

# Linuxリソース管理の改善 ～「coredump masking」と 「ネットワークメモリ管理の改善」～

Hitachi, Ltd., Systems Development Laboratory  
Linux Technology Center

\*大島 訓 <satoshi.oshima.fk@hitachi.com>  
河合 英宏 <hidehiro.kawai.ez@hitachi.com>

1. coredump masking
2. ネットワークメモリ管理の改善



## 1. coredump masking

- 河合 英宏が開発

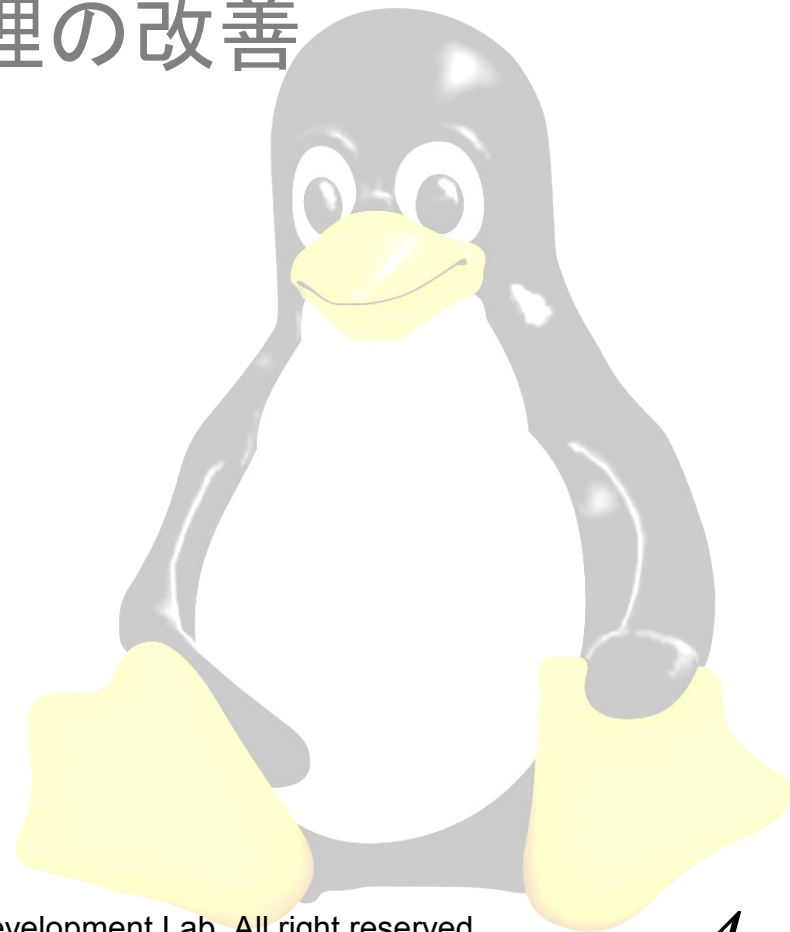
## 2. ネットワークメモリ管理の改善

- 青木 英郎、安井 隆弘、大島 訓が開発

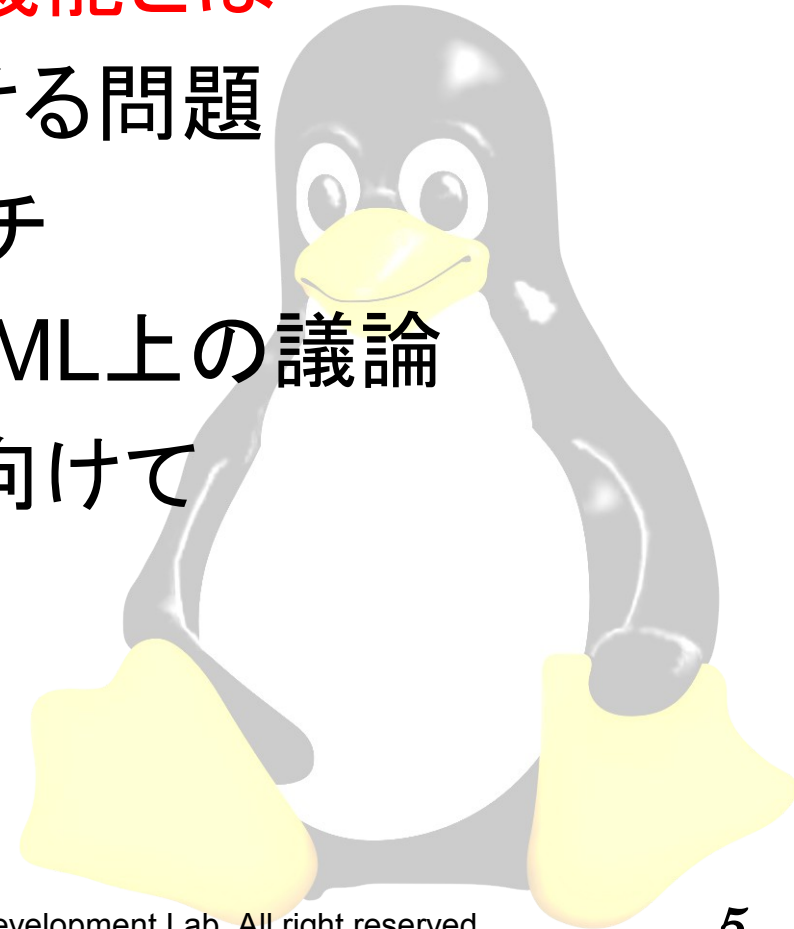


1. coredump masking

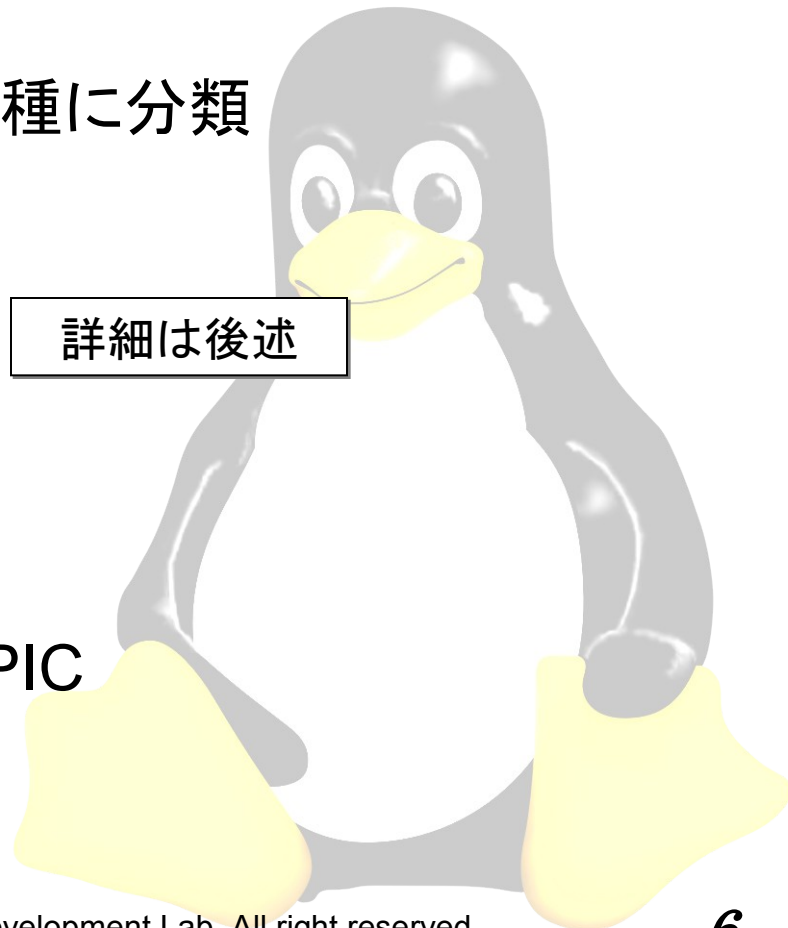
2. ネットワークメモリ管理の改善



- **coredump masking機能とは**
- コアダンプ処理における問題
- 問題解決のアプローチ
- 設計と実装およびLKML上の議論
- RHEL 5への移植に向けて
- 使い方
- まとめ



- どんな機能？
  - コアダンプ時に出力するメモリ領域の種類をプロセスごとに指定できるようにする
- メモリ領域の種類？
  - プロセスのメモリ領域を次の4種に分類
    - anonymous かつ private
    - anonymous かつ shared
    - file-backed かつ private
    - file-backed かつ shared
- サポート状況は？
  - カーネル: 2.6.23以降
  - バイナリ形式: ELF、ELF-FDPIC



- coredump masking機能とは
- コアダンプ処理における問題
- 問題解決のアプローチ
- 設計と実装およびLKML上の議論
- RHEL 5への移植に向けて
- 使い方
- まとめ

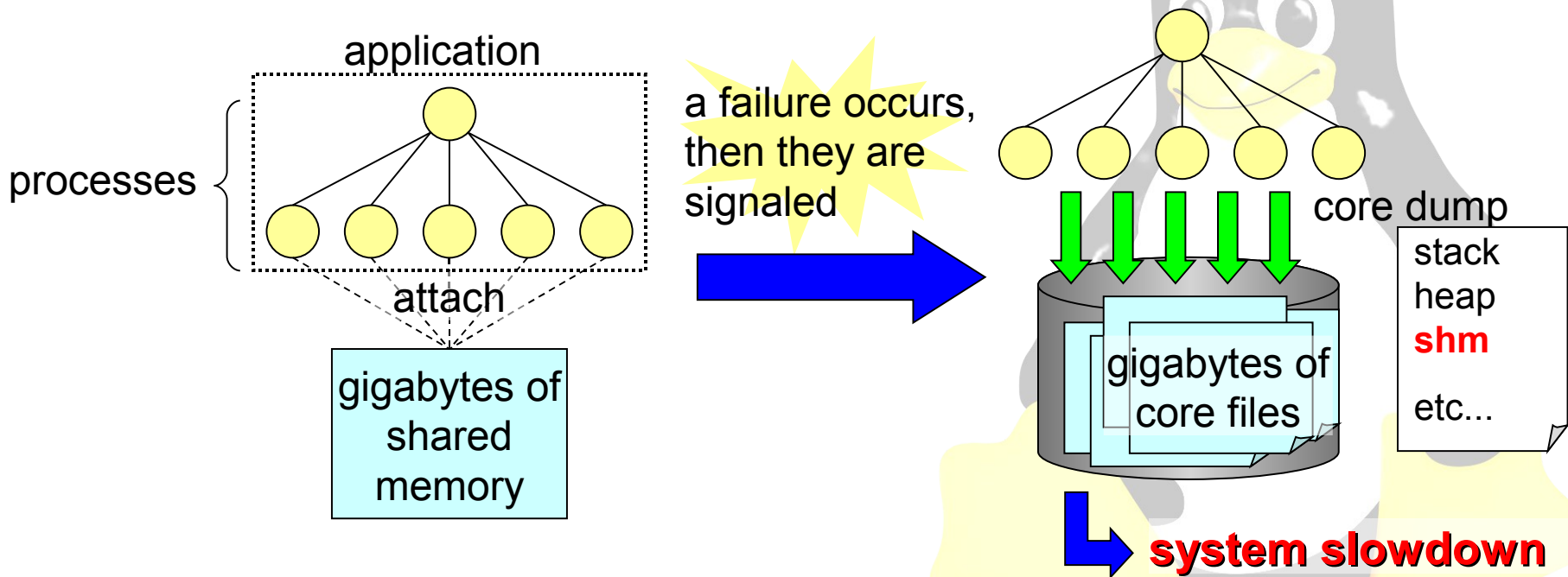


# A Problem on Core Dump

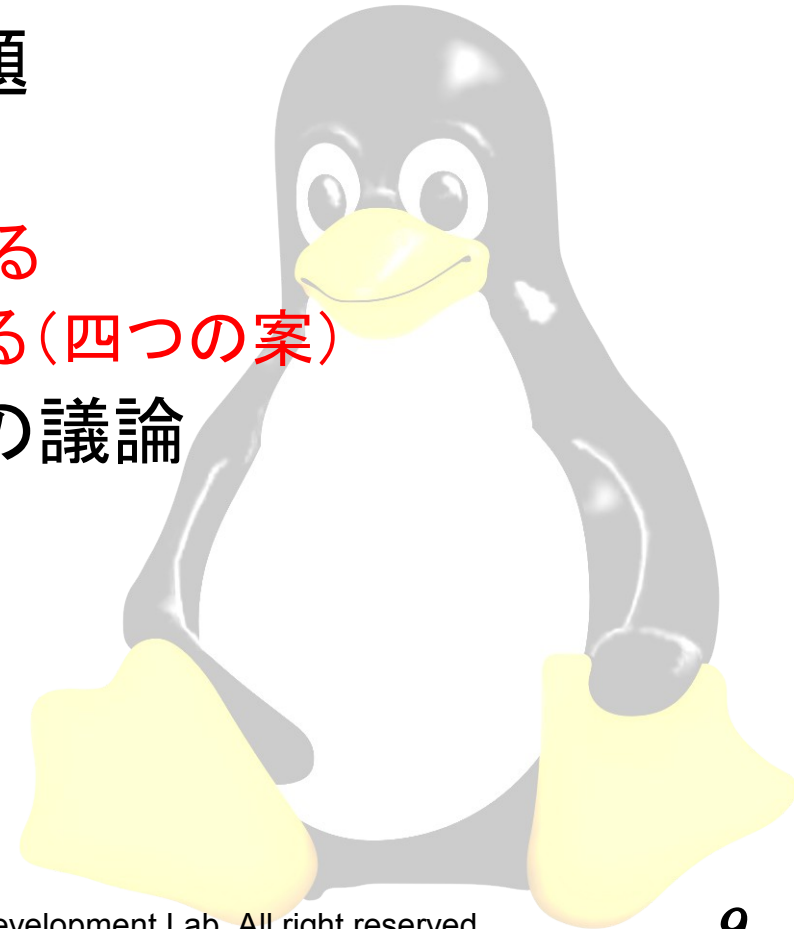
DBが使用する共有メモリ量が増大してきた

障害時に共有メモリ領域を含む巨大なコアファイル群が生成される

大量のI/Oにより長時間システム全体がスローダウンする



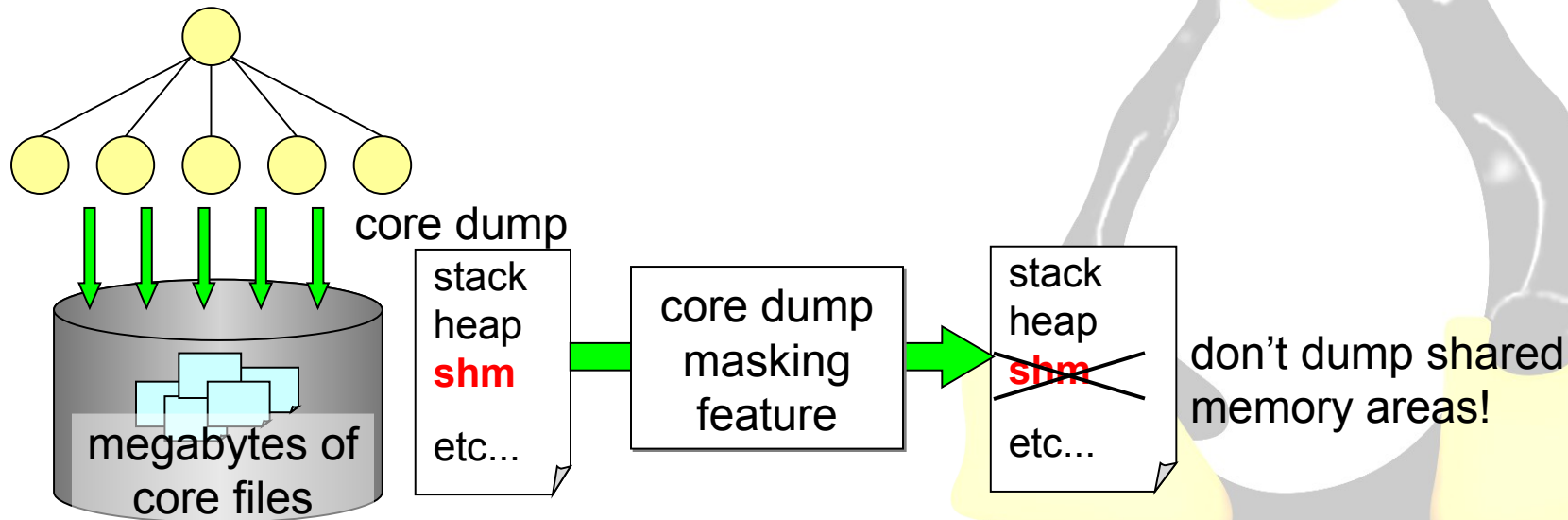
- coredump masking機能とは
- コアダンプ処理における問題
- **問題解決のアプローチ**
  - コアファイルサイズを小さくする
  - 共有メモリのダンプを抑止する(四つの案)
- 設計と実装およびLKML上の議論
- RHEL 5への移植に向けて
- 使い方
- まとめ



# How Should We Make Core Files Smaller?

コアファイルサイズが大きくならないようにしたい！

- ✗ 方針1: `ulimit -c` でコアファイルサイズを制限する
  - 重要な情報が抜け落ちてしまう(特にメモリ空間の後方にあるスタック)
- 方針2: 共有メモリをダンプしないようにする
  - 同じ内容の共有メモリを何度もダンプするのは無駄
  - 障害解析に必要な情報が含まれている可能性は低い



- 障害時は確実にコアダンプが採れる必要がある
- 効率的にコアダンプできる
  - システムスローダウンを起こしたくない
  - できるだけ早くサービスを復旧したい
- 特定のプロセスに対して、共有メモリ領域をコアダンプしないようユーザ空間から**設定**できる
- 子プロセスをfork(2)するとき、**設定**を親プロセスから継承する
- execve(2)後も**設定**を維持する
- 任意のプロセス、任意の時点で**設定**を変更できる
  - ただしパーミッションチェックは行う

# How Should We Omit Shared Memory Areas? (1)

- google-coredumper (Markus Gutschke案)
  - ユーザ空間でコアダンプするためのライブラリ
  - シグナルハンドラの延長で動作する
  - 同ライブラリを改造し共有メモリ領域のダンプを抑止可能にする
- 長所
  - カーネルを修正して対策するのに比べ敷居が低い
- 短所
  - 障害が起きたプロセス上で動作するため、ダンプに失敗する危険がある
    - google-coredumperの動作に必要なデータが壊れていた場合
    - シグナルハンドラの延長でフォルトが起きた場合

**障害が起きたときは確実にダンプを採りたい！**

- ptrace(ライクな機能)を使ったコアダンプ (David Howells案)
  - コアダンプ時にカーネルから別プロセスを起動し、そのプロセスが ptrace を使って対象プロセスのコアをダンプする機能をカーネルに追加する
  - コアダンプを行うプロセスは、設定によっては共有メモリ領域をダンプしないようにする
- 長所
  - 障害を起こしたプロセスとは別のプロセスがダンプを採るので確実性が高い
- 短所
  - カーネルの修正が必要
  - ptraceはワード単位でアタッチ先プロセスのメモリを読むため低速

システムスローダウンを防ぐという要求を満たさない

# How Should We Omit Shared Memory Areas? (3)

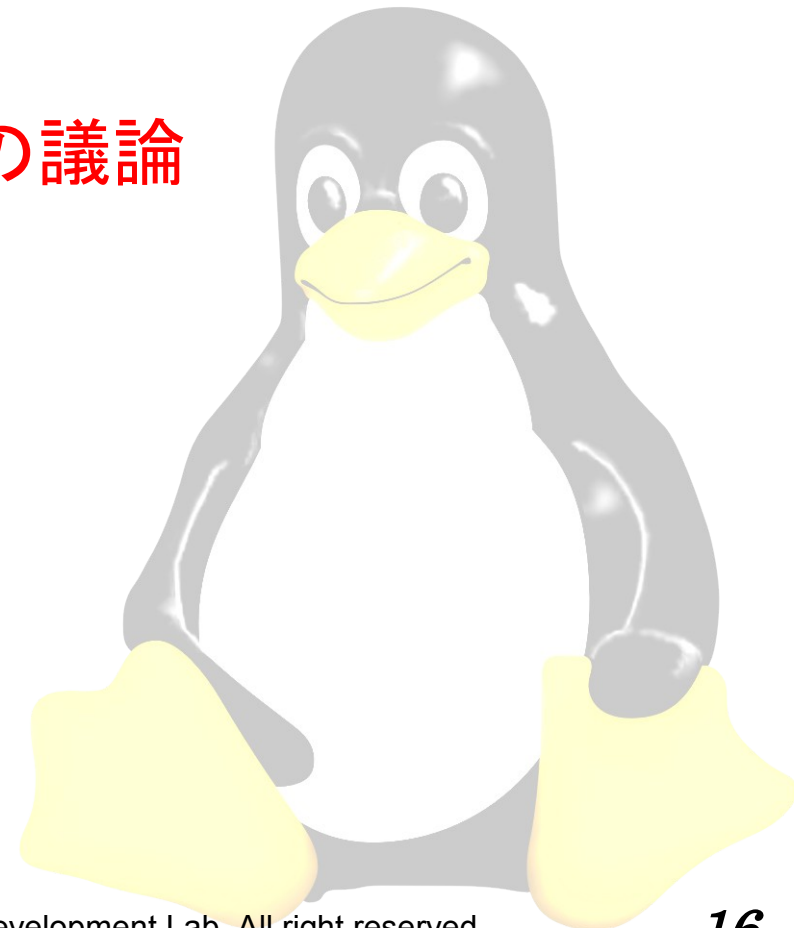
- core dump over pipe (Andrew Morton案)
  - コアダンプの内容をファイルではなくパイプを通して任意のコマンドに送る機能を使う
  - コアダンプの内容を受け取ったコマンドは共有メモリ領域の内容を破棄した後、コアファイルを生成する
- 長所
  - 障害を起こしたプロセスとは別のプロセスがダンプを採るので確実性が高い
  - カーネルの修正は不要
- 短所
  - ダンプしないつもりの巨大な共有メモリ領域の内容も送ってしまう(2GBの共有メモリを転送するのに7秒要した)
  - パイプを通した通信はメモリコピーやプロセス切り替えが多く非効率

システムスローダウンを防ぐという要求を満たさない

- カーネルのmaydump( )を改造 (我々の案)
  - メモリ領域(VMA)単位でダンプするしないを決定する関数
  - テキスト領域や、MAP\_SHAREDフラグ付きでファイルをmmapした領域など、何処かのファイルに残るメモリ領域をダンプしないようにするのに用いられる
  - 共有メモリのダンプの可否を設定で制御できるように改造する
- 長所
  - 障害を起こしたプロセスのメモリ空間ではなくカーネル空間でダンプするので確実性が高い
  - ダンプ処理のオーバヘッドが小さい
- 短所
  - カーネルの修正が必要

要求を満たすコアダンプ方法

- coredump masking機能とは
- コアダンプ処理における問題
- 問題解決のアプローチ
- **設計と実装およびLKML上の議論**
  - 初期バージョン
  - 議論の経過
  - 最終バージョン
- RHEL 5への移植に向けて
- 使い方
- まとめ

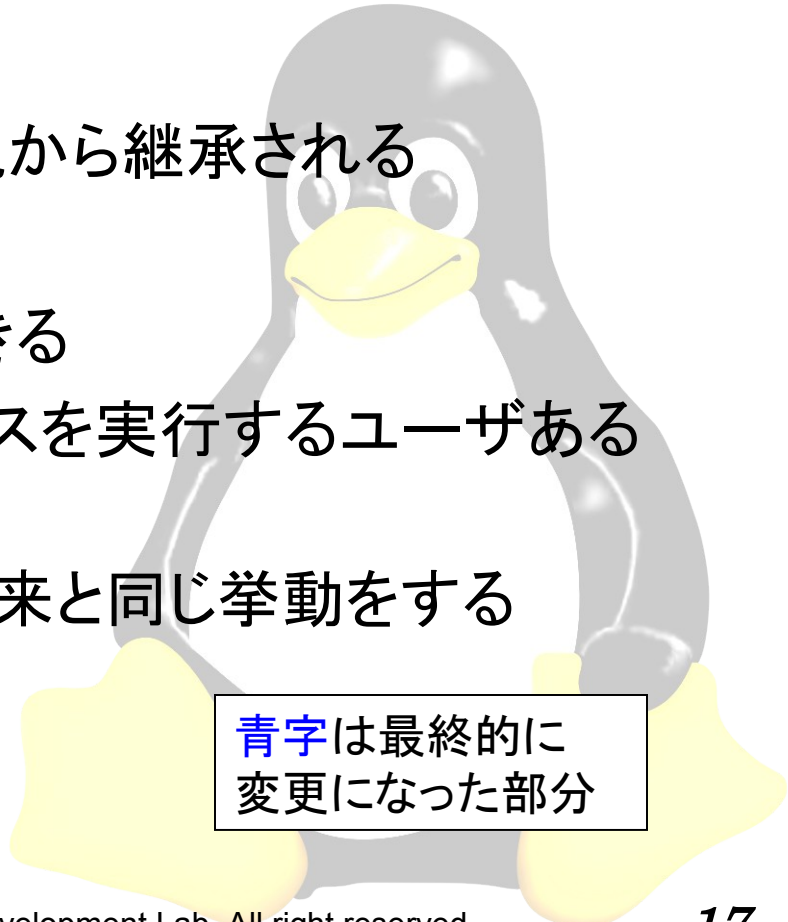


- 仕様

- 設定によりanonymousかつsharedマッピングなメモリ領域をコアダンプしないようにできる
- プロセスごとに設定できる
- 設定は子プロセス生成時に親から継承される
- 設定はexec後も維持される
- 設定は任意の時点で変更できる
- 設定を変更できるのはプロセスを実行するユーザあるいはrootユーザである
- 何も設定を変更しなければ従来と同じ挙動をする

- 設定インタフェース

- /proc/<PID>/**coremask**



青字は最終的に  
変更になった部分

- 設定値の表現
  - ビット0が1ならanonymousかつsharedマッピングのメモリ領域をダンプしない
- 設定値の格納
  - `mm_struct`に`core_mask`メンバを追加し、そこに格納する



mm\_structの`dumpable:2' の隣に  
coredump\_omit\_anon\_memory:1を追加すればmm\_structの  
サイズを増やさずに済む – Andrew Morton

- Andrew提案の方法は新たな問題を生む
  - dumpableへの書き込みと  
coredump\_omit\_anon\_memoryへの書き込みが同  
時に起こると、一方の書き込み内容が失われる  
⇒ 排他制御が必要

新規定義したグローバルスピンロックを使って排他制御するよう  
にした

ロックは難しい。新しいロックを定義すべきではない。  
- Pavel Machek

- Andrew MortonとF2Fミーティングを行い、ロックを取る代わりにアトミックビット操作を使うという提案を受けた
  - ただしdumpableは0~2の整数値をとるため、アクセスには2回のビット操作が必要 ⇒ dumpableの同時読み書きが競合する

dumpableの変更		メモリ中の状態(2進)		
変更前	変更後	変更前	変更中	変更後
0	1	00	01	01
0	2	00	10*	11
1	0	01	00	00
1	2	01	11	11
2	0	11	10*	00
2	1	11	11	01

\*) 読み取り時は10を11と見なす

dumpableの0~2の値を二つのフラグで表現する。値の変更中に読み取りを行った場合、変更前か変更後のどちらかを返すようにする。

ulimit -cmask=<bitmask> というインタフェースが良いのでは？  
- Pavel Machek

- 一見良さそう
  - 設定はfork(2)時に継承され、execve(2)後も維持される
  - コアファイルのサイズ上限を指定する機能が既にある
- しかし…
  - soft/hard limitの仕組みに適合しない
  - コマンドの実行後は容易に設定を変更できない
    - コマンド内部からsetrlimit(2)を呼び出す仕組みが必要

Pavellは最後までulimit方式を推したが、最終的にはAndrew Mortonからprocインタフェース方式で合意が得られた

コアダンプ中に設定を変更しても大丈夫？

– Pavel Machek

- ダンプの可否判定 (`maydump()`) は各メモリ領域につき2回実施されるが、双方の判定が異なると壊れたコアファイルが生成されてしまう  
⇒ コアダンプ中に設定を変更できないようにする必要あり
- 対策
  - 案1) コアダンプ開始時に設定値をローカル変数にコピーし、以降その値を使う (河合)
  - 案2) グローバルに定義したread/writeセマフォでコアダンプ中の設定変更をブロックする (David Howells)

最終的に、Andrew Mortonと意見の一致した案1を採用した

- 仕様

- メモリ領域を4種に分類し、その種類ごとにコアダンプする／しないを設定できる
- プロセスごとに設定できる
- 設定は子プロセス生成時に親から継承される
- 設定はexec後も維持される
- 設定は任意の時点で変更できる
- 設定を変更できるのはプロセスを実行するユーザあるいはrootユーザである
- 何も設定を変更しなければ従来と同じ挙動をする

- 設定インタフェース

- /proc/<PID>/coredump\_filter

青字は最終的に変更になった部分

- 設定値の表現
  - ビット0が1ならanonymousかつsharedマッピングのメモリ領域をダンプしない
- 設定値の格納
  - mm\_structのdumpableメンバを削除し、代わりにflagsメンバを追加する
  - flagsのビット0～1を従来のdumpable用に、ビット2～5をcoredump\_filter用に使用する
  - flagsへのアクセスにはアトミックビット操作を用いる
  - dumpable部の2ビットへのアクセスはget/set\_dumpable()によって、アトミックに行っているかのように見せかける

青字は最終的に  
変更になった部分

- coredump masking機能とは
- コアダンプ処理における問題
- 問題解決のアプローチ
- 設計と実装およびLKML上の議論
- **RHEL 5への移植に向けて**
- 使い方
- まとめ



## RHEL 5へのバックポートにおける大問題 ⇒ kABIの維持

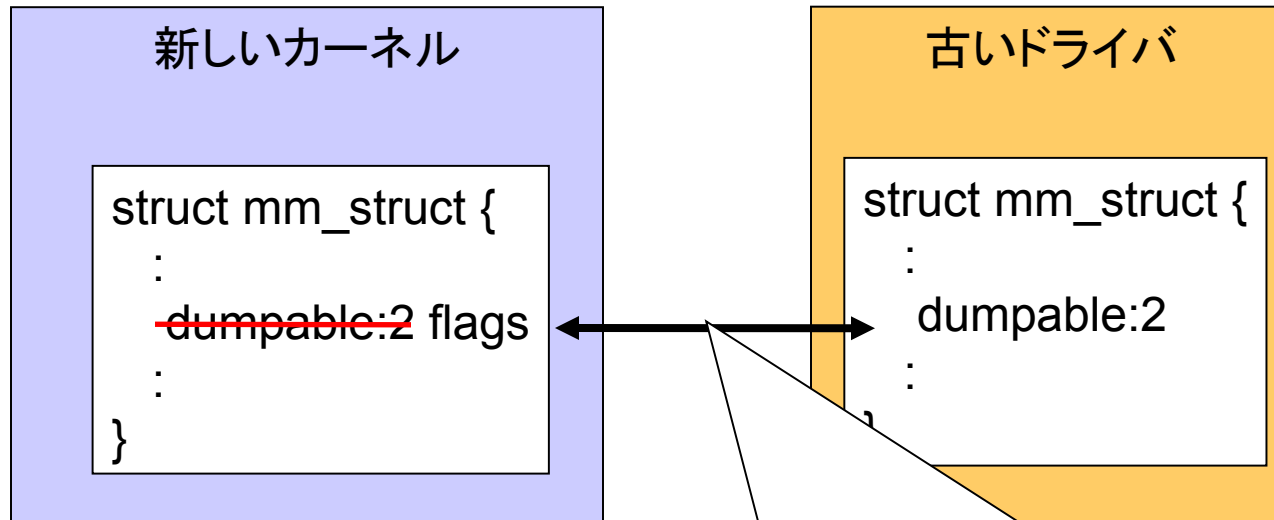
kABI(kernel Application Binary Interface)って？

カーネルがカーネルモジュール用に公開するインタフェースのこと。  
具体的にはEXPORTされた変数や関数の型定義、およびそれらの型(構造体)から辿れる構造体の定義を指す。

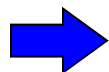
なぜkABIを維持する必要があるの？

カーネルのバージョンアップとドライバのバージョンアップを独立して行えるようにするため。

coredump-maskingパッチはmm\_structの定義を変えている

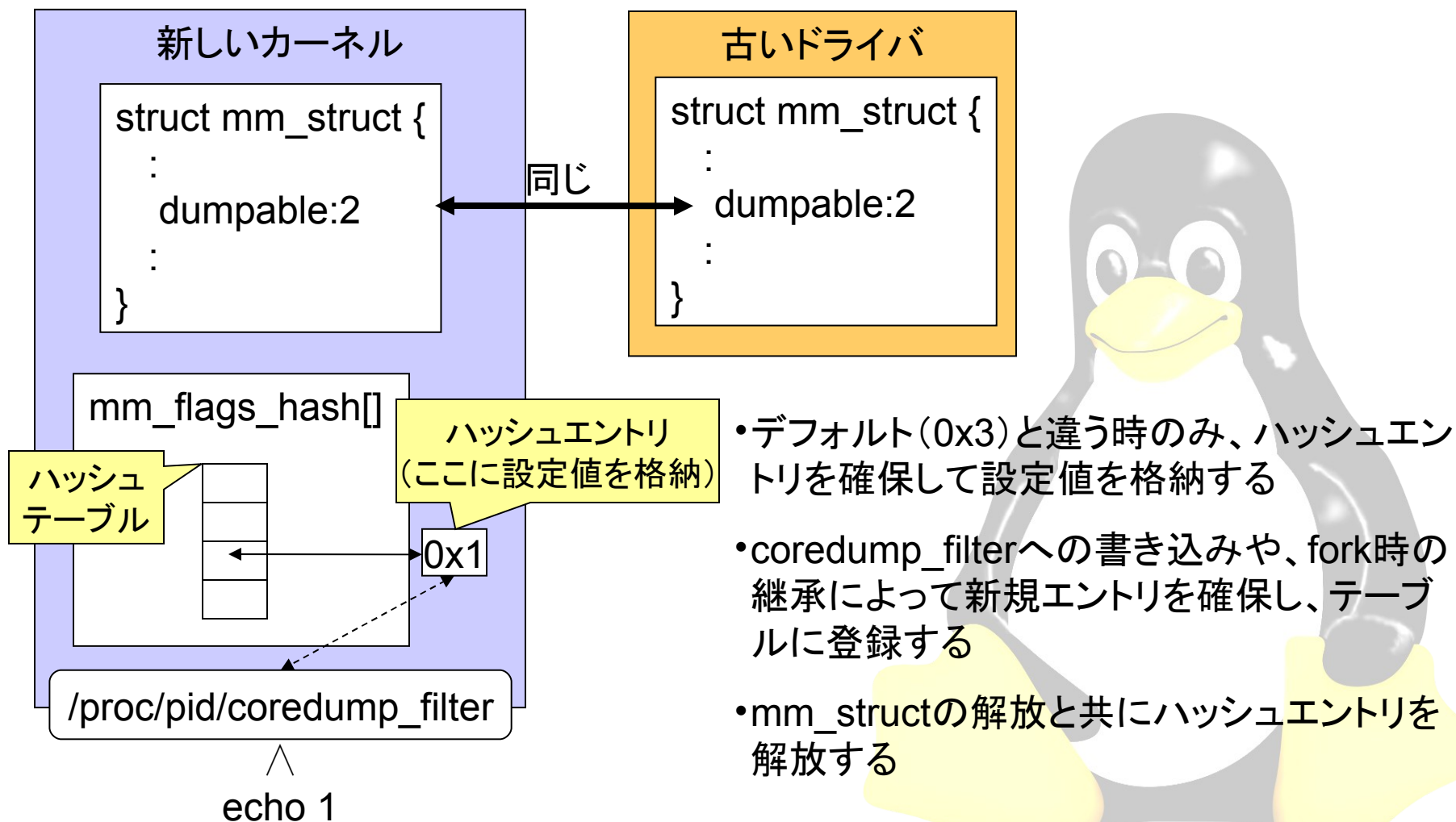


古いドライバは新しいカーネルのflagsをdumpableだ  
と思ってアクセスするため正常に動作しない  
(実際にはkABIチェックの仕組みが働き、この古いドライバを  
組み込めないようになっている)



新しいRHELカーネルで従来のドライバを動かすにはmm\_structを変えずに  
coredump-masking機能を実装する必要がある!

## ハッシュテーブルを使って各プロセスの設定値を管理する



- デフォルト (0x3) と違う時のみ、ハッシュエントリを確保して設定値を格納する
- `coredump_filter` への書き込みや、fork時の継承によって新規エントリを確保し、テーブルに登録する
- `mm_struct` の解放と共にハッシュエントリを解放する

- coredump masking機能とは
- コアダンプ処理における問題
- 問題解決のアプローチ
- 設計と実装およびLKML上の議論
- RHEL 5への移植に向けて
- **使い方**
- まとめ



- 設定インタフェース: `/proc/<PID>/coredump_filter`
  - PID: 設定対象のプロセスのPID (厳密に言うとスレッドグループID)。マルチスレッドプロセスの場合、各スレッドは一つの `coredump_filter` ファイルを共有する
  - `coredump_filter` は4ビットのビットマスクで、各ビットはメモリ領域の種類に対応する。ビットが1にセットされていれば、コアダンプの際にその種類のメモリ領域をファイルに出力する

n-th bit	memory segment type	default
0	anonymous かつ privateマッピング	1
1	anonymous かつ sharedマッピング	1
2	file-backed かつ privateマッピング	0
3	file-backed かつ sharedマッピング	0

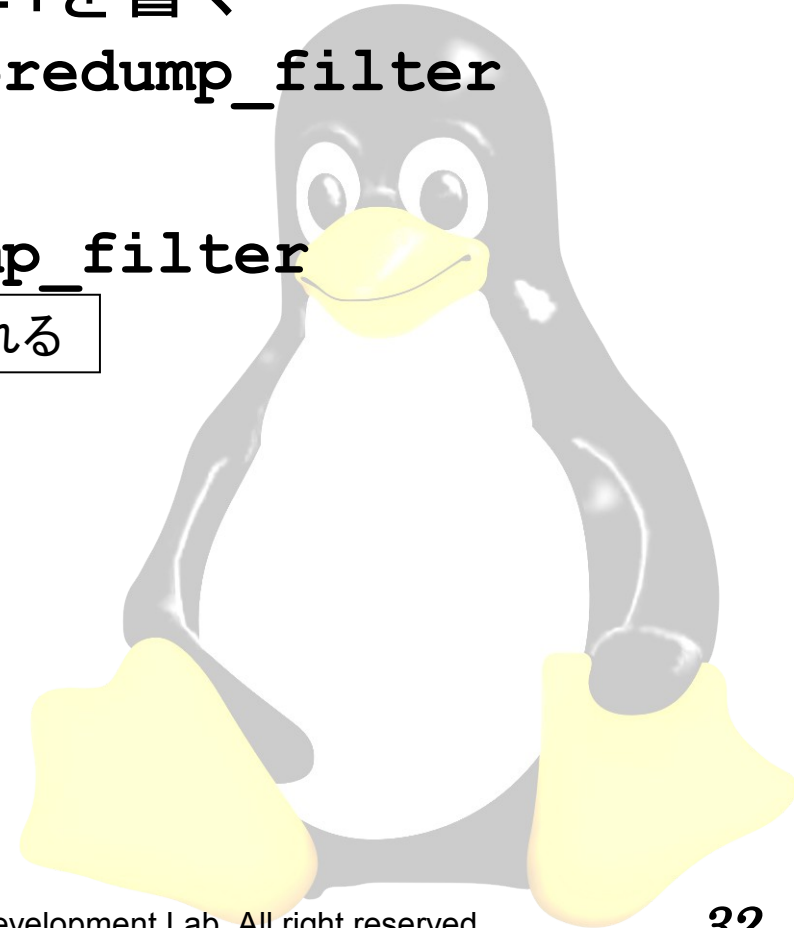
# How to Use: メモリ領域の種類

	private	shared
anonymous デフォルト: ダンプする	<ul style="list-style-type: none"><li>・スタック</li><li>・ヒープ</li><li>・BSS</li><li>・静的なデータ領域</li><li>・MAP_ANONYMOUS   MAP_PRIVATE フラグを指定して mmapした領域</li><li>・MAP_PRIVATE フラグを指定して ファイルをmmapし、かつ一度でも書き込みを行った領域</li></ul>	<ul style="list-style-type: none"><li>・IPC共有メモリ</li><li>・MAP_SHAREDでファイルをmmapした領域で、該当ファイルが削除されていた場合</li><li>・MAP_ANONYMOUS   MAP_SHARED フラグを指定してmmapした領域</li></ul>
file-backed デフォルト: ダンプしない	<ul style="list-style-type: none"><li>・テキスト領域</li><li>・MAP_PRIVATEフラグを指定して ファイルをmmapし、かつ一度も書き込みを行っていない領域</li></ul>	<ul style="list-style-type: none"><li>・MAP_SHAREDフラグを指定してファイルをmmapした領域</li></ul>

元はfile-backedなメモリ領域がanonymousになるケースなので要注意。

現在のシェルから実行するコマンドに対して、共有メモリ領域をダンプしないようにする

1. `/proc/self/coredump_filter` に1を書く  
`$ echo 1 > /proc/self/coredump_filter`
2. 設定できたことを確認する  
`$ cat /proc/self/coredump_filter`  
00000001 ← 16進8桁で表示される
3. コマンドを実行する  
`$ ./some_command`

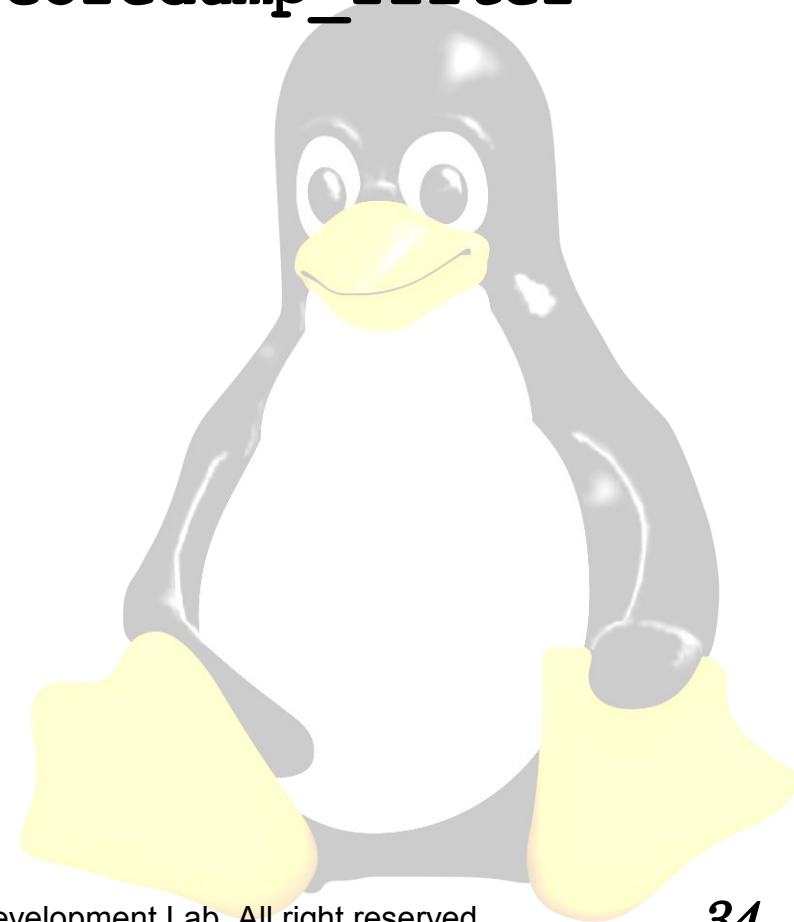


複数ある子プロセスのうち、一つだけ共有メモリ領域をダンプする  
ようにする

1. `/proc/self/coredump_filter` に1を書く  
`$ echo 1 > /proc/self/coredump_filter`
2. 複数の子プロセスをforkするアプリを実行する  
`$ ./start_app`
3. 共有メモリ領域をダンプさせる子プロセス (PID 1234  
とする) に対して、`coredump_filter` を3に書き換える  
`$ echo 3 > /proc/1234/coredump_filter`

PID 1000のプロセスに対して、フルダンプするようにする

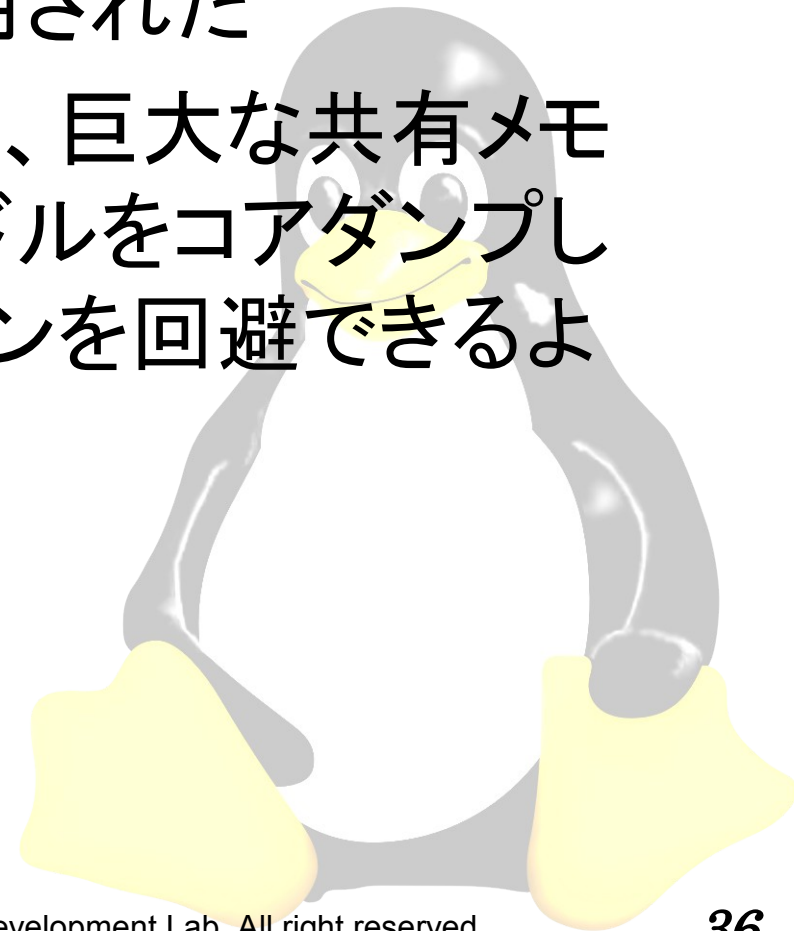
1. `/proc/1000/coredump_filter` に0xf (15)を書く  
`$ echo 0xf > /proc/1000/coredump_filter`



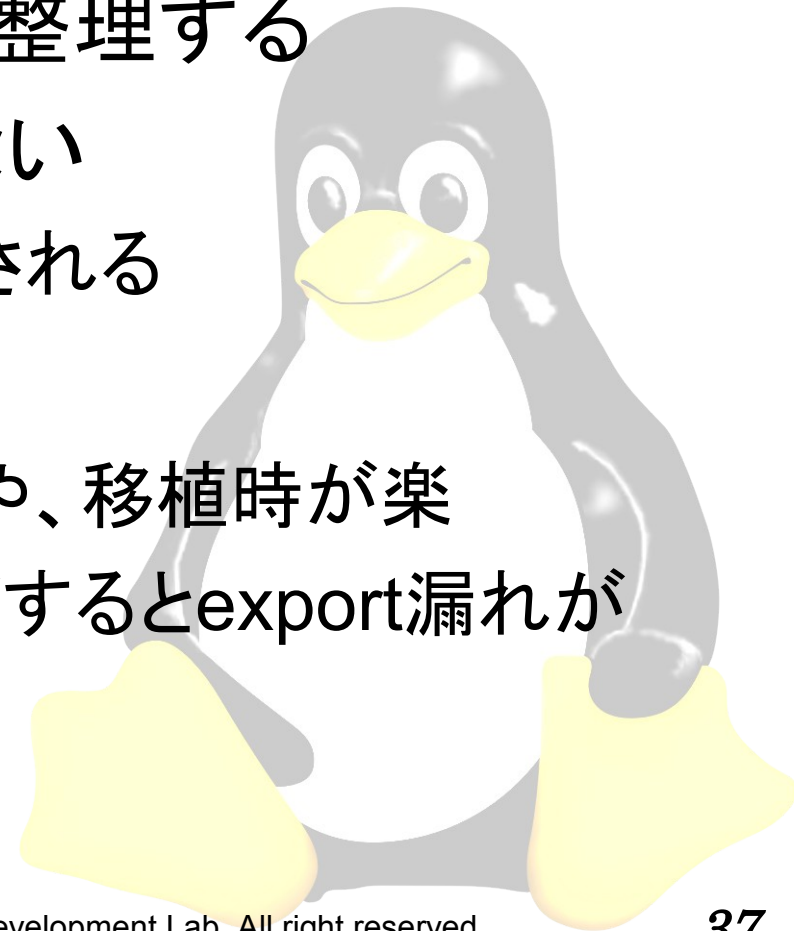
- coredump masking機能とは
- コアダンプ処理における問題
- 問題解決のアプローチ
- 設計と実装およびLKML上の議論
- RHEL 5への移植に向けて
- 使い方
- **まとめ**



- メモリ領域の種類ごとに、コアダンプする／しないを設定する機能を開発し、upstream kernel 2.6.23-rc1に採用された
- 同機能を利用することで、巨大な共有メモリを使用するアプリ／ミドルをコアダンプしてもシステムスローダウンを回避できるようになった

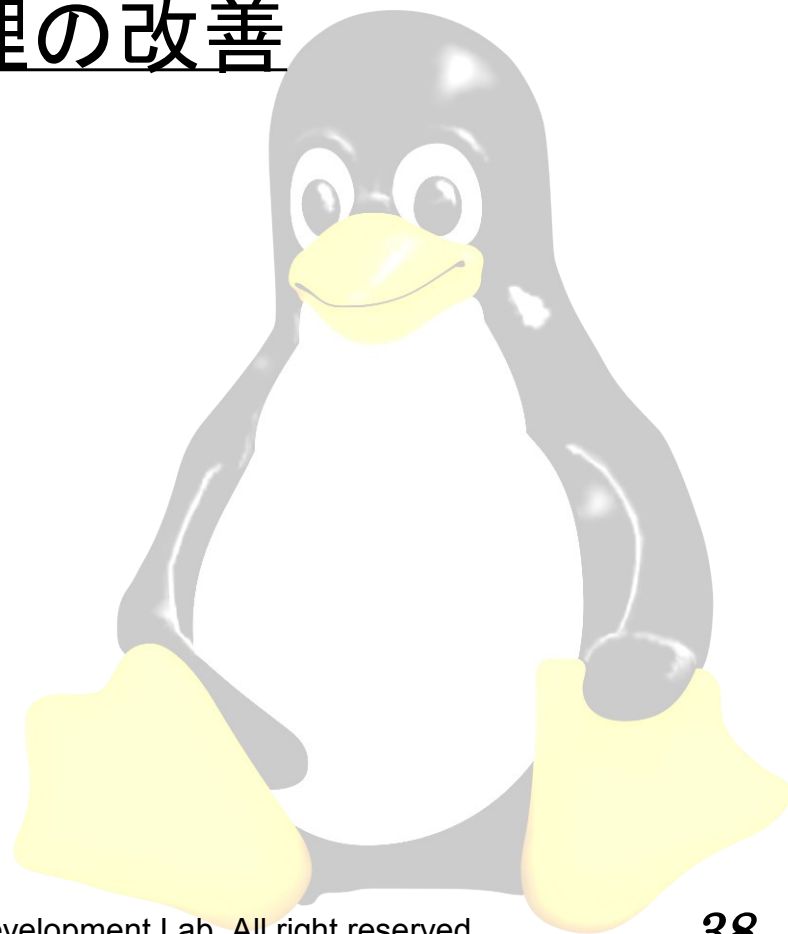


- 機能をカーネルで実現する必要性を明確に述べる
- 各々の提案の長所/短所を整理する
- 反対する人がいても諦めない
  - メンテナが納得すれば採用される
- テストはしっかりやる
  - 自動化しておくとりテイク時や、移植時が楽
  - make allmodconfigでビルドするとexport漏れが発見できる



1. coredump masking

2. ネットワークメモリ管理の改善



1. 何が変わったか？
2. 開発の動機
3. netdevでの議論の推移
4. 実装
5. インターフェース変更によって発生したバグ
6. 使い方
7. まとめ



- ユーザにとって

- UDPでもカーネルが使用するメモリの上限設定が可能に

- TCPでは、メモリ使用量の上限が設定可能だった

- カーネル開発者にとって

- stream型プロトコル専用だったメモリアカウントインターフェースが、stream-datagram汎用になり、統合された



- 問題

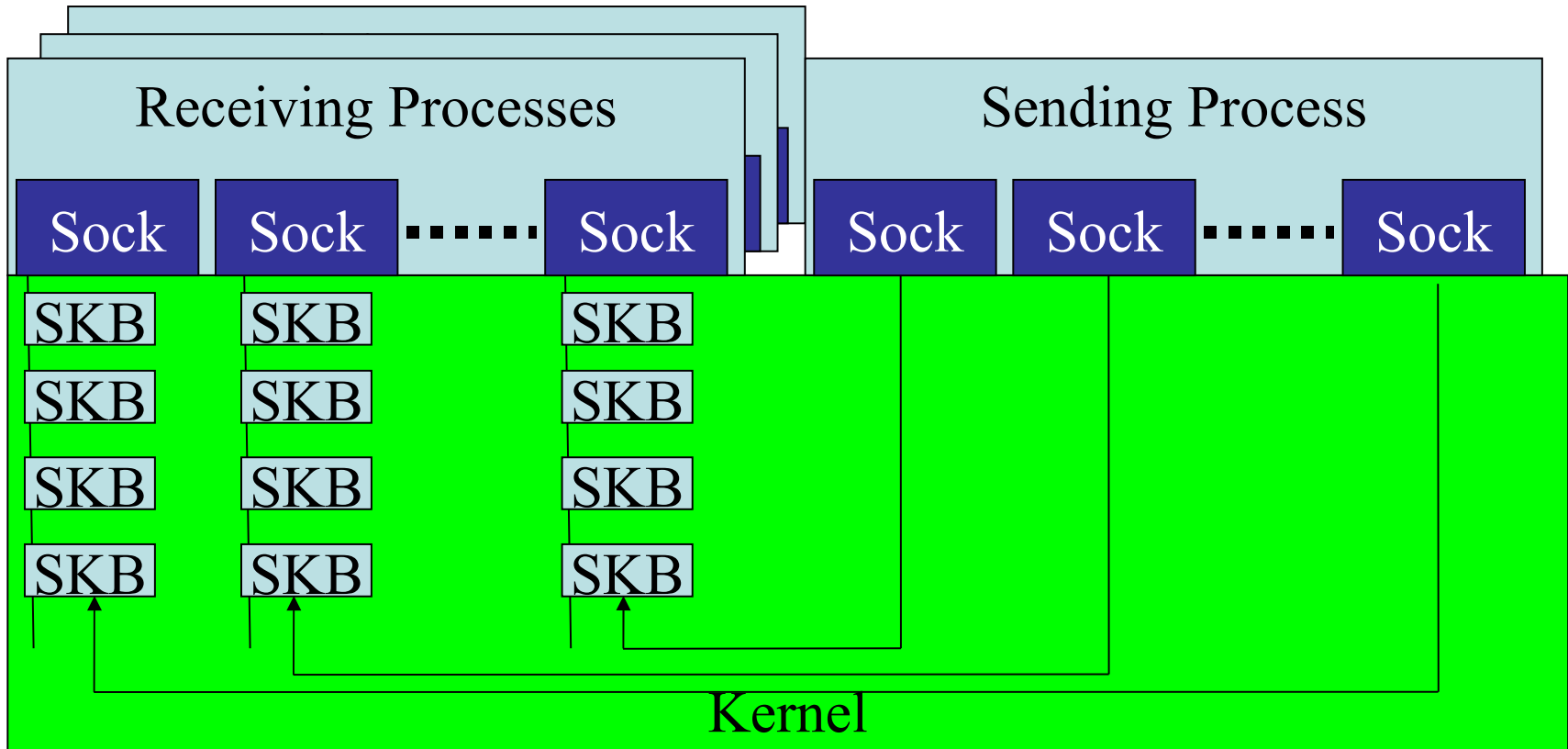
- 悪意あるユーザがルート権限なしに、大量の(事実上無制限の)カーネルページをピンダウンすることができる

- シナリオ

- 受信用プロセスとして、数万のプロセスを生成
- 各プロセスが1000個のUDPソケット作成
- 送信用プロセスが各ソケットにメッセージ送信
- 受信用プロセスは受信(recvmsg)を行わない

大量のメモリがピンダウン⇒システムダウン！

- 受信キューに大量のソケットバッファが滞留



UDPにもメモリ使用量の上限が必要

- 当初の実装方針
  - メモリアカウンティングはシステムワイド
  - TCPのメモリアカウンティングとは別にUDPにアカウンティングを実装
- 問題点
  - システムワイドなアカウンティングだけでは、メモリを使い切ると通信不可
  - 新たなアトミックカウンタ導入による性能低下
  - datagram用関数の一部はstreamからも使用

システムワイドのアカウントティングでは、上限に達した場合、通信は出来なくなってしまうため、問題の解決にならない

– Chris Wright

ユーザ毎の課金の方が良い

– Herbert Xu

- 事前に悪意あるユーザが誰で、何人いるか特定できない
- ユーザ毎の上限を低く設定することに

システムワイドの上限は維持しユーザ毎の最低使用量を認める

アトミックカウンタを新たに導入するとスケーラビリティの問題が発生する  
- Andi Kleene他

Christoph LameterがやっているCPU毎に課金する領域を作る方法をTCP/UDPにも応用しては？  
- Eric Dumazet

- TCP/UDPにCPU毎の課金領域までには必要ないのでは？
- メモリ確保はsk\_forward\_allocに統合して、メモリ確保開放の際は、ソケット毎のロックを取得する方法を検討してはどうか？(Herbert Xu案)

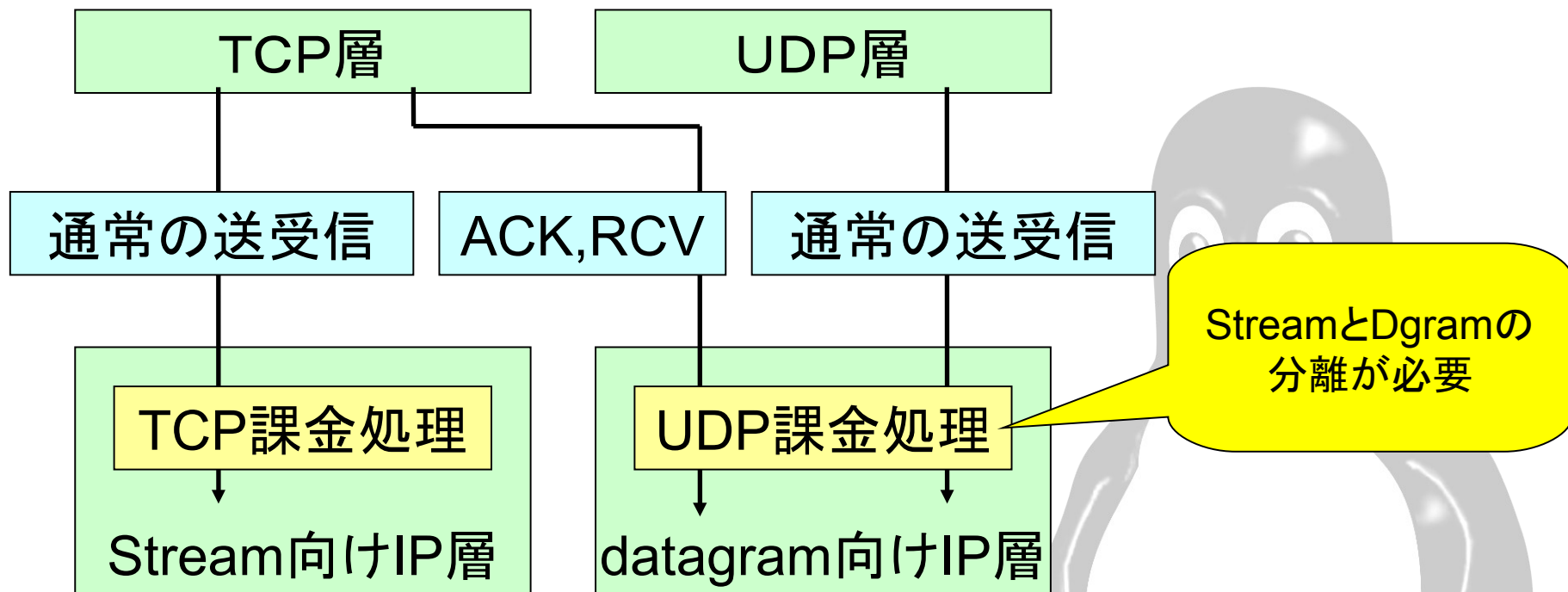
Sk\_forward\_alloc統合案を採用

(if (sk->sk\_type == SOCK\_DGRAM)というコードに対して)  
なぜstreamとdatagramを分離するようなコードが必要なのか？  
UDP課金コードのほとんどは、TCPからのコピーなのにさらに分岐を必要とするのであれば、この部分こそ改善の必要がある  
- David Miller

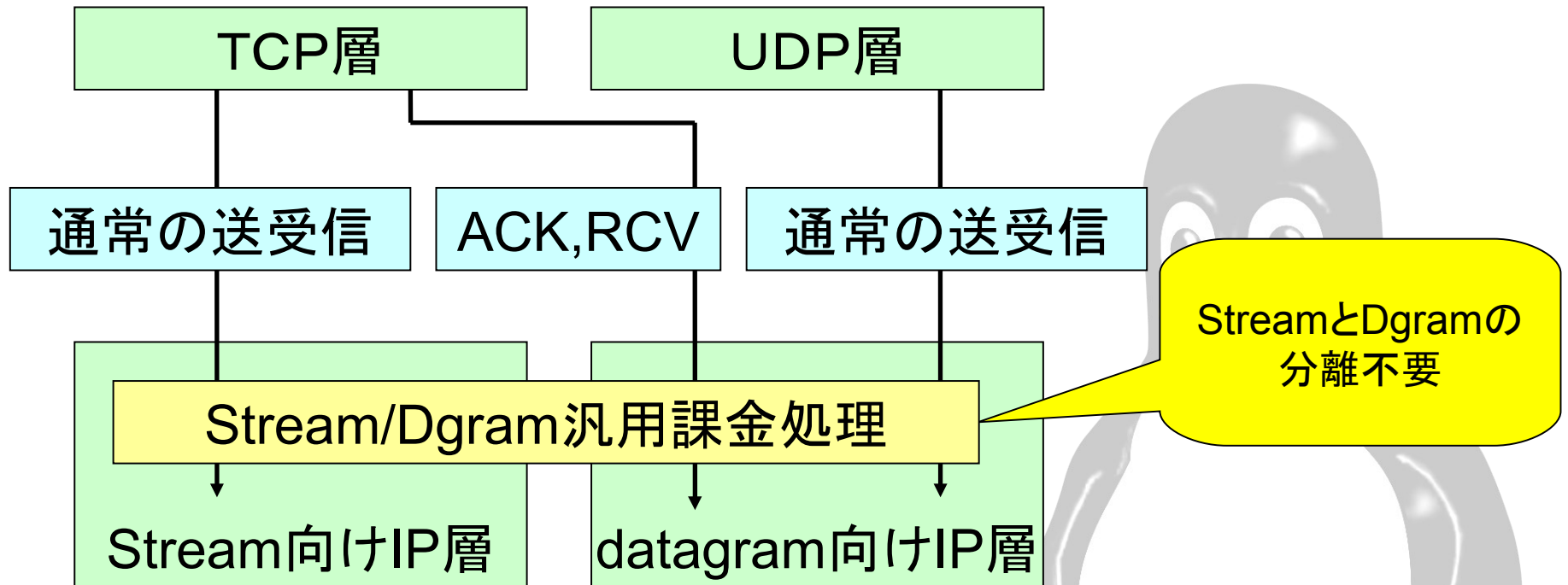
- 変更が広範囲になること(その結果、パッチが採用されなくなることを恐れて、変更範囲を限定
- その裏目に出て、汚いコードに

TCPとUDPのアカウントティングコードの統合を決断

- 当初のパッチ



- 現在のパッチ



- UDPのIPv4およびIPv6をサポート
- 2.6.25-rc1でupstreamへ

- 名称変更:

- sk\_stream\_free\_skb() -> sk\_wmem\_free\_skb()
- \_\_sk\_stream\_mem\_reclaim() -> \_\_sk\_mem\_reclaim()
- sk\_stream\_mem\_reclaim() -> sk\_mem\_reclaim()
- sk\_stream\_mem\_schedule -> \_\_sk\_mem\_schedule()
- sk\_stream\_pages() -> sk\_mem\_pages()
- sk\_stream\_rmem\_schedule() -> sk\_rmem\_schedule()
- sk\_stream\_wmem\_schedule() -> sk\_wmem\_schedule()
- sk\_charge\_skb() -> sk\_mem\_charge()

- 削除:

- sk\_stream\_rfree(): sock\_rfree()に統合
- sk\_stream\_set\_owner\_r(): skb\_set\_owner\_r()に統合
- sk\_stream\_mem\_schedule()

- 追加:

- sk\_has\_account(): アカウンティングをサポートしているか判定
- sk\_mem\_uncharge(): sk\_mem\_charge()の反対の機能

ルート権限で/proc/sys/net/ipv4/以下に新設されたudp\_mem, udp\_rmem\_min, udp\_wmem\_minに値を書き込む[単位:ページ]

## 1. udp\_mem

3つの値を持つベクタになっている:

一番目:メモリ使用量の適正化を開始

二番目:TCPにインターフェースを合わせるためのダミー

三番目:UDPメモリ使用量の上限

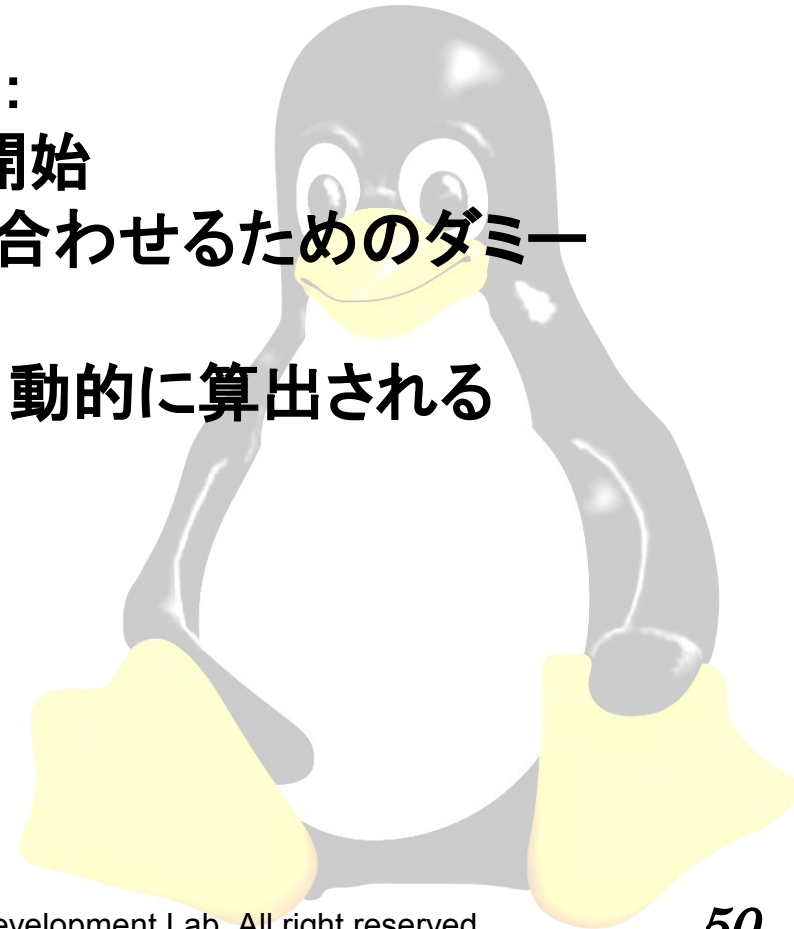
デフォルト値はメモリ搭載量から自動的に算出される

## 2. udp\_rmem\_min

UDP受信用メモリの最小値

## 3. udp\_wmem\_min

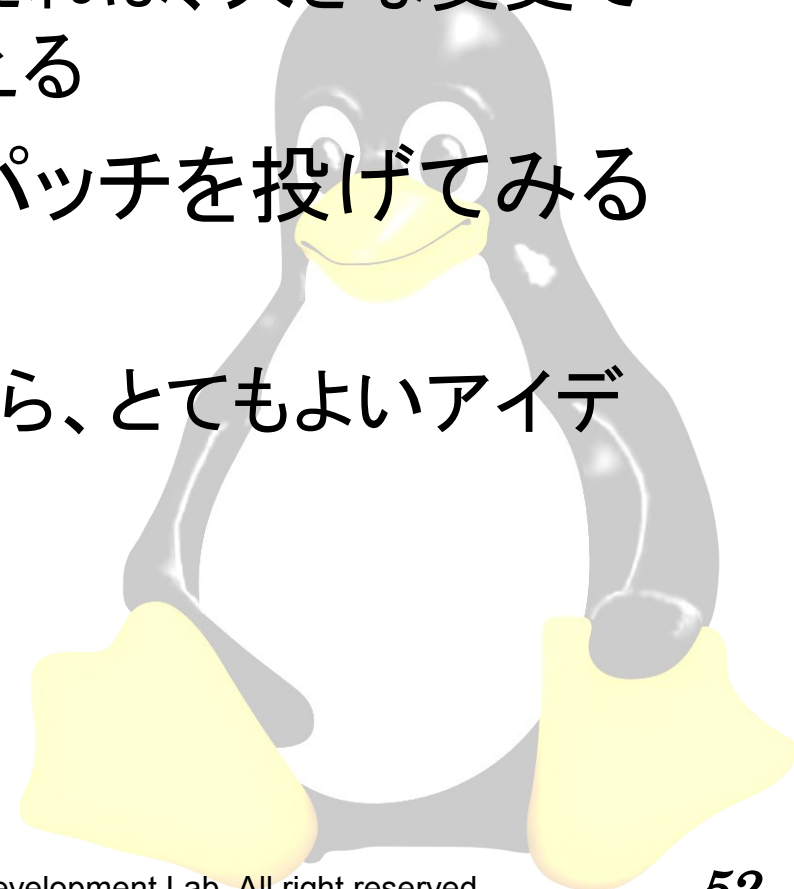
UDP送信用メモリの最小値



- UDP(IPv4/IPv6)のメモリ使用量に上限を設定できるようになった
- プロトコルに対するメモリアカウンティング向けインターフェースがstream/datagramで統合
- 他のメモリアカウンティングに対応していないプロトコルでアカウンティングを実装することが容易になった



- 大きな変更を恐れず、最初から問題点をすべて相談したほうが良い
  - 必要性さえ理解してもらえれば、大きな変更であっても受け入れてもらえる
- 早いうちにupstreamにパッチを投げてみる  
ことが重要
  - 様々な知見を持った人から、とてもよいアイデアがもらえる



- 一部のプロトコル／ドライバでメモリアカウ  
ンティング(`skb->truesize`)にバグが潜在し  
ているケースがある(`netlink`等)
- `skb->truesize`の書き換えをプロトコルやド  
ライバが直接行わないようインターフェー  
スを切る方法を提案する



- *“The fact is that the networking does not participate with the rest of the system wrt. memory pressure callbacks. That's the problem.” – David Miller – ([netdev@vger.linux.org](mailto:netdev@vger.linux.org):9/27/2007)*
- 将来:
  - メモリプレッシャコールバックによるメモリアカウンティングとドロップ
  - 単位時間当たりのコールバック回数に基づく、動的な閾値
- 課題:
  - 既にACKを返してしまったTCPパケットをどうやって取り扱うか

- Linuxは、日本およびその他の国における Linus Torvalds氏の登録商標です。
- その他の会社や製品名称に言及したものは、それぞれの所有者の商標です。





ご清聴ありがとうございます

