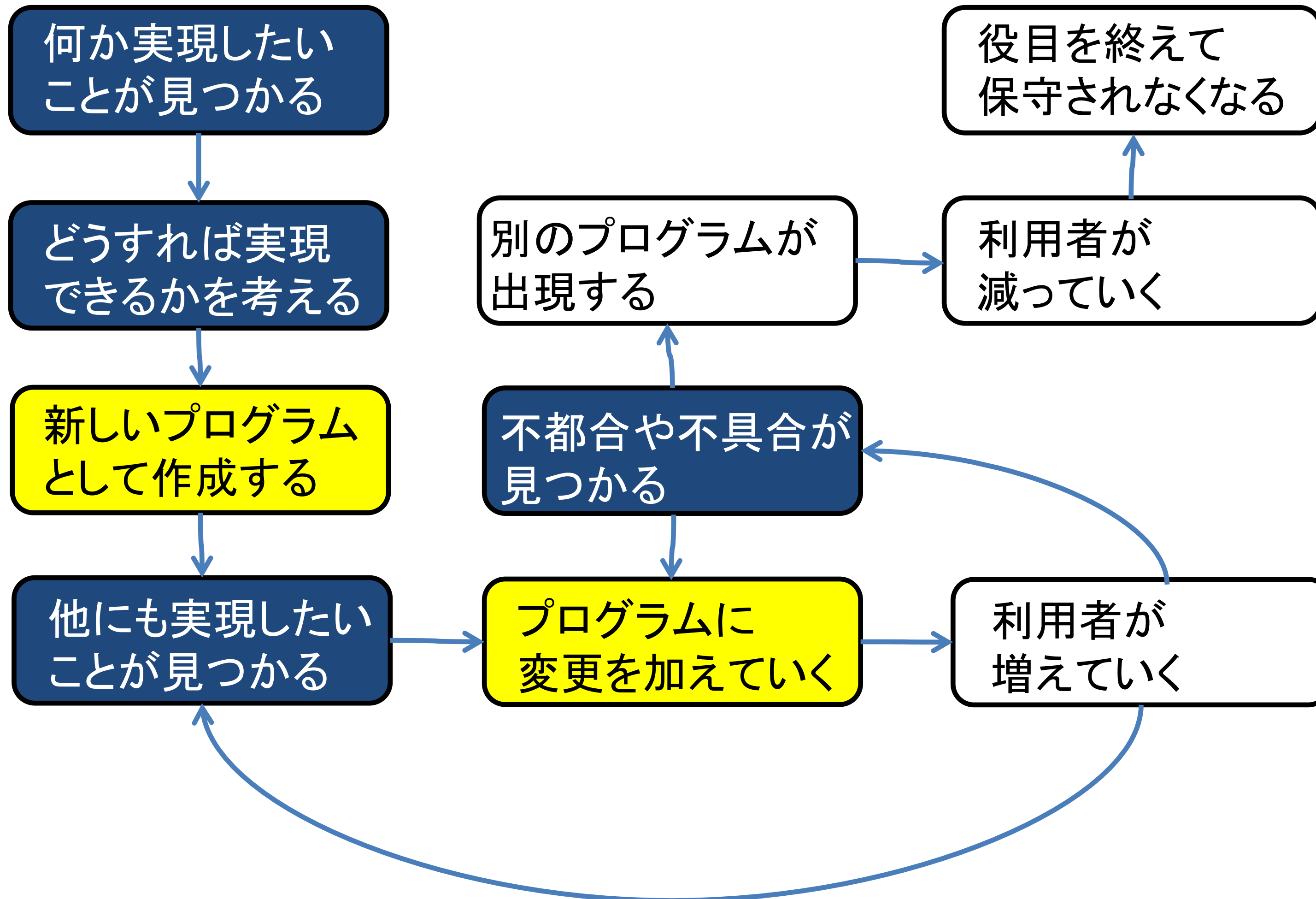
プログラムのライフサイクル

- 同じプログラムが永久に存在しつづけることは無い。



プログラムに関わる人たち

- 「開発する人」と「利用する人」とが異なることがある。



- プログラムを開発する側の視点
  - サーバ／デーモン型
  - システムツール型
  - ユーティリティ型
  - （その他にもあるだろうけれど省略）
- プログラムを利用する側の視点
  - サポート対象（問題発生時に面倒を見てくれたり不具合を修正したりしてくれる）ソフトウェアか否か
    - プログラムについて詳しくない人は、プログラムの内容（規模や複雑さなど）を考慮しないまま、サポート対象か否かだけで利用の是非を判断してしまう。
- プログラムを保守（サポート）する側の視点
  - 保守を行いやすいプログラムや構成になっているか否か

ＯＳＳを採用した業務システムの現実

- 基本姿勢
  - ＯＳＳ・・・問題が見つかったところをアップデートしながら（修正を加えながら）利用していく。
  - 業務システム・・・最初から完璧に仕上げたものをアップデートせずに（修正を加えずに）利用していく。
- 業務システムにＯＳＳを採用するということは、ＯＳＳの基本姿勢を受け入れるということ。
  - みんなが使う部分はみんなで協力して作ろうという理想
    - 成果を共有すれば、無駄な開発を避けられるよね？
  - リグレッションやセキュリティバグに振り回されるという現実
    - 自分の関心のある領域以外での問題が絶えない。

- ソフトウェアのアップデートが難しい業務システムでは、導入時点で不完全なものを使うのはリスクになる。そして、内容を把握せず、十分な試験を行わずに導入してしまった結果、長期安定試験や負荷試験では発見できない不具合に悩まされている。
  - 想定以上の負荷をかけたり、何ヶ月も連続運転して初めて発現するような不具合がある。ソフトウェアの複雑化や、マシンパワー（搭載ＣＰＵ数やメモリ量）の増加により、試験期間中に不具合を見抜けないことが多い。
- ＳＩｅｒは、自前でプログラムを書くことを避け、既存のサポート対象のプログラムに依存してきた。しかし、他人が書いたサポート対象のソフトウェアを組み合わせて構築した結果、内容がブラックボックスになって自力でサポートできなくなるという不幸が起きている。

- 仕様書という言語でしか理解できない人たちを大量に生みだしてしまい、プログラムの内部動作を理解しないで使っているため、トラブルに弱い。そして、トラブルに対応するためのスキルを持つ人の育成を怠ったため、サポートはいつも人手不足。

- ＯＳＳを開発する側、利用する側、保守する側が異なる方向を向いている。
  - 昔ながらの「設計」「構築」「運用」「保守」という役割分担は、日本人が得意としていたすりあわせによる連携プレーが通用しなくなったことで、役割分担の壁を越えられず、フィードバックができない分断状態にある。
    - これはとってもマズイ状況なのでは？

自前でプログラムを書かないのは何故？

- 書き方を知らないから？それとも、責任やサポートを他人に押し付けたいから？
  - でも、他人が書いたプログラムは落とし穴だらけ。自分が期待しているほど安定して動作することは殆どない。特に、セキュリティが重要視されるようになった現在では、品質の悪い部分を攻撃され、そのプログラムを捨てざるを得ないまでになることも。
  - 試験時には発生しなかった問題が運用時に発生して慌てている。試験時に問題が発生しなかったらＯＫというものではない。
  - ソフトウェアのアップデートが頻繁に出るが、再試験する余力が無いので適用できない。その結果、設定変更などの回避策で済ませようとする。
    - これを繰り返し続けてきたのが現在のＳＩｅｒでは？

問題が発生すると原因究明と再発防止策が重要視される傾向がある。

- でも、問題発生時に何処をチェックしたら良いのかが解らない？
  - 問題の原因や理由を知りたければブラックボックスでは駄目。
    - システムの動作を見えるようにするツールがあります。
      - http://I-love.SAKURA.ne.jp/tomoyo/LCJ2014-ja.pdf
    - パフォーマンスなどを測定するツールがあります。
      - http://www.slideshare.net/brendangregg/linux-performance-analysis-and-tools
  - システムの中身を知っていれば、ソフトウェアアップデートの影響範囲を判断できるので、条件反射的に拒否するような対応は無くせる筈では？
    - 連携プレーを復活させられないだろうか？

どうすれば良いの？

- 責任範囲という縛りから抜け出そう。
  - 「自分に関係しないことは気にしない」では済まされない。
    - システムのトラブル／セキュリティの脅威は、助け合いなしでは対処できない。
  - セキュリティで悩む前にできることがあるよね？
    - 思いやりのあるプログラミング
      - プログラムは自分だけのものではない／他人からどう見えるかを考える想像力
        - 誰でも理解できて保守できるプログラムになっていますか？

悩んでないで、手を動かしてみよう。

- Ｗｅｂの世界では公開ＡＰＩを使って自分好みのクライアントプログラムを自作する動きが出てきている。ＯＳの世界だって、システムコールという公開ＡＰＩがあるのだから、自分好みのシステムプログラムを自作したっていいんじゃない？
- 特定の目的に特化したシステムツールやユーティリティなら、Ｃ言語でも数十行から数百行程度の規模で目的を達成できるのでは？
  - 必要としていない機能による不具合やリソース消費に振り回されるよりも気楽だと思うよ。
  - システムコールを使ったプログラミングができるようになると、問題事象の再現ができるようになる。それは、ひいては問題事象を未然に防ぐことにも繋がる。

Unix 的な考え方

- 単機能なプログラムを作って組み合わせて使おう。
  - あるコマンドの標準出力を別のコマンドの標準入力へとパイプで接続する。コマンドの exit コードが 0 （正常終了）か 0 以外かで処理を分岐する。

```
---------- 死活監視スクリプト　ここから ----------
#!/bin/sh
if curl -I -s http://www.example.com/ | head -n 1 | grep -qF 200
then
  echo "Server is alive."
  exit 0
else
  echo "Server is dead."
  exit 1
fi
---------- 死活監視スクリプト　ここまで ----------
```

この死活監視スクリプトにはどんな罠が存在している？

- コマンドの exit コードが 0 ではないことが、サーバがダウンしていることを意味しているとは限らない。
  - メモリ不足やリソース使用量の上限制限に引っ掛かったことが原因で、 curl / head / grep を実行するための子プロセスの作成に失敗するかもしれない。
  - オープン可能なファイル数の上限制限に引っ掛かったり、ウィルス対策ソフトのリアルタイムスキャンがタイムアウトエラーを起こしたことが原因で、 curl / head / grep が必要とする共有ライブラリファイルなどのオープンに失敗するかもしれない。
  - クライアント側や途中のネットワークがダウンしているなど、サーバ側ではない問題によって curl コマンドがエラーとなるかもしれない。
  - 他にもいろいろな可能性が考えられる。

- exit コードで伝達できる情報は、何らかの問題事象が発生したかどうか程度。
  - しかも、利用する側が問題だと考える条件と、プログラムが 0 以外の exit コードを設定する条件とが合致しているとは限らない。
  - exit コード頼みのシェルスクリプトではエラーハンドリングが困難。信頼性が求められるシステムでは、エラーハンドリングをきちんと行うことが必要。

シェルスクリプトに頼っていて大丈夫？

- スクリプトで対応できたのは、良識的で常識的な使い方をしていた昔の話。今日では様々な罠が待ち受けている。
  - マシンパワーの増加／業務負荷の増加／セキュリティ問題
    - 昔のプログラムが想定していた常識的な使われ方から逸脱してきている。
    - 悪用や競合状態や妨害を想定していない古いプログラムが使われ続けている。
    - プログラムの内部動作を理解していないと、セキュリティを考えることもできない。
  - 単機能なプログラムを組み合わせて利用するという方法は、人間がシェルから手動で実行する場合には便利だが、死活監視のように定期的に自動実行する用途には不向き（オーバーヘッドが大きく、エラーハンドリングが貧弱）かもしれない。

自作プログラムに置き換えるとどうなる？

- 死活監視スクリプトの処理内容をＣ言語で書き直した例を示します。

---------- Ｃ言語による死活監視プログラム例　ここから ----------

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

static in_addr_t get_addr(const char *hostname)
{
        struct hostent *hp = gethostbyname(hostname);
        return *(in_addr_t *) hp->h_addr_list[0];
}
```

```
int main(int argc, char *argv[]) {
    char buffer[8192];
    char const query[] = "HEAD / HTTP/1.0¥r¥nHost: www.example.com¥r¥n¥r¥n";
    struct sockaddr_in addr;
    const int fd = socket(PF_INET, SOCK_STREAM, 0);
    unsigned int status;
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = get_addr("www.example.com");
    addr.sin_port = htons(80);
    connect(fd, (struct sockaddr *) &addr, sizeof(addr));
    send(fd, query, sizeof(query) - 1, 0);
    shutdown(fd, SHUT_WR);
    recv(fd, buffer, sizeof(buffer) - 1, MSG_WAITALL);
    close(fd);
    sscanf(buffer, "HTTP/%*u.%*u %u", &status);
```

```
    if (status == 200) {
            printf("Server is alive¥n");
            return 0;
    } else {
            printf("Server is dead¥n");
            return 1;
    }
}
```

---------- C言語による死活監視プログラム例　ここまで ----------

思いやりのあるプログラムにするには何が足りない？

- 前述したプログラム例では、（勉強目的のために意図的に）エラーハンドリングやセキュリティホール対策を省略しています。どのようなエラーや問題が発生するかを考え、適切に書き直してみましょう。
- また、システムコールでエラーが発生した場合、 errno という変数がセットされるので、プログラムは errno という粒度でどのようなエラーが発生したのかを利用者に知らせることができます。 errno がネットワーク関連のエラーを示している場合、 ping や traceroute を実行して通信経路上の状態を確認するというのも親切かもしれませんね。

思いやりのあるシステムにするには何が足りない？

- しかし、なぜ当該 errno がセットされたのかを知るためには、システムコールの先にある、ＯＳ（カーネル）内の動作まで突き詰めていく必要があります。具体的には、カーネルが何らかのメッセージを出力していないかを調べたり、システムコールにより行われた処理のどこで問題が発生しているのかを SystemTap を用いて追跡したりすることになります。
- そして、ソフトウェア的な問題であるとは限らない（例えば、ネットワークインタフェースが故障したとか、通信経路上のどこかで輻輳が発生していたとかいうケースもある）ため、どんなに頑張っても全ての原因を利用者に知らせることはできません。

- カーネル空間とユーザ空間という動作環境の区別があり、アプリケーション／ミドルウェア／ライブラリ／カーネルという役割分担が存在する世界で、アプリケーションのことしか気にしていない利用者に、どこでどのようなトラブルが発生しているのかを解りやすく伝えることは困難です。
  - 丁寧なアプリケーションを作るだけではカバーできない範囲について、どのようなことができるかを考えてみましょう。
    - 教育などによって、内幕を知ってもらう？
    - 役割分担を考え直す？
  - セキュリティ上の問題を含めてトラブルに振り回されている状況を改善するためにも、「何もしないで諦める」というのはご勘弁を・・・。

おまけ：システムプログラミングに必要なのは？

- 開発環境・・・仮想化環境で構いません。
- マニュアルの調べ方・・・ man ページを見れば解ります。
- ソースコードの見つけ方・・・インターネット上には、いろいろなサンプルプログラムが存在しています。システムコールやライブラリ関数をどのように使えば良いのかの参考になります。
- 問題意識・・・プログラムは問題を解決するための道具です。どんな問題を解決したいのかを意識してください。
- 根気・・・人目を惹きやすいアプリケーションプログラムとは違い、縁の下の力持ちであるシステムプログラムを手伝ってくれるという人は少ないので、「自分がやらなきゃ誰がやる」という覚悟が必要です。でも、成し遂げることができたら、「物事を最初から最後まで通して考える能力」というご褒美が貰えます。

おまけ：熊猫さくらが伝えたいこと

- 表面だけの関係で済むならば、それは楽なのかもしれません。
  - その代わり、自己や自組織の利益や都合を優先したことの代償は高くつきますよ。
- 現実の問題を知らないまま／知らせないまま進めることは、解決したい問題よりも大きな問題を生じさせかねません。
  - 誰かの利益は別の誰かの不利益になっているかもしれません。実社会で起きていることをよく考えてみましょう。
- 都合の悪いことを隠しながら物事を進めていけるのがプロではなく、都合の悪いことを隠さずに相談して解決していけるのがプロだと思います。
  - ジリ貧状態であっても、それに気付いていないか、気付いていても何もできないという人が圧倒的です。
  - 勇気を持って情報発信／行動してください。