Call of DragonFly 〜 DragonFly の呼び声

Naohiro Aota <naota@elisp.net>

2011 年 02 月 20 日

自己紹介

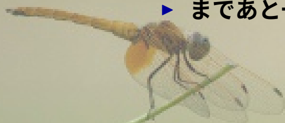
- 名前 青田直大
- 所属
  - 大阪大学
    - 3 回と 4 回の狭間

自己紹介

- 名前 青田直大
- 所属
  - 大阪大学
    - 3 回と 4 回の狭間
  - Gentoo JP developer

自己紹介

- 名前 青田直大
- 所属
  - 大阪大学
    - 3 回と 4 回の狭間
  - Gentoo JP developer
  - Gentoo official developer…
    - まであと一歩



自己紹介

- 名前 青田直大
- 所属
  - 大阪大学
    - 3 回と 4 回の狭間
  - Gentoo JP developer
  - Gentoo official developer···
    - まであと一歩
- いろいろ
  - GSoC 2010 参加
    - Gentoo DragonFly を作りました

自己紹介

- 名前 青田直大
- 所属
  - 大阪大学
    - 3 回と 4 回の狭間
  - Gentoo JP developer
  - Gentoo official developer…
    - まであと一歩

- いろいろ
  - GSoC 2010 参加
    - Gentoo DragonFly を作りました
  - セプキャン 2010 Linux 組

自己紹介

- 名前 青田直大
- 所属
  - 大阪大学
    - 3 回と 4 回の狭間
  - Gentoo JP developer
  - Gentoo official developer…
    - まであと一歩
- いろいろ
  - GSoC 2010 参加
    - Gentoo DragonFly を作りました
  - セプキャン 2010 Linux 組
  - AsiaBSDCon 2011 で発表します

自己紹介

- 名前 青田直大
- 所属
  - 大阪大学
    - 3 回と 4 回の狭間
  - Gentoo JP developer
  - Gentoo official developer…
    - まであと一歩
- いろいろ
  - GSoC 2010 参加
    - Gentoo DragonFly を作りました
  - セプキャン 2010 Linux 組
  - AsiaBSDCon 2011 で発表します

BSD なの?

- FreeBSD からでた BSD の一種
  - FreeBSD 4.8 からの分岐
- スレッドや SMP に対する方向性の違い

カーネル プロセスとか

- 2 つのスケジューラ
  - LWKT スケジューラ
    - LWKT (Light Weight Kernel Thread) をふくむ全ての実行可能なものをスケジュールする
  - User Thread スケジューラ
    - 実行可能なユーザスレッドを選択
    - LWKT を使って走らせる

カーネル プロセスとか

- 2 つのスケジューラ
  - LWKT スケジューラ
    - LWKT (Light Weight Kernel Thread) をふくむ全ての実行可能なものをスケジュールする
  - User Thread スケジューラ
    - 実行可能なユーザスレッドを選択
    - LWKT を使って走らせる
- User Thread
  - 1 つのプロセスに 1 以上の LWP (Light Weight Process) がある

カーネル プロセスとか

- 2 つのスケジューラ
  - LWKT スケジューラ
    - LWKT (Light Weight Kernel Thread) をふくむ全ての実行可能なものをスケジュールする
  - User Thread スケジューラ
    - 実行可能なユーザスレッドを選択
    - LWKT を使って走らせる
- User Thread
  - 1 つのプロセスに 1 以上の LWP (Light Weight Process) がある
- process checkpoint
  - SIGCKPT を送ることで process のスナップショットができる
  - どうにもうまくいったりいかなかったり…

カーネル メモリとか

- 2 つのメモリアロケータ
  - kmalloc(): 基本的なアロケータ
    - cpu ごとの slab アロケータを使う
    - 基本的にロックなし

カーネル メモリとか

- 2 つのメモリアロケータ
  - kmalloc(): 基本的なアロケータ
    - cpu ごとの slab アロケータを使う
    - 基本的にロックなし
  - objcache
    - object oriented …らしい
    - コンストラクタ・デストラクタとか設定できるみたい
    - (Linux の slab ぽい気がするんだが)
    - 大きなものに使われるらしい
    - これもロックなし

ファイルシステム

- devfs
  - デバイスファイルシステム
  - Linux でいえば udev
  - ドライブのシリアルナンバが見れる
  - なので、ディスクをどこにさしても大丈夫

ファイルシステム

- devfs
  - デバイスファイルシステム
  - Linux でいえば udev
  - ドライブのシリアルナンバが見れる
  - なので、ディスクをどこにさしても大丈夫
- NFS
  - async らしい…
  - LWKT が 2 個ほど動いてる

ファイルシステム

- devfs
  - デバイスファイルシステム
  - Linux でいえば udev
  - ドライブのシリアルナンバが見れる
  - なので、ディスクをどこにさしても大丈夫
- NFS
  - async らしい…
  - LWKT が 2 個ほど動いてる
- HAMMER
  - ZFS や btrfs の対抗馬!
  - fsck なしとか robust とか
  - snapshot 機能とか undo とか
  - まぁそんな感じですね…

ファイルシステム

- devfs
  - デバイスファイルシステム
  - Linux でいえば udev
  - ドライブのシリアルナンバが見れる
  - なので、ディスクをどこにさしても大丈夫
- NFS
  - async らしい…
  - LWKT が 2 個ほど動いてる
- HAMMER
  - ZFS や btrfs の対抗馬!
  - fsck なしとか robust とか
  - snapshot 機能とか undo とか
  - まぁそんな感じですね…
  - cron job 動かしておかないと大変なことに!

ファイルシステム その2

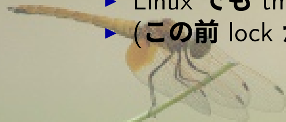
- nullfs
  - Linux の loopback 相当らしい
  - ただ bind 相当のこともできるようだ

ファイルシステム その2

- nullfs
  - Linux の loopback 相当らしい
  - ただ bind 相当のこともできるようだ
- tmpfs
  - 空きメモリと swap を使ったファイルシステム
  - Linux でも tmpfs

ファイルシステム その2

- nullfs
  - Linux の loopback 相当らしい
  - ただ bind 相当のこともできるようだ
- tmpfs
  - 空きメモリと swap を使ったファイルシステム
  - Linux でも tmpfs
  - (この前 lock がへったよーと言ってましたね)



ファイルシステム その2

- nullfs
  - Linux の loopback 相当らしい
  - ただ bind 相当のこともできるようだ
- tmpfs
  - 空きメモリと swap を使ったファイルシステム
  - Linux でも tmpfs
  - (この前 lock がへったよーと言ってましたね)
- variant symlink
  - 評価時にリンク先が決定する symlink
  - くわしくは次のページで!

variant symlink

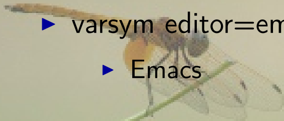
- ln -s 'best_editor_${editor}' best_editor
- echo Emacs > best_editor_emacs
- echo Vim > best_editor_vim

variant symlink

- ln -s ‘best_editor_${editor}’ best_editor
- echo Emacs > best_editor_emacs
- echo Vim > best_editor_vim
- ls -l
  - best_editor -> best_editor_${editor}
  - best_editor_emacs
  - best_editor_vim

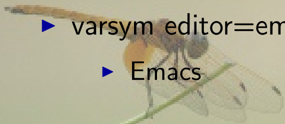
variant symlink

- ln -s ‘best_editor_${editor}’ best_editor
- echo Emacs > best_editor_emacs
- echo Vim > best_editor_vim
- ls -l
  - best_editor -> best_editor_${editor}
  - best_editor_emacs
  - best_editor_vim
- varsym editor=emacs; cat best_editor
  - Emacs



variant symlink

- ln -s 'best_editor_${editor}' best_editor
- echo Emacs > best_editor_emacs
- echo Vim > best_editor_vim
- ls -l
  - best_editor -> best_editor_${editor}
  - best_editor_emacs
  - best_editor_vim



- varsym editor=emacs; cat best_editor
  - Emacs
- varsym editor=vim; cat best_editor
  - Vim

variant symlink

- ln -s 'best_editor_${editor}' best_editor
- echo Emacs > best_editor_emacs
- echo Vim > best_editor_vim
- ls -l
  - best_editor -> best_editor_${editor}
  - best_editor_emacs
  - best_editor_vim
- varsym editor=emacs; cat best_editor
  - Emacs
- varsym editor=vim; cat best_editor
  - Vim
- **エディタ戦争が終結した瞬間である**

ディスク

- dsched
  - I/O **スケジューラ**
  - rtprio, idprio
    - **プロセスがいつはしるかの制御**
- swapcache
  - SSD **上の** swap **にファイルシステムのデータ・メタデータをキャッシュする**
  - swap **上限は** 32G (32bit), 512G (64bit)

ディスク

- dsched
  - I/O **スケジューラ**
  - rtprio, idprio
    - **プロセスがいつはしるかの制御**
- swapcache
  - SSD **上の** swap **にファイルシステムのデータ・メタデータをキャッシュする**
  - swap **上限は** 32G (32bit), 512G (64bit)
  - (**いつのまにか** swapoff **ができるようになってた**…!)

目標

- 小さくてメンテしやすいシステム

目標

- 小さくてメンテしやすいシステム
  - (それみんな目指して は いますよねー :()

目標

- 小さくてメンテしやすいシステム
  - (それみんな目指して は いますよねー :()
- キャッシュモデル
  - MESI をファイルやネットワークに拡張
  - ロックを排除していきたい、ということなのだろう…
- IO デバイスモデルの刷新
  - IO がマルチスレッドに動くようにしたい、らしい?
  - Linux の IO は disk ごとになってたっけ…?

目標

- パッケージ
  - 別バージョンを同時にインストールできるように

目標

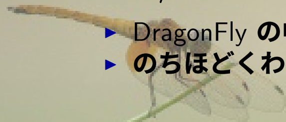
- パッケージ
  - 別バージョンを同時にインストールできるように
  - (それ Gentoo で…)

目標

- パッケージ
  - 別バージョンを同時にインストールできるように
  - (それ Gentoo で…)
- ポート/メッセージングモデル
  - DragonFly の中核をになう、と言っても過言ではない

目標

- パッケージ
  - 別バージョンを同時にインストールできるように
  - (それ Gentoo で…)
- ポート/メッセージングモデル
  - DragonFly の中核をになう、と言っても過言ではない
  - のちほどくわしく!! チャンネルはそのまま!!

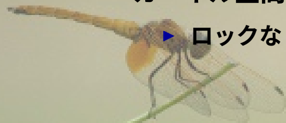


目標

- LWKT
  - 各 CPU ごとに 1 つの LWKT スケジューラ
    - 特別な場合を除いて CPU 間の移動がない
    - マルチプロセッサ用にロックをとらずにスケジューリングが可能

目標

- LWKT
  - 各 CPU ごとに 1 つの LWKT スケジューラ
    - 特別な場合を除いて CPU 間の移動がない
    - マルチプロセッサ用にロックをとらずにスケジューリングが可能
  - カーネル空間で動いている時に CPU を移動しない
    - ロックなしで CPU ごとのグローバルなキャッシュを使える



目標

- LWKT
  - 各 CPU ごとに 1 つの LWKT スケジューラ
    - 特別な場合を除いて CPU 間の移動がない
    - マルチプロセッサ用にロックをとらずにスケジューリングが可能
  - カーネル空間で動いている時に CPU を移動しない
    - ロックなしで CPU ごとのグローバルなキャッシュを使える
  - クリティカルセクションに入ってない時はいつでも割り込み可能
    - 割り込みが速い

目標

- IPI
  - CPU 間のメッセージ
  - async に動く

目標

- IPI
  - CPU 間のメッセージ
  - async に動く
- User API
  - 全てのシステムコールをメッセージベースに
  - 引数データを構造化して渡す
  - (旧来の) システムコールをメッセージに変換するレイヤを作る

ディストリビューション

- DragonFly
  - DragonFly 2.8

ディストリビューション

- DragonFly
  - DragonFly 2.8
- Gentoo/DragonFly
  - DragonFly 2.6

メッセージ

- DragonFly **の中核**: **メッセージ**
- **ソースコードは** /usr/src/sys/kern/lwkt_msgport.c **らへんに**

デモ

- 実際にどういうふうにメッセージがとびかっているか見てみましょう

API

- void lwkt_sendmsg(lwkt_port_t, lwkt_msg_t)
  - **非同期メッセージ送信**
- int lwkt_domsg(lwkt_port_t, lwkt_msg_t, int);
  - **同期メッセージ送信**

API

- void lwkt_sendmsg(lwkt_port_t, lwkt_msg_t)
  - **非同期メッセージ送信**
- int lwkt_domsg(lwkt_port_t, lwkt_msg_t, int);
  - **同期メッセージ送信**
- int lwkt_forwardmsg(lwkt_port_t, lwkt_msg_t);
  - **メッセージ転送**
- void lwkt_abortmsg(lwkt_msg_t);
  - **メッセージの中止**

非同期メッセージ送信

- メッセージの返信先が設定されていること
- メッセージが送信されていないことを確認して

非同期メッセージ送信

- メッセージの返信先が設定されていること
- メッセージが送信されていないことを確認して
- 返信中・同期待ち・未送信フラグを落として

非同期メッセージ送信

- メッセージの返信先が設定されていること
- メッセージが送信されていないことを確認して
- 返信中・同期待ち・未送信フラグを落として
- メッセージを送信

非同期メッセージ送信

- メッセージの返信先が設定されていること
- メッセージが送信されていないことを確認して
- 返信中・同期待ち・未送信フラグを落として
- メッセージを送信
- 非同期にできなかったら
  - エラー情報をつけて返信

同期メッセージ送信

- メッセージの返信先が設定されていること
- メッセージが送信されていないことを確認して

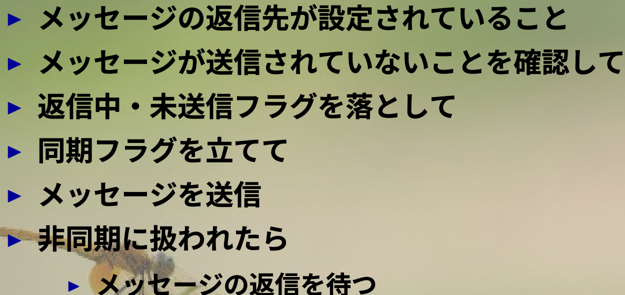
同期メッセージ送信

- メッセージの返信先が設定されていること
- メッセージが送信されていないことを確認して
- 返信中・未送信フラグを落として
- 同期フラグを立てて

同期メッセージ送信

- メッセージの返信先が設定されていること
- メッセージが送信されていないことを確認して
- 返信中・未送信フラグを落として
- 同期フラグを立てて
- メッセージを送信

同期メッセージ送信



- メッセージの返信先が設定されていること
- メッセージが送信されていないことを確認して
- 返信中・未送信フラグを落として
- 同期フラグを立てて
- メッセージを送信
- 非同期に扱われたら
  - メッセージの返信を待つ

同期メッセージ送信

- メッセージの返信先が設定されていること
- メッセージが送信されていないことを確認して
- 返信中・未送信フラグを落として
- 同期フラグを立てて
- メッセージを送信
- 非同期に扱われたら
  - メッセージの返信を待つ
- 同期的に扱われたら
  - 送信済み・返信中のフラグ立てる

メッセージの転送

- メッセージが配送・返信途中でないことを確認して

メッセージの転送

- メッセージが配送・返信途中でないことを確認して
- メッセージを送信

メッセージの転送

- メッセージが配送・返信途中でないことを確認して
- メッセージを送信
- メッセージが同期的に扱われたら
  - メッセージのもとの返信ポートに返信

メッセージの中止

- メッセージが中止可能であり
- まだ送信完了していなければ
  - メッセージの中止関数を呼び出す

そして内部

- と、ここまではフロントエンド

そして内部

- と、ここまではフロントエンド
- 実装の詳細はそれぞれのポートの中に
- ポートにはいくつかのバックエンドがある
  - 単一スレッドのポート

そして内部

- と、ここまではフロントエンド
- 実装の詳細はそれぞれのポートの中に
- ポートにはいくつかのバックエンドがある
  - 単一スレッドのポート
  - スピンロックでロックされるポート
  - serialize toke でロックされるポート

そして内部

- と、ここまではフロントエンド
- 実装の詳細はそれぞれのポートの中に
- ポートにはいくつかのバックエンドがある
  - 単一スレッドのポート
  - スピンロックでロックされるポート
  - serialize toke でロックされるポート
  - 受信のみのポート
  - 返信のみのポート
  - なにも受け付けない = panic するポート

そして内部

- と、ここまではフロントエンド
- 実装の詳細はそれぞれのポートの中に
- ポートにはいくつかのバックエンドがある
  - 単一スレッドのポート
  - スピンロックでロックされるポート
  - serialize toke でロックされるポート
  - 受信のみのポート
  - 返信のみのポート
  - なにも受け付けない = panic するポート
- ここでは簡単な「単一スレッドのポート」を取り上げます

単一スレッドのポート

- 特定のスレッドに属するポート
- 一個のスレッドしか見ない、と仮定されている
- ロックがいらない

lwkt_thread_getport

- ポートから次のメッセージを取り出す
  - なければ NULL
- 優先キュー・通常キューの順に poll して
- メッセージがあれば取り出す

lwkt_thread_putport

- ポートにメッセージを配送
- 同一 CPU 内なら
  - メッセージの優先度にあわせたキューに追加

lwkt_thread_putport

- ポートにメッセージを配送
- 同一 CPU 内なら
  - メッセージの優先度にあわせたキューに追加
  - 宛先 port で待機中スレッドがあれば
    - それを起こす

lwkt_thread_putport

- ポートにメッセージを配送
- 同一 CPU 内なら
  - メッセージの優先度にあわせたキューに追加
  - 宛先 port で待機中スレッドがあれば
    - それを起こす
- 別の CPU なら
  - IPI を使って送信

lwkt_thread_waitmsg

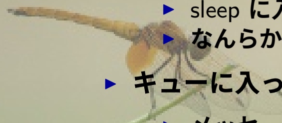
- 特定のメッセージを待機
- 前述の返信待ちとかに使う
- drop 可能でないことを確認して

lwkt_thread_waitmsg

- 特定のメッセージを待機
- 前述の返信待ちとかに使う
- drop 可能でないことを確認して
- メッセージが配送される前なら
  - 配送されるまでの間待つ
    - ポートに待機中フラグを立てて
    - sleep に入る
    - なんらかのメッセージが来たら起きる

lwkt_thread_waitmsg

- 特定のメッセージを待機
- 前述の返信待ちとかに使う
- drop 可能でないことを確認して
- メッセージが配送される前なら
  - 配送されるまでの間待つ
    - ポートに待機中フラグを立てて
    - sleep に入る
    - なんらかのメッセージが来たら起きる
  - キューに入ったら
    - メッセージを取り出す

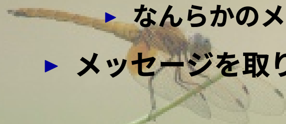


lwkt_thread_waitmsg

- 特定のメッセージを待機
- 前述の返信待ちとかに使う
- drop 可能でないことを確認して
- メッセージが配送される前なら
  - 配送されるまでの間待つ
    - ポートに待機中フラグを立てて
    - sleep に入る
    - なんらかのメッセージが来たら起きる
  - キューに入ったら
    - メッセージを取り出す
- すでに配送されてるなら
  - 単純にメッセージを取り出せばよい

lwkt_thread_waitport

- なんらかのメッセージが来るのを待つ
- poll してメッセージがない間
  - ポートに待機中フラグを立てて
  - sleep に入る
  - なんらかのメッセージが来たら起きる
- メッセージを取り出す

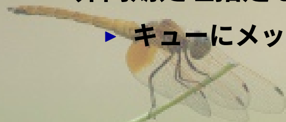


lwkt_thread_replyport

- 返信されたメッセージの処理
- 同期処理指定なら
  - ポートで待機中なら
    - スレッドを起こす

lwkt_thread_replyport

- 返信されたメッセージの処理
- 同期処理指定なら
  - ポートで待機中なら
    - スレッドを起こす
- 非同期処理指定なら
  - キューにメッセージ追加



lwkt_thread_replyport

- 返信されたメッセージの処理
- 同期処理指定なら
  - ポートで待機中なら
    - スレッドを起こす
- 非同期処理指定なら
  - キューにメッセージ追加
  - メッセージを返信・完了フラグ立てて

lwkt_thread_replyport

- 返信されたメッセージの処理
- 同期処理指定なら
  - ポートで待機中なら
    - スレッドを起こす
- 非同期処理指定なら
  - キューにメッセージ追加
  - メッセージを返信・完了フラグ立てて
  - ポートで待機中なら
    - スレッドを起こす

lwkt_thread_dropmsg

- メッセージを drop する
- キューからメッセージを取り出し
- 未配送にする

それでそれで?

- 全てがメッセージ通信に抽象化されてわかりやすい!
- debug しやすい!

それでそれで?

- 全てがメッセージ通信に抽象化されてわかりやすい!
- debug しやすい!

- …らしいですよ?