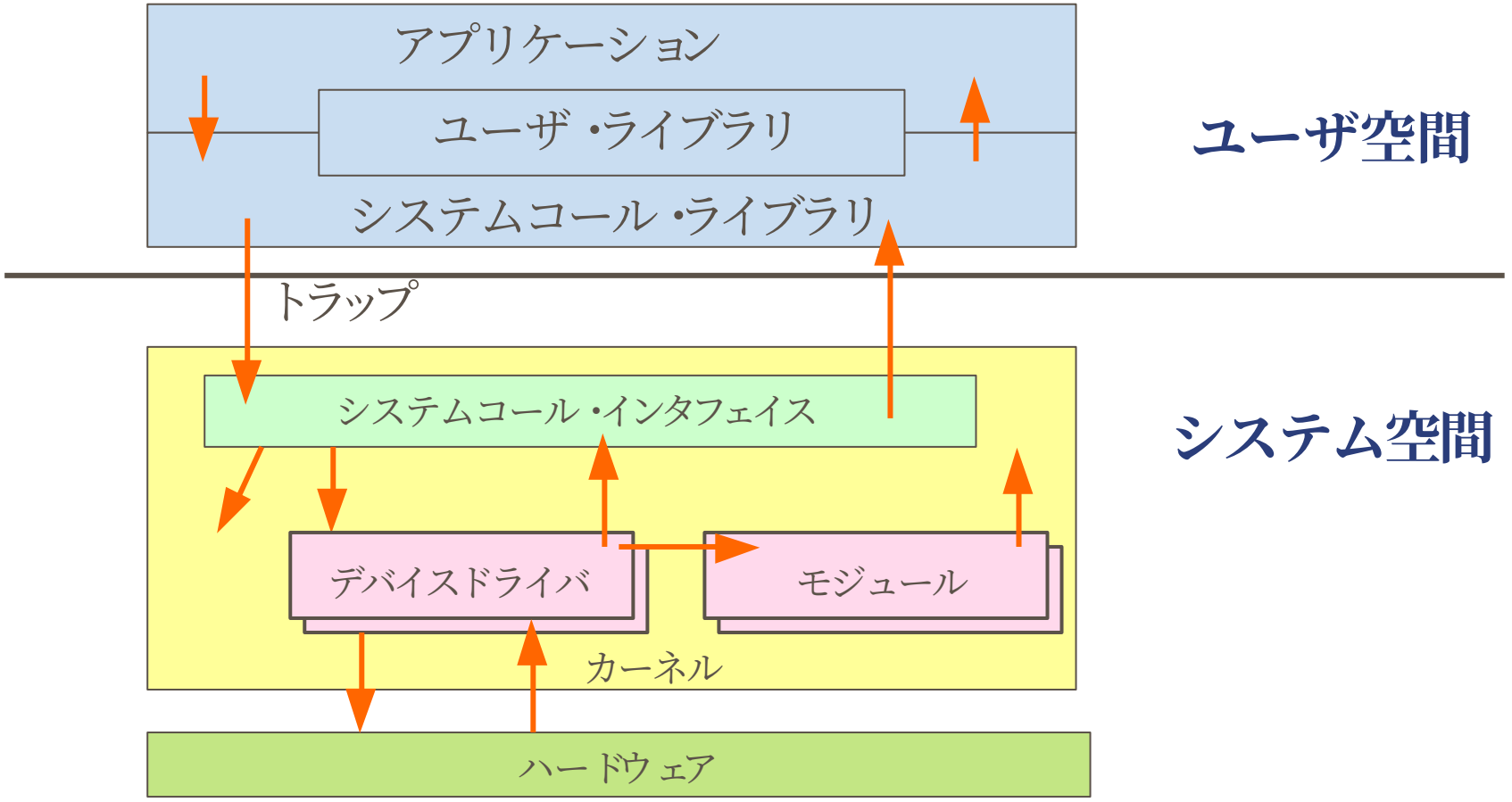
最新Linuxデバイスドライバ開発応用

- 新時代のドライバ / カーネル・モジュール開発 -

株式会社デバイスドライバーズ
日高亜友 info@devdrv.co.jp

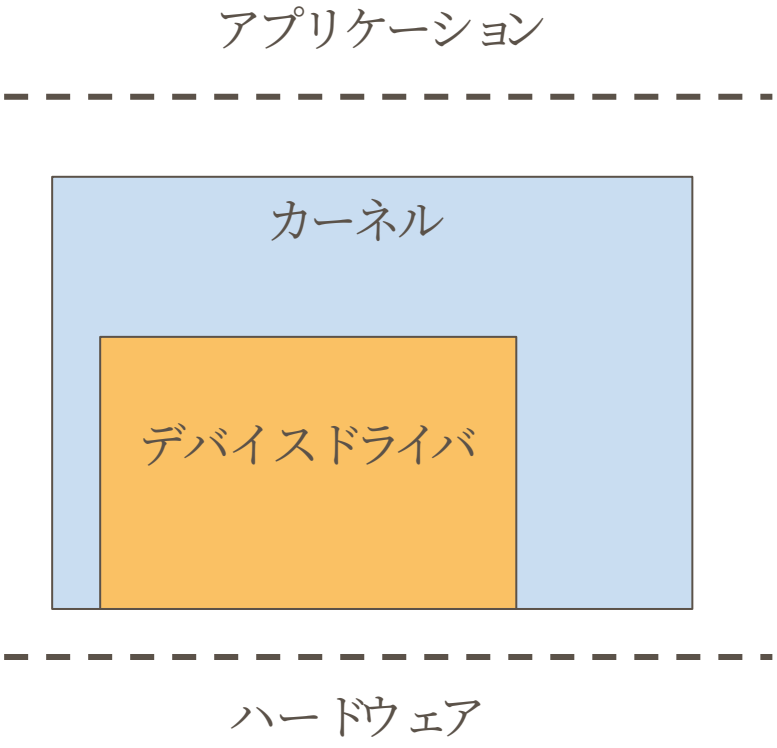
確認 :デバイスドライバ

- デバイス入出力システムコール処理の流れ

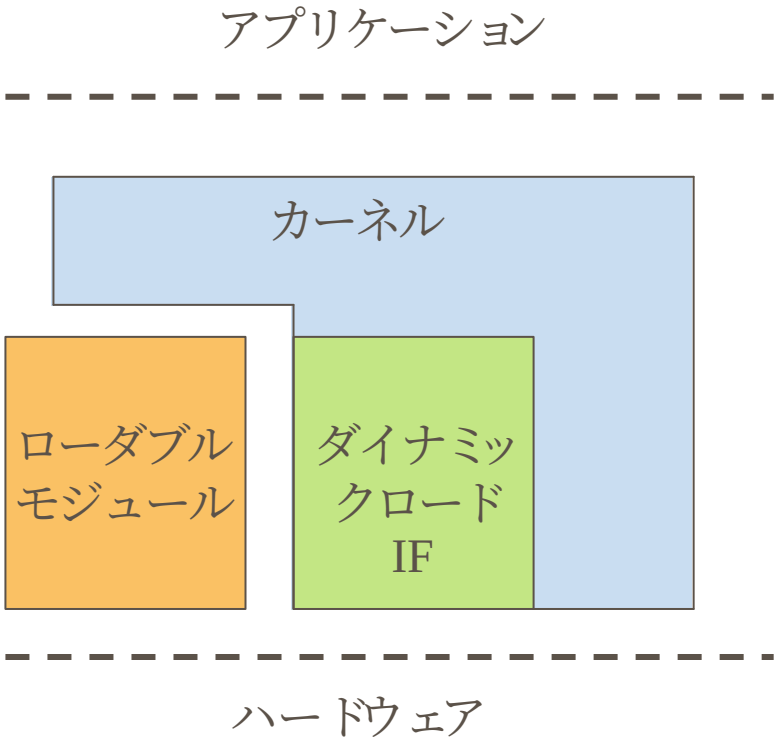


確認 :ローダブル •モジュール

スタティックリンクの ドライバ

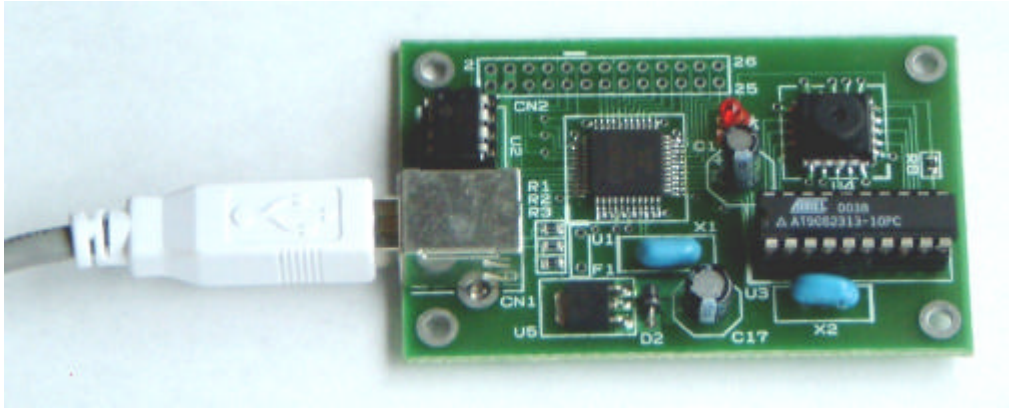


ローダブルモジュールの ドライバ



最新Linuxデバイスドライバ開発応用

- 現状と問題点
- 対応策
- デモンストレーション
- 今後の見通し



現状と問題点

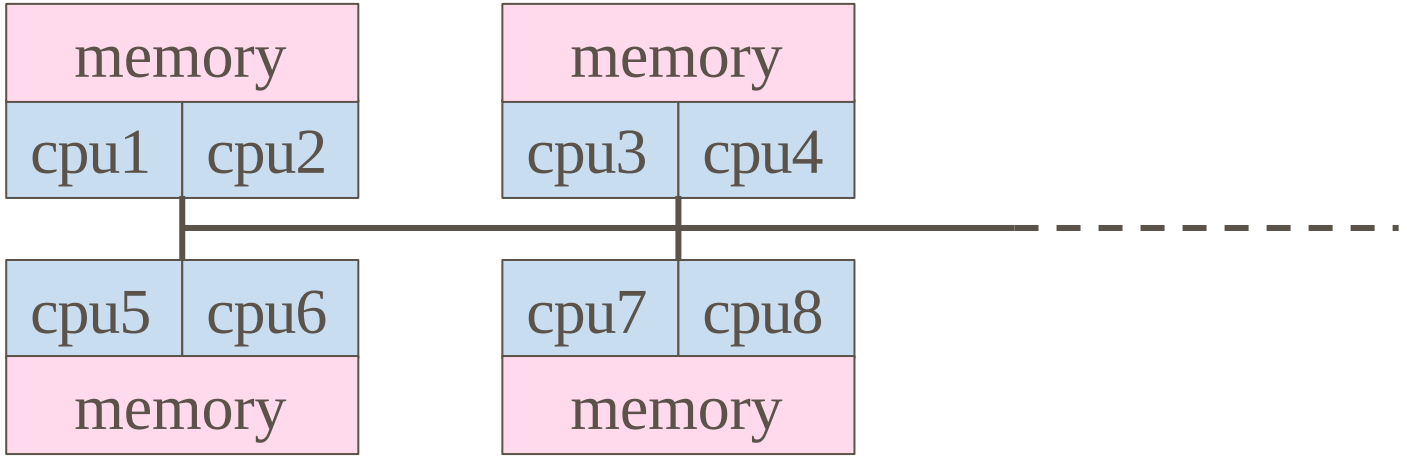
- ハードウェア進化への対応
  - プロセッサ技術の進化
  - バスとドライバの進化
- カーネル2.6の功罪
  - カーネル2.6のセールスポイント
  - カーネル2.6の問題点

プロセッサ技術の進化 :用語(1)

- 構成による分類
  - UP (Uni Processor)
  - SMP (Symmetric Multi Processor)
  - ASMP (Asymmetric Multiple Processor)
- 実装技術
  - SMT (Simultaneous Multi-Threading)
    - Intel Hyper-Threading
  - CMP (Chip Multi Processor)
  - VLIW (Very Long Instruction Word)

プロセッサ技術の進化 :用語(2)

- マルチプロセッサ •アーキテクチャ分類
  - UMA (uniformed memory access)
  - NUMA (non- uniformed memory access)
  - NORMA (no-remote memory access)
    - shared nothing, cluster, grid computer

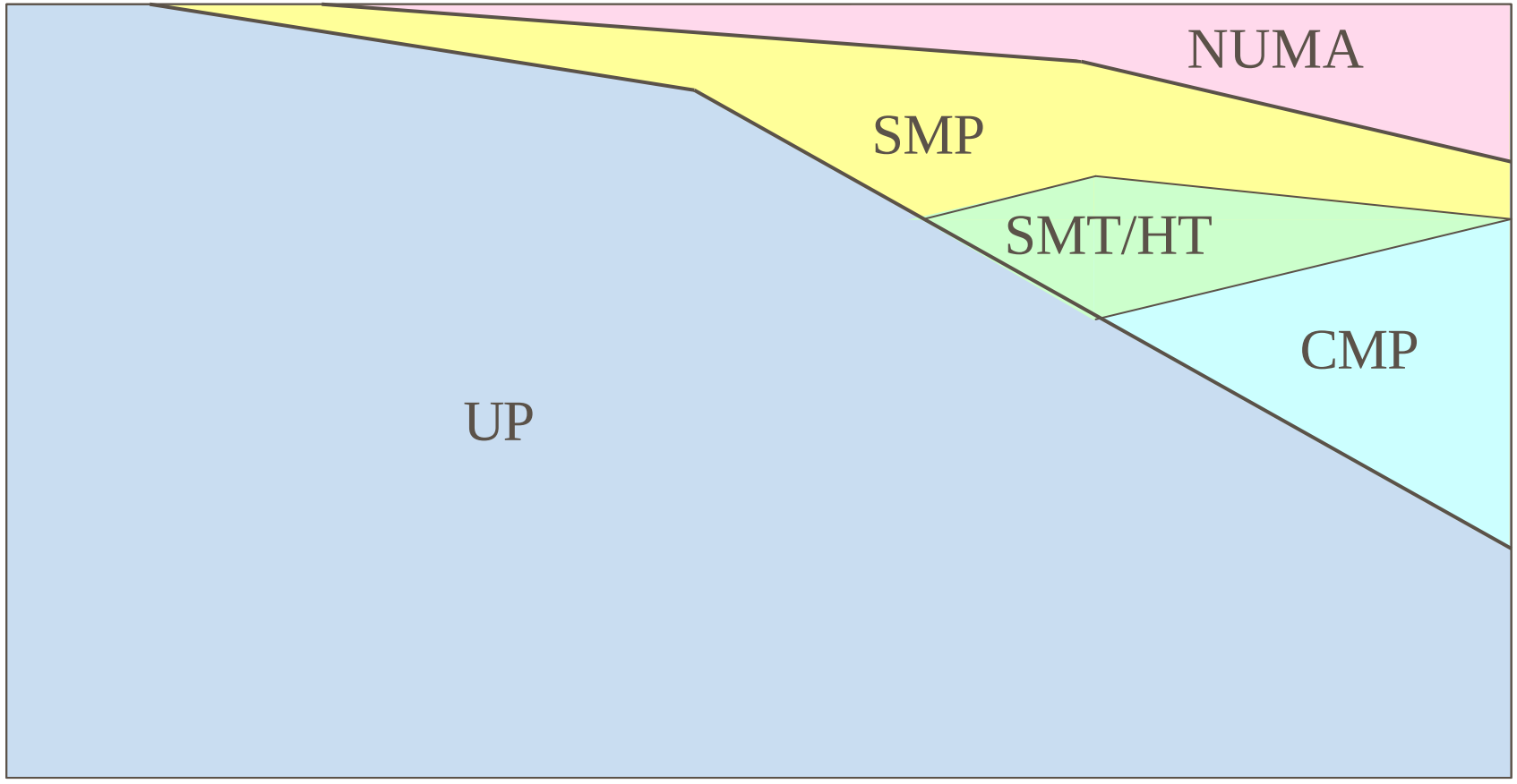


プロセッサ技術の進化

- CPUアーキテクチャの進化
  - ムーアの法則の継続
    - 増加したトランジスタ数の使い道は？
  - UPからCMPへ
    - 既存リソースの活用
  - VLIWは？
    - Transmeta vs. Intel ia64

マルチプロセッサ技術の進化予測

- 今はどこにいるか？

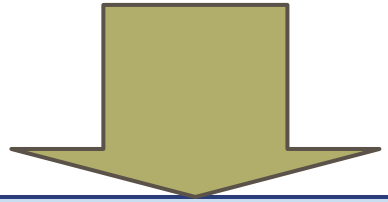


並列処理による性能向上

- UP
  - マシン語、uOPレベル（デコーダと演算器の仕事）
  - スーパースカラー
- VLIW
  - コンパイラレベル（コンパイラの仕事）
- マルチプロセッサ
  - ソースコードレベル（プログラマの仕事）
  - OSはサポートするだけ

バスとドライバの進化

- 新しいバス規格
  - SATA
  - PCI-Express
  - 802.11x
  - UWB (wireless USB)
- 現状のドライバ実装
  - レガシーデバイス
  - ブロックIO, Storage
  - 非HotPlug対応
  - 非PowerManagement



デバイスドライバ、カーネル・モジュールの開発負担

現状と問題点

- ハードウェア進化への対応
  - プロセッサ技術の進化
  - バスとドライバの進化

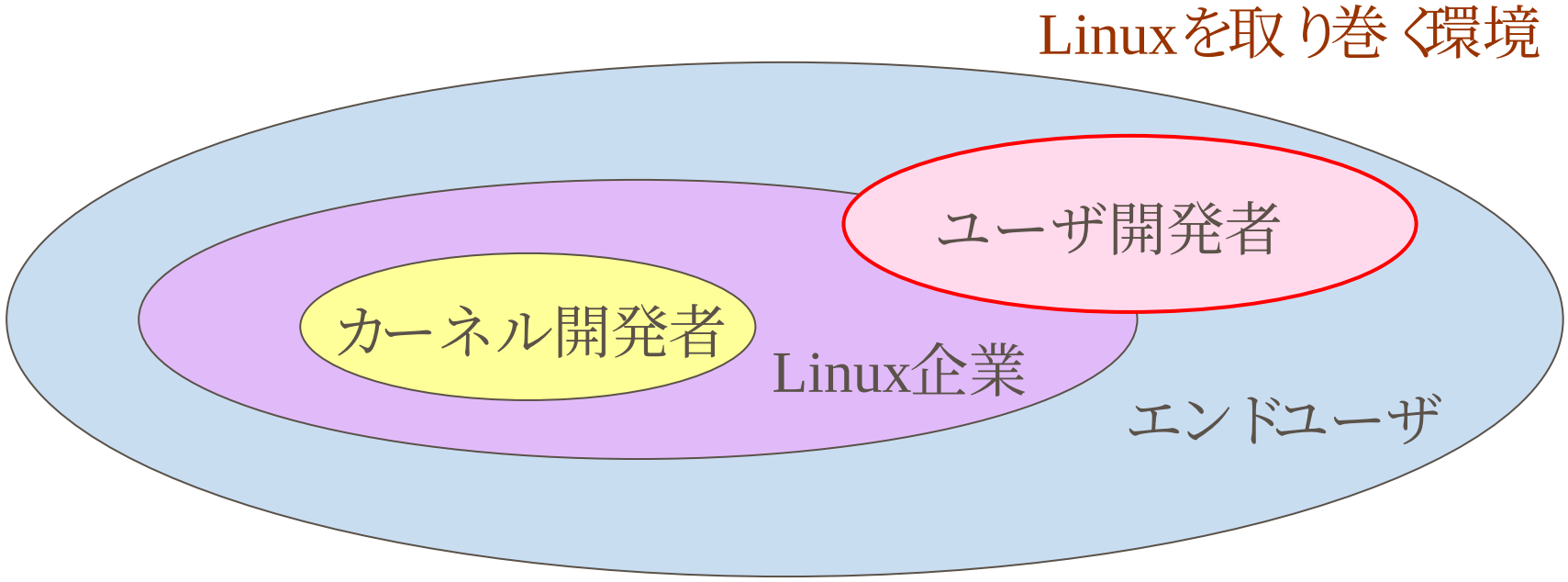
- カーネル2.6の功罪
  - カーネル2.6のセールスポイント
  - カーネル2.6の問題点

カーネル2.6の功罪

- カーネル2.6のセールスポイント
  - マルチCPU対応の強化とスケジューラの改良
    - NUMA, O(1)スケジューラ, プリエンプション・カーネル
  - メモリ管理の改良
  - ブロックIOの改良
    - IOスケジューラ
  - ネットワーク機能強化
  - 他プロジェクトのマージ
    - uclinux, SELINUX, oprofiler, …

カーネル2.6の功罪(2)

- カーネル2.6の問題点
  - 作り手側のメッセージ（情報）ばかりが強調されていて、ユーザ側開発者の立場に立ったメリット、デメリットが見えて来ない



ユーザ開発者にとってのメリット

- HZ(jiffies)の変更
  - jiffiesが64bitになってオーバーフローしなくなった
- ドライバ周りの整理と改良
  - USB, サウンド, IPV6, …
- SYSFSの恩恵
- パワーマネジメントの恩恵
- スケジューラの選択とチューニング（ができる）
- VLM/dm, IPV6, 機能強化と新機能

ユーザ開発者にとってのデメリット

- HZ(Jiffies)の変更
  - HZ=100→1000:チックカウントが10倍に増えた！
    - jiffies間隔: 10ms→1ms
    - 遅いCPUの採用は想定しない
- 2.4以前へのバックポートが保証されない

バックポートに関するLKMLでの議論

2.6を業務で使っている人はどの位いますか？

ユーザ開発者にとってのデメリット

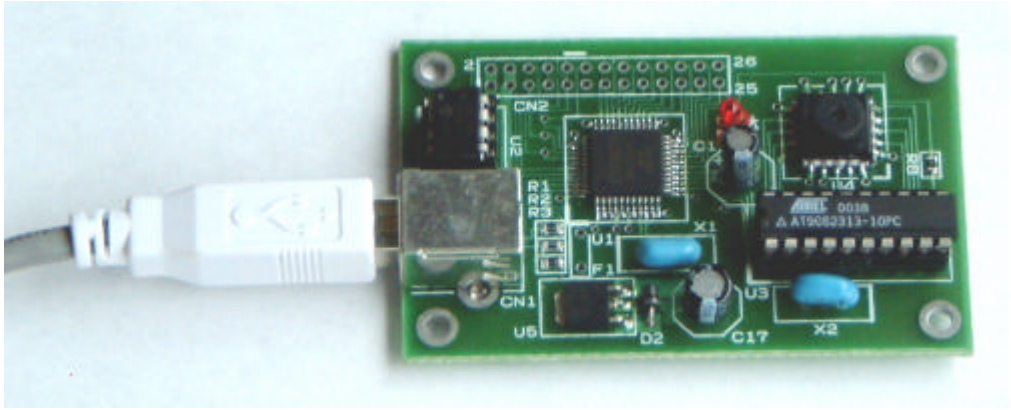
- モジュール・コンパイル手段の変更
  - 簡単では無くなった
- sysfsの強要
  - procfs →sysfs
- パワーマネジメント対応プログラミング
- スケジューラが安定しない
- マルチプロセッサ対応サービスの不足
  - 2.6に限ったことではないが…

問題点の整理と課題

- マルチプロセッサの普及
  - プログラムの書き直しが必要
- レガシー資産の蓄積と新デバイス対応
  - デバイスドライバ開発の負担
- カーネル2.6
  - カーネルのマルチCPU対応と、ドライバの対応は別
  - ユーザ開発者の連係が必要

最新Linuxデバイスドライバ開発応用

- 現状と問題点
- 対応策
- デモンストレーション
- 今後の見通し



対応策

2.6のドライバを
書いた人はいますか？

- まずはコードを書くこと
  - カーネル2.6のドライバ・モジュール
- プログラミングのヒント
  - シリアル処理とパラレル処理
  - マルチプロセッサ対応プログラミング
  - ユーザモードとカーネル・モード
- カーネル2.6機能の使いこなし
  - sysfs / kobject / libsysfs / udev
  - IOスケジューラ

カーネル2.6のモジュール

- モジュール・フォーマットの変更
  - *.o → *.ko
  - コンパイル・リンク方法の変更
  - ロード方法の変更（新しいmodule_init_tools）
- なぜ変更されたか
  - 実装上の問題（implementation problems）
  - 初期化の問題（initialization problems）
  - 削除に関する問題（removal problems）

カーネル外モジュール用Makefile(1)

```
#
TARGET:= hello.ko
all: ${TARGET}
hello.ko: hello.c
	make -C /usr/src/linux-`uname -r` M=`pwd` V=1 modules
clean:
	make -C /usr/src/linux-`uname -r` M=`pwd` V=1 clean
obj-m:= hello.o

clean-files := *.o *.ko *.mod.[co] *~
```

カーネル・ツリー

KBUILD_VERBOSE値
デフォルトは '0'
(サイレント・モード)

make オプション

カーネル外モジュール用Makefile(2)

```
#
TARGET:= hello.ko
all: ${TARGET}
hello.ko: hello1.c  hello2.c
	make -C /usr/src/linux-`uname -r` M=`pwd` V=1 modules
clean:
	make -C /usr/src/linux-`uname -r` M=`pwd` V=1 clean
obj-m:= hello.o
hello-objs := hello1.o hello2.o
clean-files := *.o *.ko *.mod.[co] *~
```

参照: Documentation/kbuild/*.txt

hello.koを構成する複数のオブジェクト・モジュール

対応策

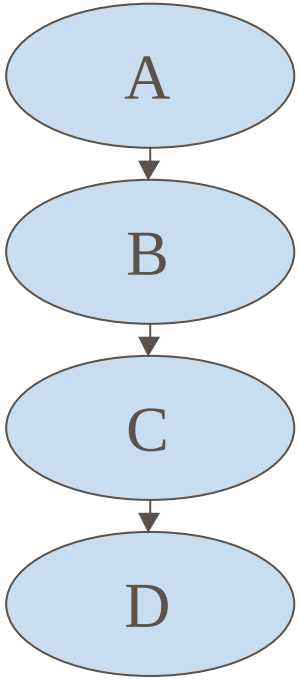
- まずはコードを書くこと
  - カーネル2.6のドライバ・モジュール
- プログラミングのヒント
  - シリアル処理とパラレル処理
  - マルチプロセッサ対応プログラミング
  - ユーザモードとカーネル・モード
- カーネル2.6機能の使いこなし
  - sysfs / kobject / libsysfs / udev
  - IOスケジューラ

プログラミングのヒント

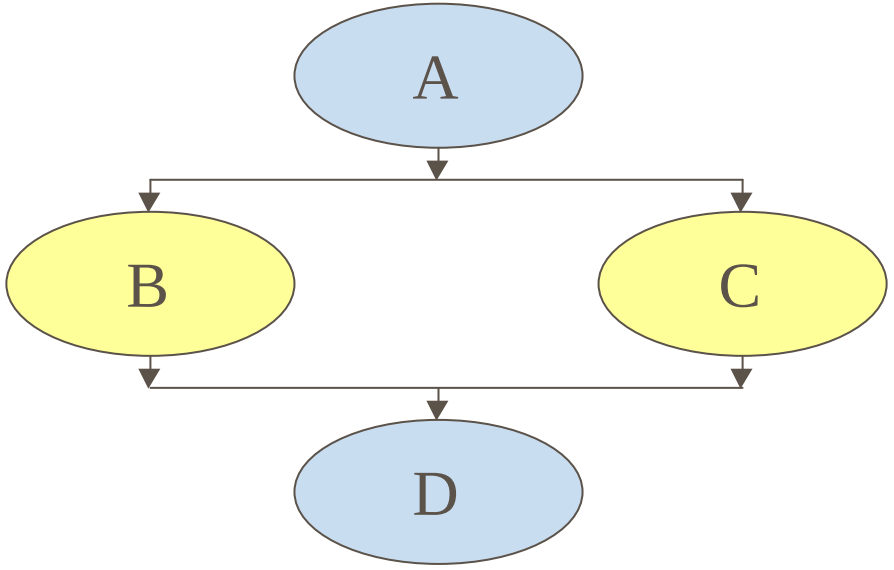
- シリアル処理からパラレル処理へ
  - パラレル処理とシリアライズ、操作の遅延
  - taskletとtask_queue
- 排他制御
  - SpinLockとAtomic操作
  - マルチプロセッサ(SMP)とHyperThreading(HT)
- カーネル・スレッド
- ユーザモードとカーネルモード

シリアル処理とパラレル処理

- マルチスレッド化→パラレル化



従来のモジュール



マルチスレッド化

スケジュールと遅延実行

- なぜスケジュール、遅延実行するか
  - 優先順位に応じたグルーピングを行って、リソースをバランス良く共有する
- カーネル2.6のサポート
  - softirq
  - tasklet
  - workqueue(schedule_task)
    - 従来のtask_queueをカーネル2.5で実装し直した
  - カーネルスレッド

遅延実行: tasklet

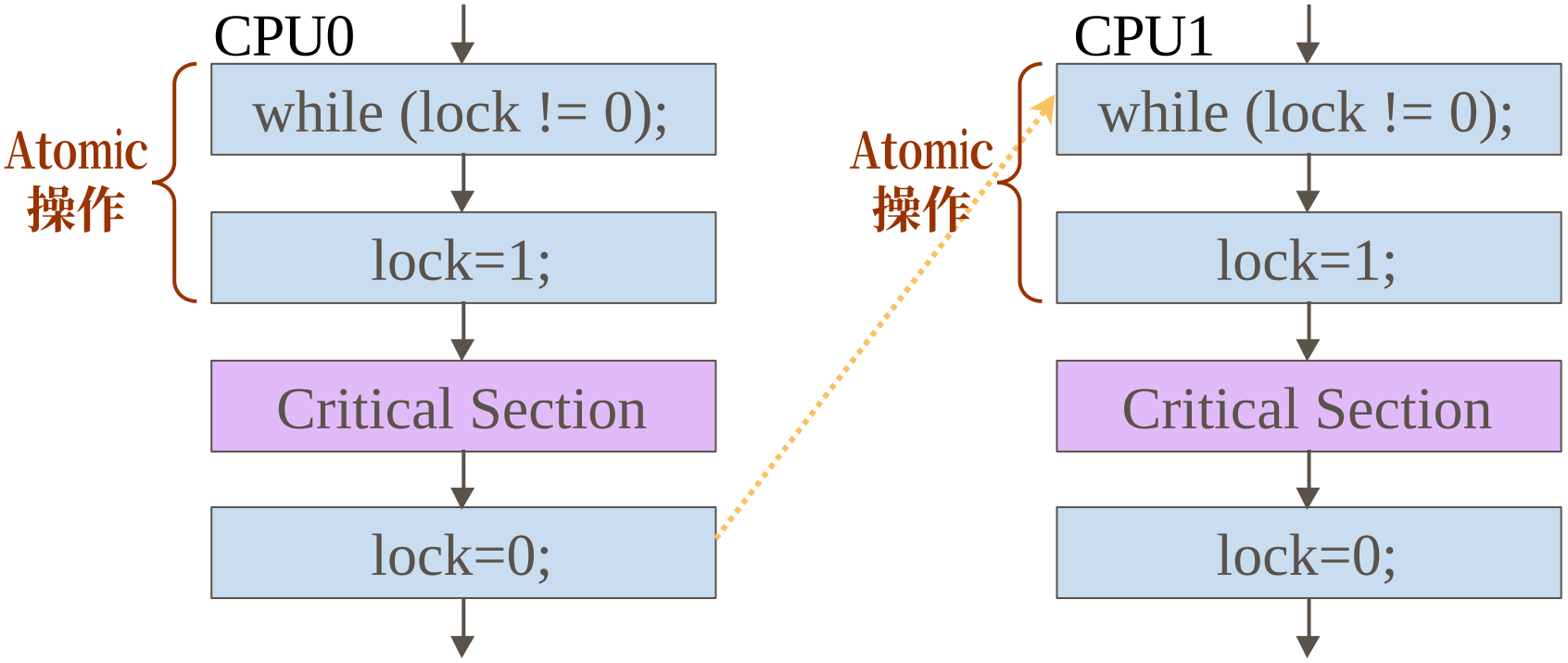
- softirq(ソフトウェア割込み)の汎用的な実装
  - 割り込みコンテキストの遅延実行
  - ソフトウェア割り込み(softirq) で実行
  - 制限事項：
    - リソースを待てない
    - スリープできない
    - ユーザ空間にアクセスできない
    - スケジューラを起動できない
    - セマフォが使えない(SpinLockかatomic操作を使う)

遅延実行: workqueue

- 汎用workqueue
  - システム全体でキューとカーネルスレッドを共有
  - ユーザエントリの関数の実行が遅れる可能性
- 専用workqueue
  - 専用のカーネルスレッドが割り当てられる
  - 自由にスケジュール可能
  - 特定モジュールが固有のworkqueueを利用
    - AIO, BlockIO, …
  - ドライバ・モジュールがGPLである必要

排他制御: SpinLock

- マルチプロセッサで機能する一種の「セマフォ」
- シングルプロセッサでは何もしない



SpinLock (2)

- SpinLockの実装パターン（模式的な概略）

カーネル2.4

```
while (lock != 0)  {
        ;


}
```

Intel HT推奨

```
while (lock != 0)  {
  asm("PAUSE");


}
```

HTスケジューラ

```
while (lock != 0)  {
  if (condition)
      schedule();
}
```

SpinLockとHyper Threading

- SpinLockは万全か？
  - 記述が簡単、後付けでコーディングできる
  - 取り合えず書いておいても弊害はない
  - 必ずループする
    - 常時ダーティなメモリアクセス（遅い：キャッシュされない）
    - Test and Set命令の頻発による負荷
- HyperThreadingの問題とスケジューラ
  - HT専用スケジューラ
  - monior/mwait 命令

排他制御: Atomic操作

- ループせずに排他制御する切り札
  - CPU間で同期を取るために使用できる事が保障されている(Lock命令)
- Atomic操作をセマフォとして使う
  - SMPとUPのコードの同一性
- SpinLockとの使い分け
  - 2.6のプリエンプティブ・カーネルの登場
  - SpinLockがプリエンプションを引き起こす

Atomic操作 (2)

- プログラム例

```
atomic_t  a;  /* atomic_set(&a, 1); */
if (atomic_read(&a) == 0)
        return BUSY;
if (atomic_dec_and_test(&a)) { /* a == 0 */
         critical();
        atomic_inc(&p->live)
} else {
        atomic_inc(&p->live)
        reurn BUSY;
}
```

Atomic操作で、

Atomic値を1減じてその結果の値が‘0’ならばTrue(1)を返す。

Linuxデバドラ本の解説は間違い

Atomic操作（3）

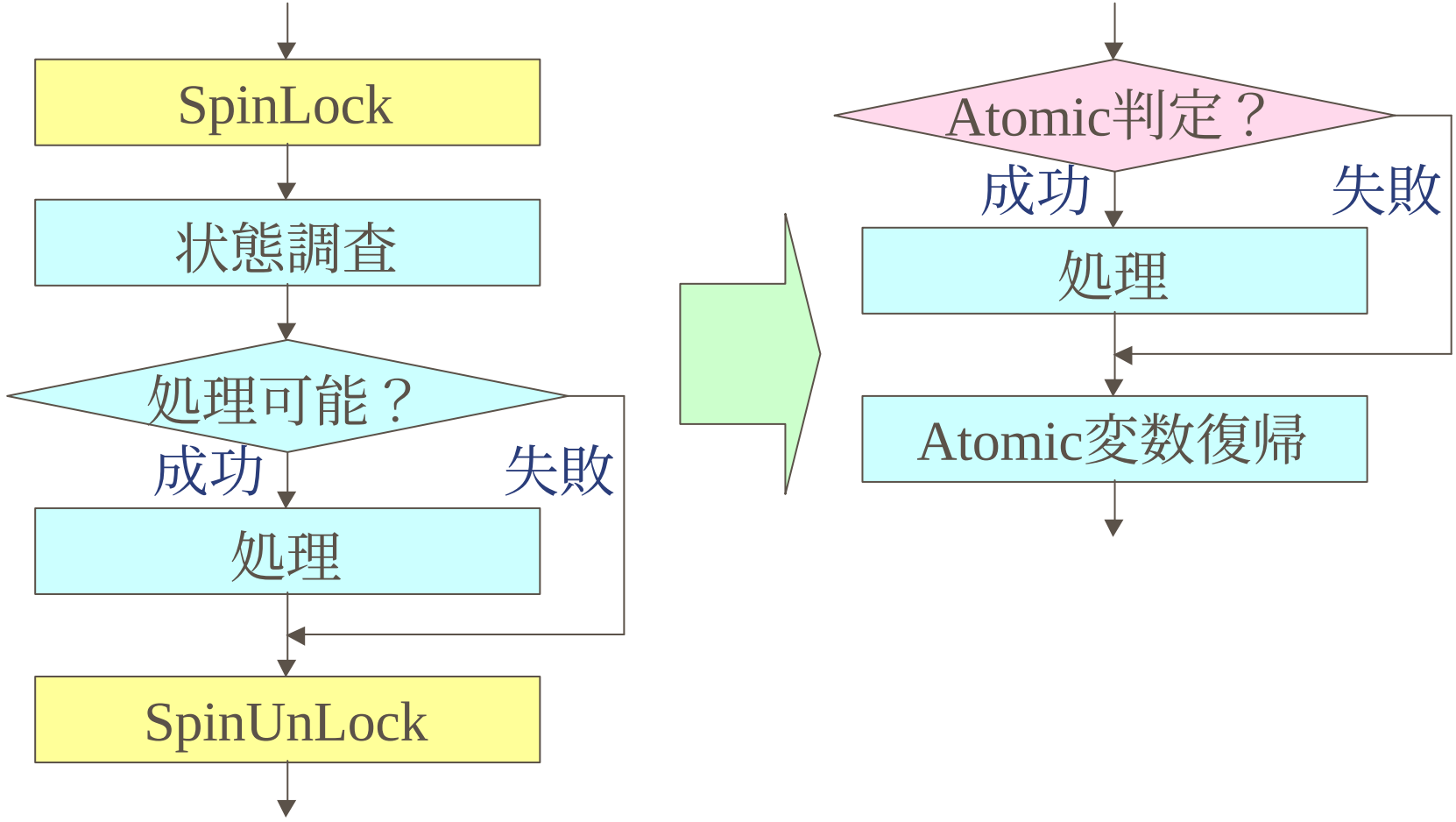
- Atomic操作関数

atomic_set	セット
atomic_read	読み出し
atomic_add	加算
atomic_sub	減算
atomic_inc	インクリメント
atomic_dec	デクリメント

atomic_inc_and_test	+1して検査
atomic_dec_and_test	-1して検査
atomic_add_and_test	加算して検査
atomic_sub_and_test	減算して検査
atomic_test_and_inc	用意されていない!
atomic_test_and_dec	
test_and_set_bit	ビットを立てる
test_and_clear_bit	ビットクリア

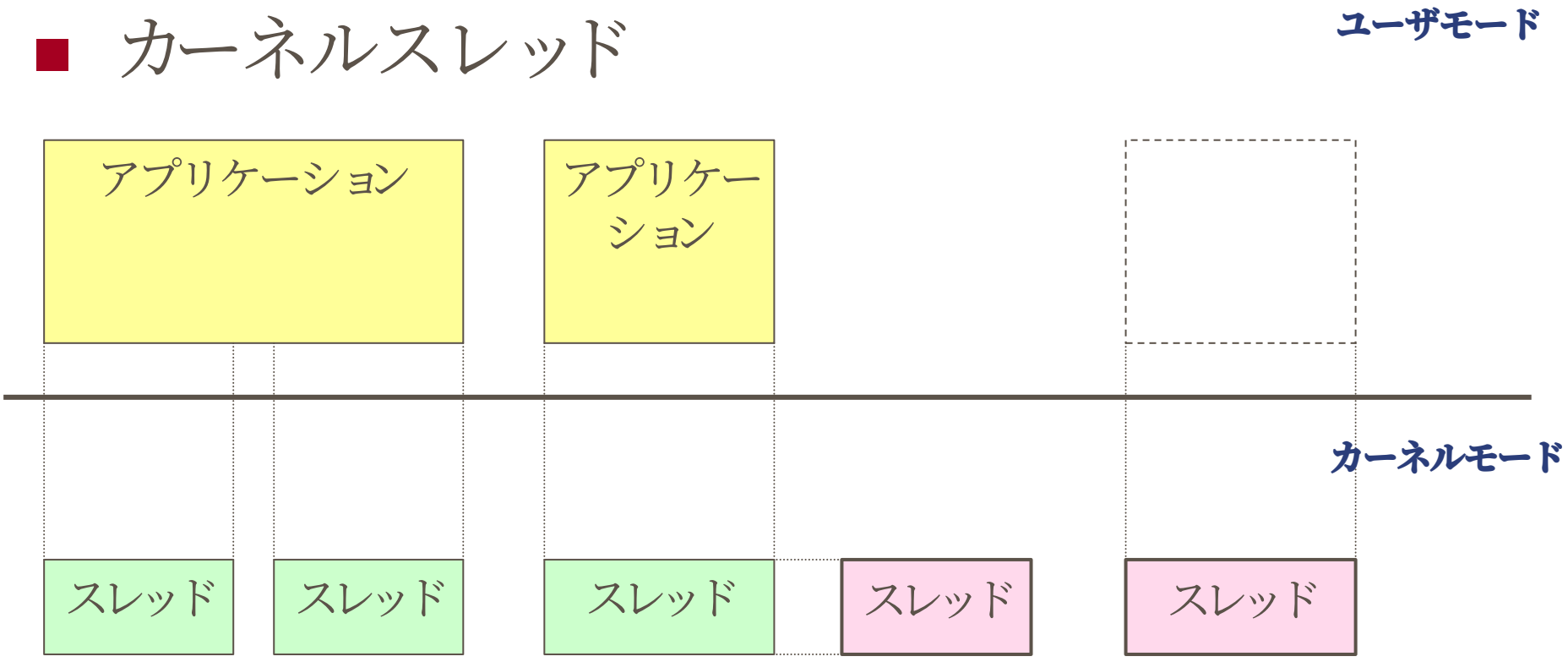
SpinLockとAtomic操作

- 典型的な処理の例



カーネルスレッド

- カーネルスレッド



スレッドは、スケジューリングの単位
プロセッサ割り当ての単位

カーネルスレッド(2)

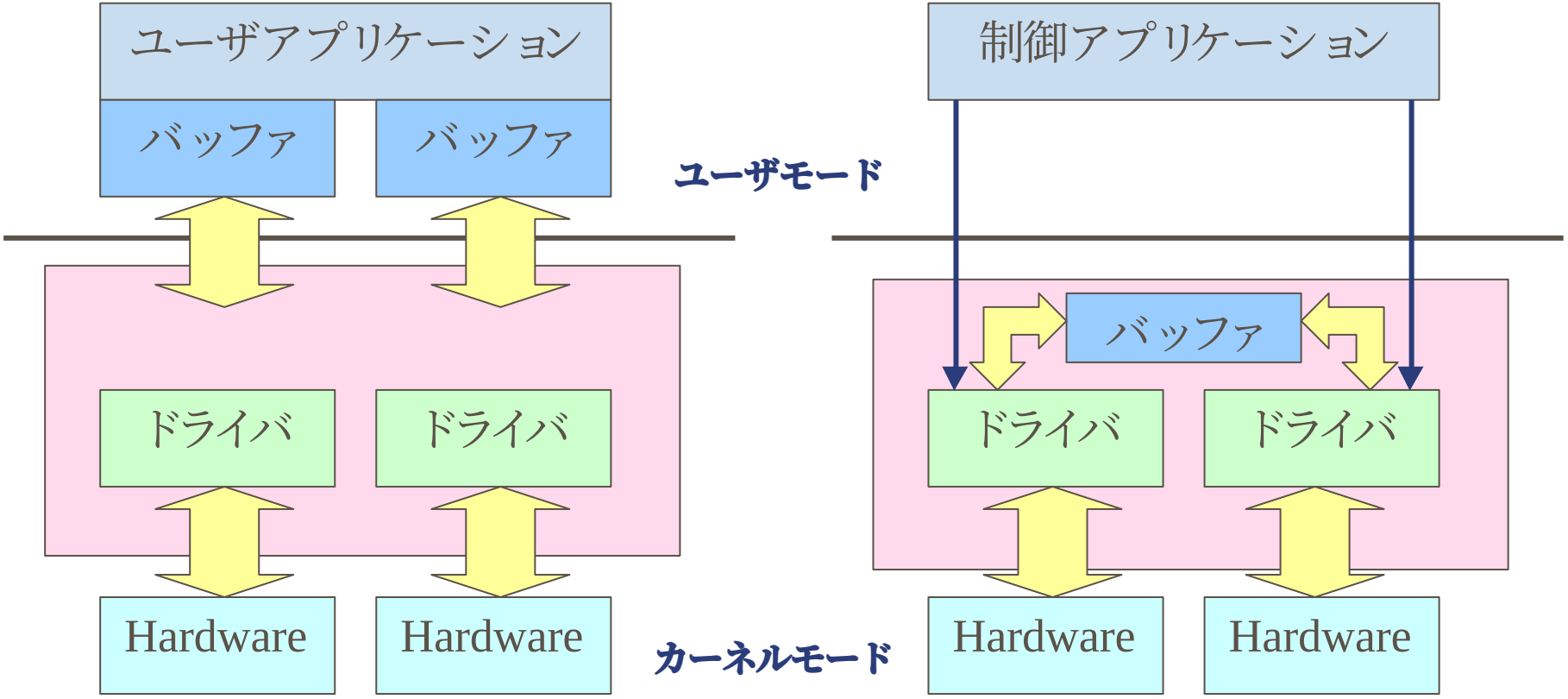
```
  PID TTY      STAT   TIME COMMAND
    1 ?        S      0:04 init [3]
    2 ?        SW     0:00 [keventd]
    3 ?        SW     0:00 [kapmd]
    4 ?        SWN    0:00 [ksoftirqd_CPU0]
    5 ?        SW     0:48 [kswapd]
    6 ?        SW     0:32 [bdflush]
    7 ?        SW     0:00 [kupdated]
    8 ?        SWN    0:10 [mdrecoveryd]
   13 ?        SW<    0:04 [raid1d]
   14 ?        SWN    0:11 [raid1syncd]
   15 ?        SW<    0:00 [raid1d]
  935 tty1     S      0:00 -bash
 2540 ?        SW     0:00 [kth_sleep]
```

ps ax の表示例

テスト用に作成した
カーネルスレッド

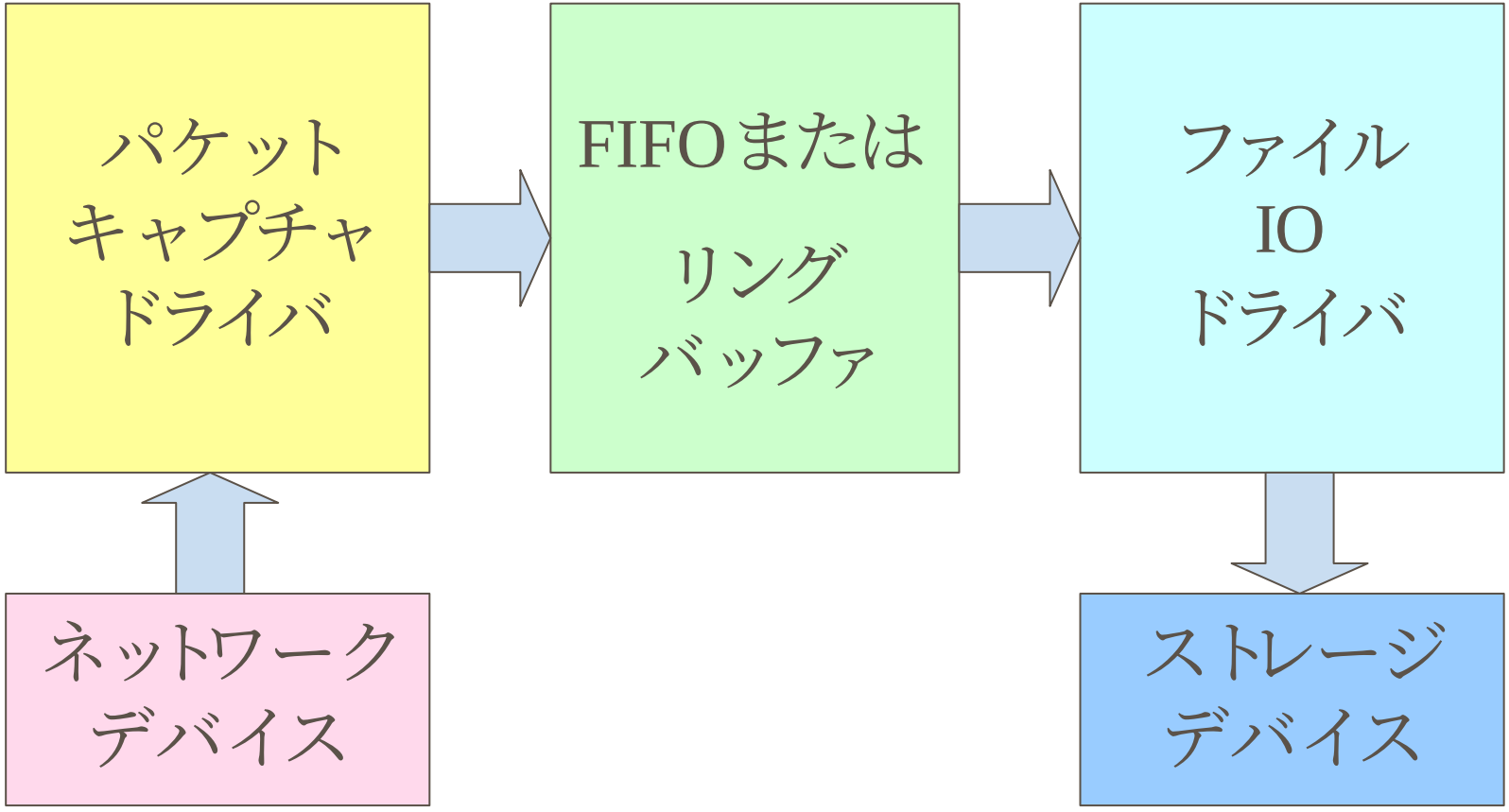
ユーザモードとカーネルモード

- IO処理の高速化



カーネルモード処理の例

- ネットワークパケット処理の例



対応策

- まずはコードを書くこと
  - カーネル2.6のドライバ・モジュール
- プログラミングのヒント
  - シリアル処理とパラレル処理
  - マルチプロセッサ対応プログラミング
  - ユーザモードとカーネル・モード
- カーネル2.6機能の使いこなし
  - sysfs / kobject / libsysfs / udev
  - IOスケジューラ

sysfs / kobject / udev / libsysfs

- sysfs
  - システム、デバイス、バス、ドライバを管理する新しい仮想ファイルシステム
- kobject
  - sysfsの構成要素
- udev
  - 従来のdevfsを置き換える「仮想デバイスノード・ファイル・システム」
- libsysfs
  - sysfsにアクセスするライブラリ（udevで使用）

IO スケジューラ

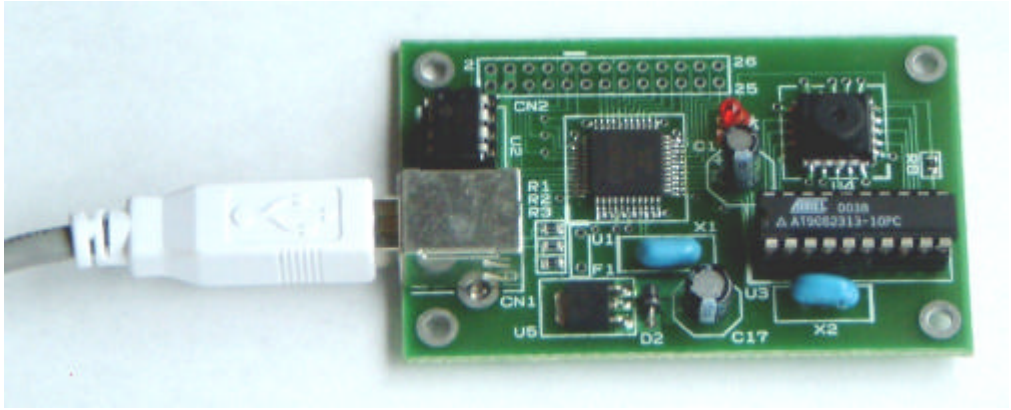
- デバイスIOのスケジューリング
  - elevator = as(anticipatory) 予測スケジューラ (デフォルト)
  - elevator = deadline　IO待ちを最小限にする
    - ファイルIO、データベース処理
  - elevator = cfq (Complete Fair Queuing disk I/O scheduler)
    - マルチメディア用、レイテンシの押さえ込み
  - elevator = noop
- IOスケジューラの動的な変更

まとめ :新時代のプログラミング

- 根本的には設計段階から考慮する
- パラレル処理とシリアル処理の使い分け
  - リソースを複数個持たせて同時実行
  - beforeイメージ ベースのコーディング
- SpinLock→Atomic操作の使い分け
  - ループするだけでは勿体無い…
  - lockプレフィックス :asm()でオリジナル関数も…
- カーネルスレッドの活用
- カーネル内モジュールでの処理

最新Linuxデバイスドライバ開発応用

- 現状と問題点
- 対応策
- デモンストレーション
- 今後の見通し



デモンストレーション

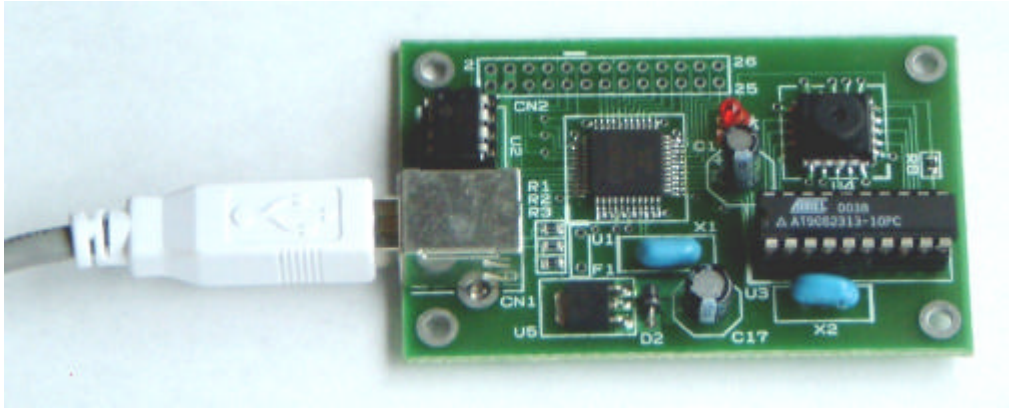
- ソースの公開

http://www.devdrv.co.jp/download/LKC/

ご自身の責任において
ご利用下さいますよう
お願い致します。

最新Linuxデバイスドライバ開発応用

- 現状と問題点
- 対応策
- デモンストレーション
- 今後の見通し



今後の見通し

- カーネル2.7は（当分）無い
  - Linusが2.7の検討よりも2.6の安定化を優先すると考えている
  - (おそらく)2.6は進化し続ける
  - （おそらく)2.4の進化は止まる
  - CMPが出ると世の中が変わる
- Hackする事！
  - とにかくテストしてみる
  - とにかくコードを書く

ありがとうございました。