Linuxによる制御と
デバイスドライバ開発

東北大学大学院 工学研究科

熊 谷 正 朗

kumagai@emura.mech.tohoku.ac.jp

Contents:

○背景

○Linux によるロボットの制御

- プロセスによるハードウェアのアクセス
- 制御のためのプロセス一定周期実行

○Linuxキャラクタ型デバイスドライバの開発

- モジュール
- ドライバ(file_operation メソッド)
- ドライバ作成に便利な機能
  ドライバによるハードウェアのアクセス
- 事例集

背景

ロボット研究者の憂鬱

昔：

- パソコンでハードウェア制御といったらMS-DOS
- MS-DOSでは資源はプログラム＝製作者の思いのまま
- とにかく動かすことは簡単だった

最近：

- MS-DOSではほとんど性能を生かせない高性能マシン
- OSの複雑化による自由度の減少

  ハード資源 ・動作タイミング
- それ以外は便利になったのだが....

背景

Linuxを選んだわけ：

- ハードウェアへのアクセスが簡単らしい
- RT-Linux なるものの噂をきいていた
- PC-UNIXとして，論文作成などでつかっていた

RT-Linuxにしなかったわけ：

- RT-Linuxに本能が拒否反応を示した
  - メモリをさわり放題＝Segmentation fault しない
    - ＝MS-DOS時代の暴走の恐怖の想い出
  - そうでなくとも，とばすたびにfsckはいやだ :-)
- 普通のLinuxで十分だと確信をもった
- なにかあったらC-cで止められるのは便利だ
- 手間いらず

背景

デバイスドライバを作り始めたわけ：

- 「物理メモリを操作するにはドライバが必須」という思いこみ
- 時間があるうちに将来に備えて技術開拓

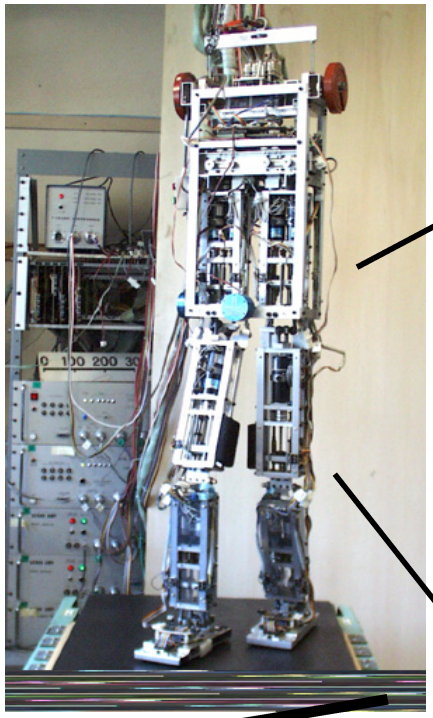
デバイスドライバをあえて作る利点：

- 「ハードに一番近いライブラリ」としての利用
- rootでは無くとも安全に使える
- 割り込みも使用可能
- ポーリングが簡単 ・確実
- selectで一括待機
- 複数ドライバの入出力形式統一による汎用化

Linuxによるロボットの制御

- ハードの操作
- 一定周期でのプロセス実行

制御対象



Master computer

Sensor slave
i486DX2-66MHz(2.5ms)

Servo slave
i386DX-20MHz(3.5ms)

Servo amp.

Shared memory

Control process
(Cycle: 10ms)

Kinematics calculating process

l.stage pattern generator

l.stage control process

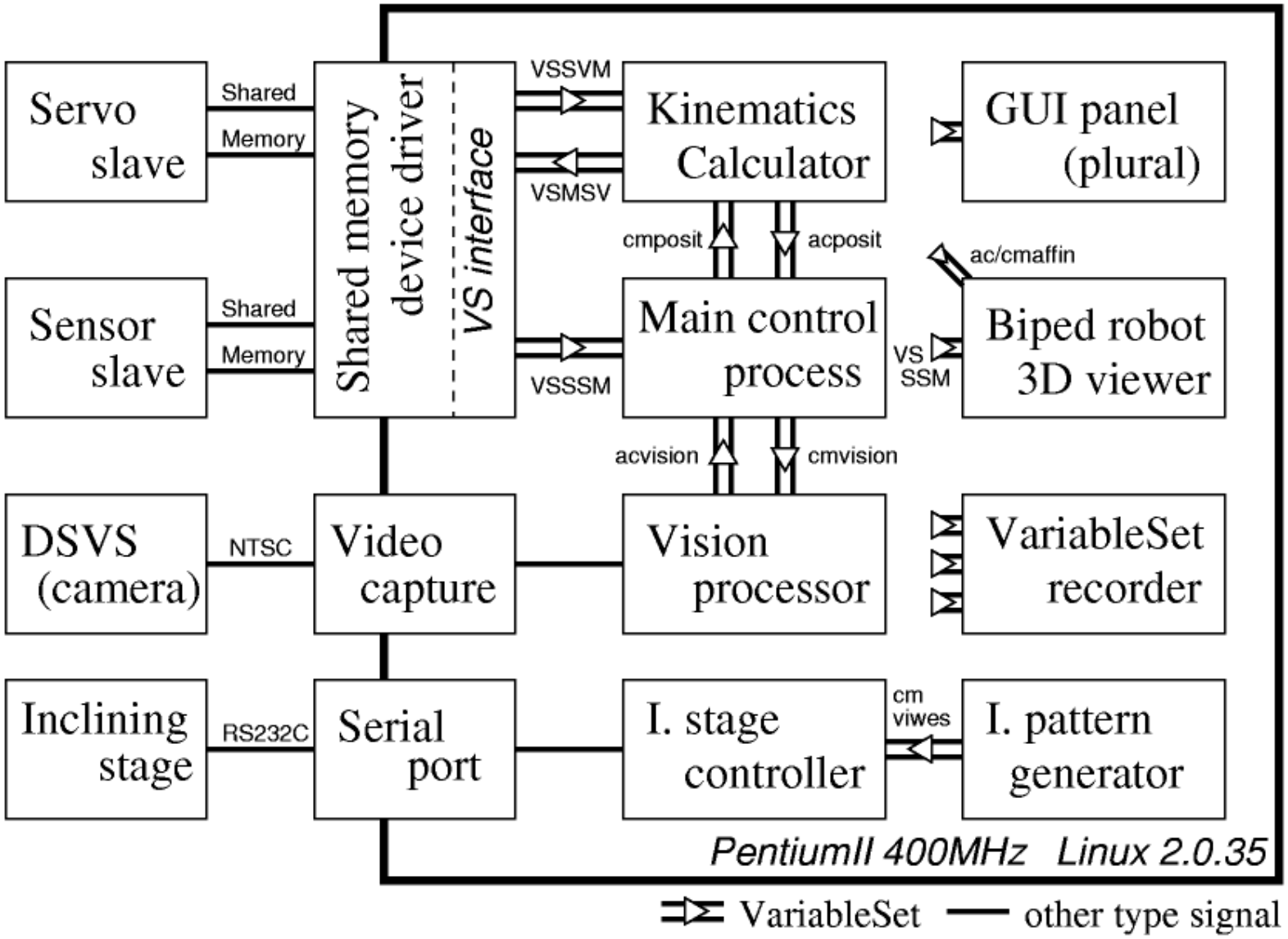
RS-232C

可変傾斜歩行実験台

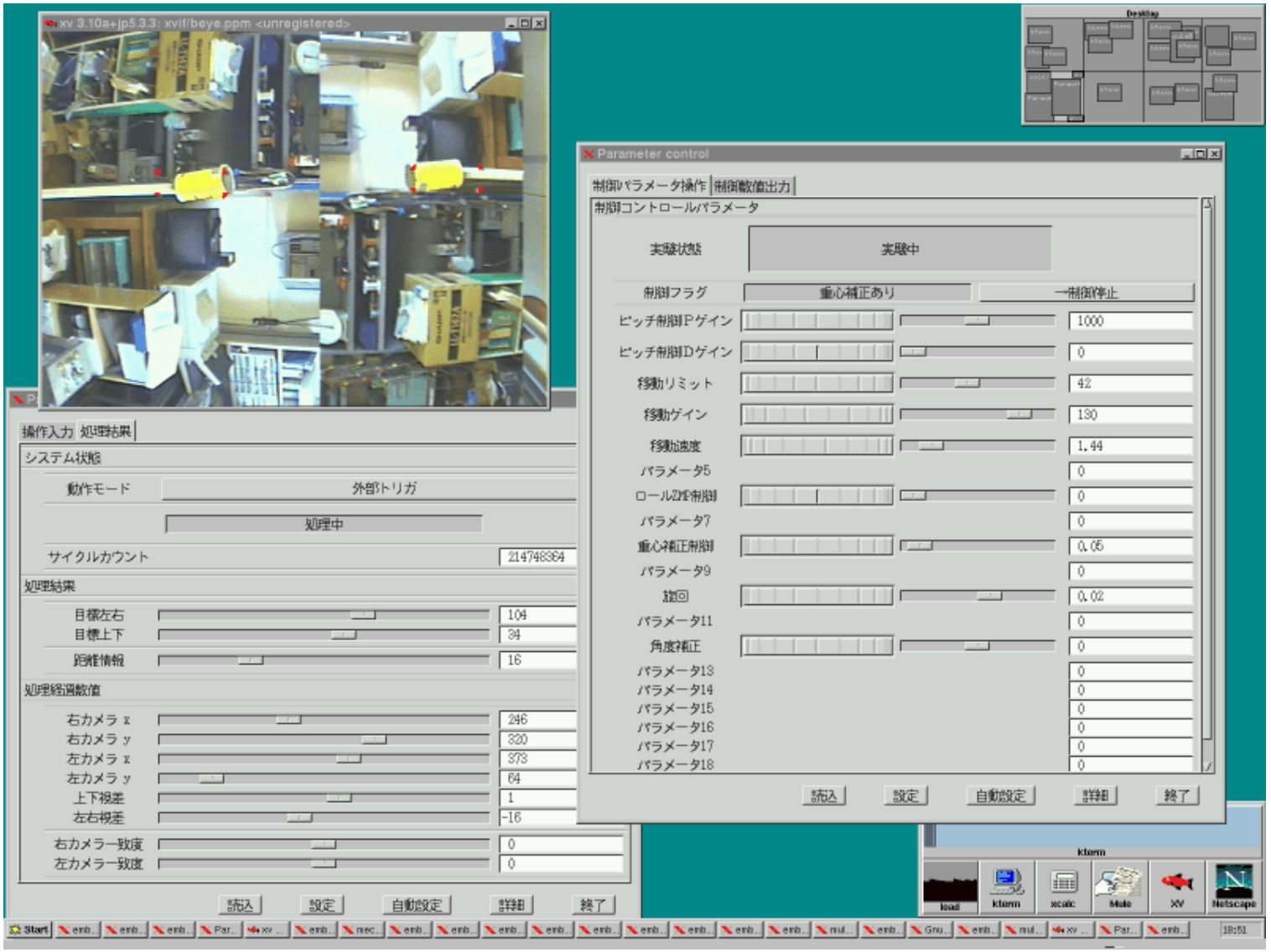
Built-in controller (V55)

Linux 2.0.35
PII-400MHz

2脚ロボット制御システム (マスタコンピュータ)



2脚ロボット制御システム　(マスタコンピュータ)



プロセスによるハード操作

I/Oポートの読み書き：

- root権限が必要 (or setuid)
- ioperm(), iopl() を用いてI/Oアクセス許可を得る
  - ioperm(): 0x0-0x3ff までの部分ごとに許可を得る
  - iopl(): I/O空間一括(含む割り込み許可)
- inb/inw/inl() によるポートの読み込み
- outb/outw/outl() によるポートへの書き込み
- /dev/port の読み書き: in/outがあれば不要？

note:
- ioperm(開始アドレス, 長さ, onoff);　onoff=1 で許可
- iopl(level);　level=3 でアクセス可
- inb(addr);
- outb(val,addr);　MS-DOSなどの表記と逆なところが要注意

プロセスによるハード操作

メモリの読み書き：

- root権限が必要 (or setuid) or パーミッション解放
- /dev/mem の読み書き open→lseek/read/write→close
- mmap によるマッピング

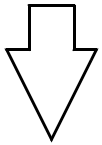
note:
mmap(0,長さ,prot,flag,ファイル記述子,オフセット);
prot: PROT_READで読み込み
flag: MAP_SHARED(他と共有可能)/MAP_PRIVATE(共有不可)
オフセット: PAGE_SIZE(0x1000)の倍数でなければならない

PCI情報の取得：

- /proc/pci の利用

制御のためのプロセス一定周期実行

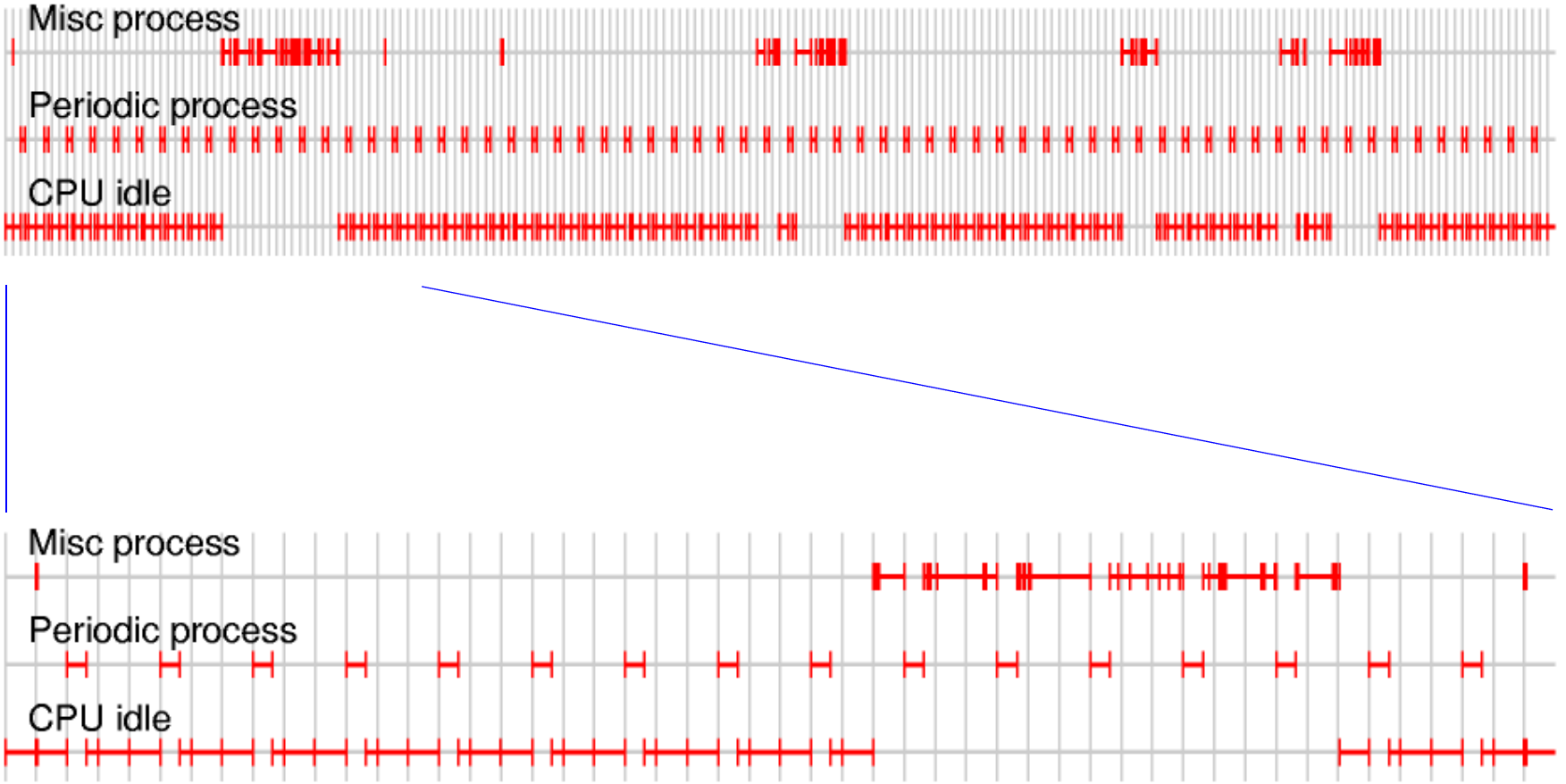
手法：
- 処理をループに入れ, 処理直前で*sleep*するプロセスを作成.
- 優先順位最大(nice --20)で実行.



非リアルタイムマルチタスクOSでありながら, 特定の処理を高精度に周期実行することが可能となる.

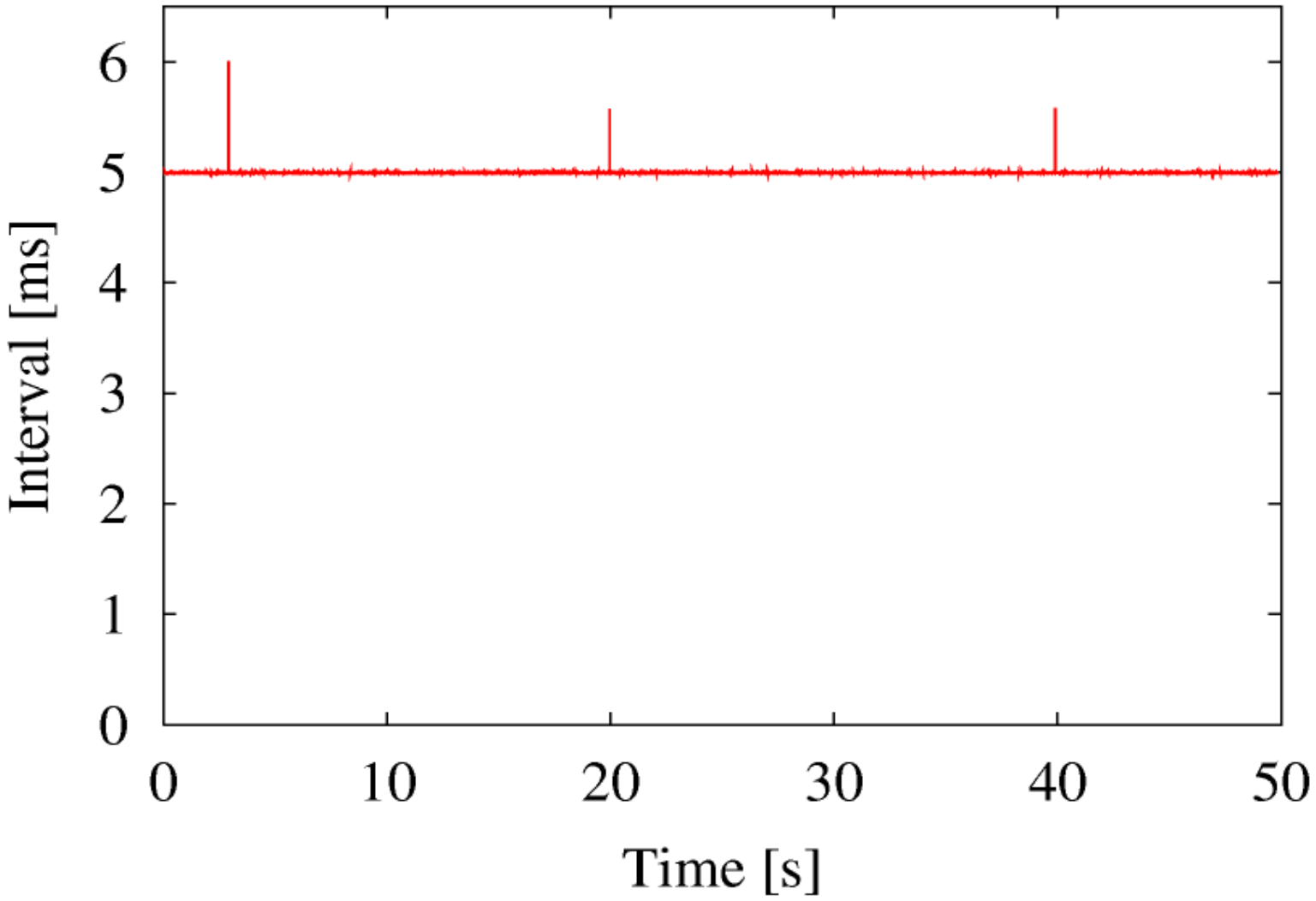
拡張：
- include/asm/param.hのHZの変更　→より短周期での実行可
- 複数プロセスの周期実行

周期実行の動作



実験環境　PentiumII 400MHz + Linux 2.0.35(Slackware 3.6)
　　タイマ割り込み周期を1msに変更
測定方法　スケジューラ監視専用ドライバ(自作)

周期実行の時間精度



Linuxデバイスドライバの開発

- モジュール
- ドライバの機能
- ドライバ作成に便利なカーネル機能
- 実装例

デバイスドライバの開発

構成要素：

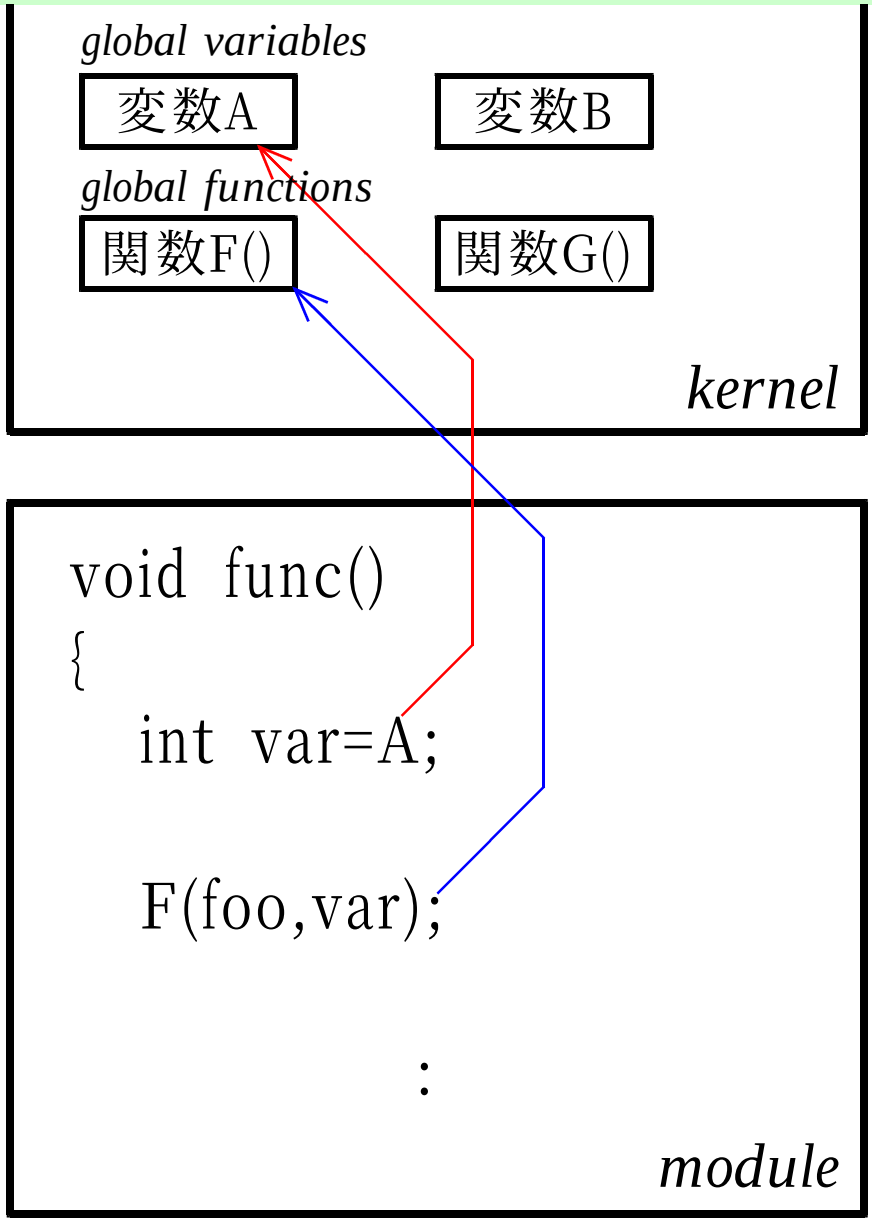
- ハードウェア操作部
  ハードウェア固有の部分
- ドライバ機能
  ユーザプロセスからの要求を解釈・実行する部分
- モジュール
  カーネルに組み込み，除去する処理部分

開発手順：

- ハードウェア操作部をプロセスレベルで可能な限り試験
- ドライバとしての部分を実装し，結合

モジュール

- カーネルに増築
- カーネル内のグローバルな変数・関数の利用が可能
- *insmod* で登録
  rmmod で除去
- 登録時にカーネル内グローバルシンボルの参照部分を処理
- 形式は *gcc -c* で生成されるオブジェクト



モジュールに必須な関数

int init_module(void)

- *insmod* 時に呼ばれる
- モジュール内の機能(ドライバなど)の登録
- 割り込みの設定など初期化作業
- *return 0;* で正常終了, 0以外で異常終了.

void cleanup_module(void)

- *rmmod* 時に呼ばれる
- 登録した機能の除去
- 要求した資源の解放

除去・解放を忘れるとハングアップの危険性高

モジュールの使い道

・デバイスドライバを作る
(後述)

・カーネル内の変数・関数を一時的に使用

init_module() で*0*以外を返すと*rmmod*の必要が無い

ドライバの概要

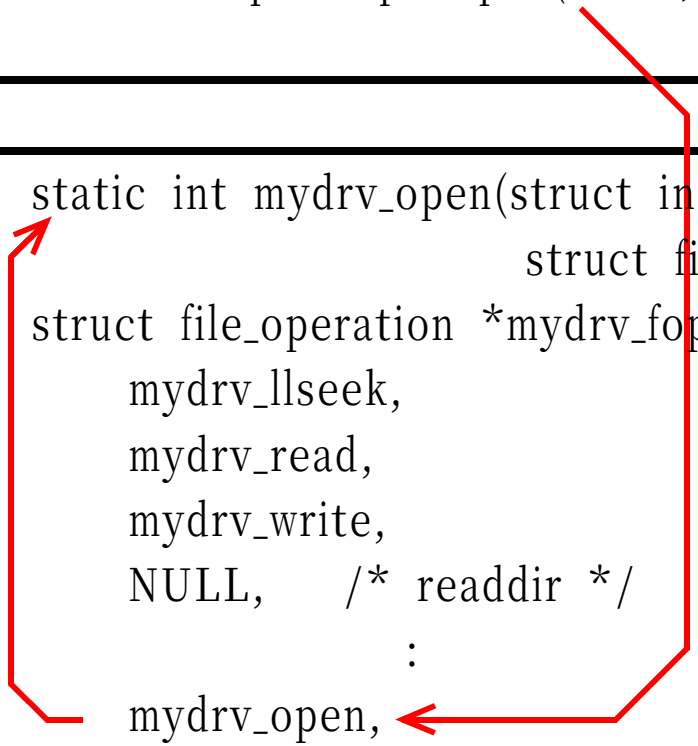
- ドライバは必ず各動作を担当する関数テーブルを持つ
- これをカーネルに登録する
- カーネルは各システムコールの処理過程でテーブルを参照, あれば呼び出し, 無ければ(*NULL*)デフォルトの動作をする

kernel

```
int ret = -ENODEV;
if (filp->f_op != NULL){
        ret = 0;
if (filp->f_op->open != NULL)
        ret=filp->f_op->open(inode,filp);
```

module

```
static int mydrv_open(struct inode*
                              struct file *);
struct file_operation *mydrv_fop={
        mydrv_llseek,
        mydrv_read,
        mydrv_write,
        NULL,     /* readdir */
                    :
        mydrv_open,
                    :
```



ドライバ機能の実装

◇カーネルの指定する形式で機能(メソッド)実装

- 各機能ごとに引数等形式が異なる
- Linux 2.0/2.1 間でも形式が異なる(半数程度)
- 一般にstaticで宣言

◇関数テーブルを作成

- 実装した機能のみ登録
- 実装しなかった機能についてはNULL

◇ドライバとして登録

- デバイスのメジャー番号，識別文字列，関数テーブル
  (アクセスはメジャー番号使用，番号選定注意)

ドライバ機能一覧

struct file_operation

- lseek　lseek(); システムコールを担当
- read　read();
- write　write();
- readdir　デバイスで未使用
- poll/select　select(); poll(); のデバイス確認
- ioctl　ioctl();
- mmap　mmap();
- open　open();
- release　close(); 等
- fsync　キャッシュ書き出し
- fasync　非同期通知
- check_media_change　ブロックデバイス用
- revalidate　ブロックデバイス用

ドライバ機能の役割

int open (struct inode *inode, struct file *file) 重要度 :高

☆L2.0: 同形式

- システムコール *open();* 内で呼ばれる
- 各種初期化動作
- マイナー番号 :*MINOR(inode->i_rdev)* 毎の処理
- *MOD_INC_USE_COUNT;* で参照カウント増加
- 返値0: 正常終了　負値: 絶対値が*errno*にセットされる

int release (struct inode *inode, struct file *file) 重要度 :高

☆L2.0: void release(struct inode *inode, struct file *file)

- システムコール *close();* 内などで呼ばれる
- 各種後始末
- *MOD_DEC_USE_COUNT;* で参照カウント減少

ドライバ機能の役割

ssize_t read (struct file *file, char *buf, size_t len, loff_t *pos)

☆L2.0: int read(struct inode *inode, struct file *file, char *buf, int len)

- システムコール *read(fd,buf,len);* 内で呼ばれる
- *buf,len* には*read();*の引数がそのまま渡される
- *pos* は *&file->f_pos* と同値 ：操作位置変数
- 返値はバイト数, *0*, 負値*(-errno)* を指定

ssize_t write (struct file *file, const char *buf, size_t len, loff_t *pos)

☆L2.0: int write(struct inode *inode, struct file *file, const char *buf, int len)

- システムコール *write(fd,buf,len);* 内で呼ばれる
- *buf,len* には*write();*の引数がそのまま渡される

ドライバ機能の役割

loff_t lseek (struct file *file, loff_t offset, int whence)

☆L2.0 int lseek(struct inode *inode, struct file *file, off_t, int);

- システムコール *lseek(fd,offset,whence);* 内で呼ばれる
- 未定義の場合, *file->f_pos* を適切に操作してくれる

int ioctl (struct inode *,struct file *, unsigned int request, unsigned long arg)

☆L2.0: 同形式

- システムコール *ioctl(fd, request, arg);* 内で呼ばれる
- 自由に定義して良い.
- 普通は *switch(request)* して, 場合に応じて *arg* をキャスト
- 本当に, なにやってもよい :-)
- ある意味, *read/write* より使いやすいかも

ドライバ機能の役割

unsigned int poll (struct file *, struct poll_table_struct *wait)

☆L2.0 int select(struct inode *, struct file *, int flag, select_table *wait)

- システムコール *select(); poll();* 内で呼ばれる
- Linux 2.0/2.1 で名前・機能が変わった(目的は同一)
- 準備ができていれば, なにもせず *return*
  できていないときは休眠情報を登録の上, *return*

poll (Linux 2.1以降)

- 返値は準備ができている条件を *OR* する
- 休眠情報設定に *poll_wait(struct wait_queue *, wait);*

select (Linux 2.0)

- 返値は*flag*で指定された条件が準備完了なら *1*
  (*select* の待機条件毎に別々に呼ばれる)
- 休眠情報設定に *select_wait(struct wait_queue *, wait);*

カーネル空間とユーザ空間の情報伝達

・メモリが仮想化されているため, ユーザ空間でのアドレス(ポインタ)はカーネル空間では直接利用することはできない

領域コピー型

copy_to_user (user, kernel, size); *memcpy_tofs(2.0)*

カーネル空間のデータをユーザ空間にコピー

copy_from_user (kernel, user, size); *memcpy_fromfs(2.0)*

ユーザ空間のデータをカーネル空間にコピー

数値単位

*get_user(val, *user);* *2.0: val=get_user(*user);*

ユーザ空間から1,2,4バイト取得(マクロ)

*put_user(val, *user);*

ユーザ空間へ1,2,4バイト押し込み

休眠のすすめ

・なにかを待つときは原則として休眠(含 時間指定休眠)
・禁止 *while(!condition()) ;*
・*poll_wait(select_wait)* は休眠待機する

休眠させる interruptible_sleep_on(struct wait_queue ** q);
起こす wake_up_interruptible(struct wait_queue ** q);

```
struct wait_queue *wq;
{   // 待つ側
    interruptible_sleep_on(&wq);
}
{   // 起こす側(割り込みetc)
    wake_up_interruptible(&wq);
}
```

時間情報の利用

・カーネルサイクル数 ： *jiffies*

タイマ割り込み 1回毎に 1ずつ増加

・実時間取得 ： *do_gettimeofday(struct timeval &tv);*

システムコール gettimeofday(); 内部関数

・Pentium命令 *RDTSC*

CPUサイクル1ごとに増加 (クロック数カウント)

・時間指定休眠

```
current->state = TASK_INTERRUPTIBLE;          // 寝た状態
schedule_timeout(wait);                       // Linux 2.1
current->timeout = jiffies + wait; schedule(); // Linux 2.2
```

タイマの利用

・指定時間後にある処理行いたい
・一定時間毎にある処理を行いたい(ポーリング)

```
static void cyclefunc(unsigned long data);
static struct timer_list cyclefunc_list =
                {NULL, NULL, 0, 0, cyclefunc};


{
   cyclefunc_list.expires = jiffies + cycle; // 再起動時間
   cyclefunc_list.data = data;
   add_timer (&cyclefunc_list);
}
```

・提供される機能は「指定時間後の関数呼び出し」
・カーネルのタイマ割り込み周期で指定
・不要時には必ず del_timer(&cyclefunc_list); で除去

ハードウェアへのアクセス@カーネル

I/Oポートの読み書き：

- *inb/inw/inl()* によるポートの読み込み
- *outb/outw/outl()* によるポートへの書き込み

メモリの読み書き：

- ブロックコピー

  memcpy_toio(phys, kernel, size); カーネルから物理メモリ

  memcpy_fromio(kernel, phys, size); 物理メモリからカーネル

- バイト・ワード・ダブルワードアクセス

  writeb[w,l](val, addr) カーネルから物理メモリ

  readb[w,l](addr) 物理メモリ参照

- 移植性は無いが動く方法

  Linux 2.0: ポインタに 物理アドレス を代入

  Linux 2.2: ポインタに (物理アドレス|PAGE_OFFSET) を代入

割り込みの利用

概念：

- 割り込みコントローラなどの処理はカーネルが行う
- 利用側は，関数をカーネルに登録する
  ＝関数を作るのみで割り込み利用可能
- 割り込みは共有可能

利用：

*void handler(int irq, void *dev_id, struct pt_regs *);*

int request_irq(irq, handler, flags, dev_name, dev_id);

void free_irq(irq, dev_id);

- *flags* には *SA_INTERRUPT*, *SA_SHIRQ* を | で指定
- *dev_id* は共有時に識別するためのなんらかのポインタ
- *handler*には割り込み番号*irq* と*dev_id*が渡される.
- １つの関数を複数登録することも可

デバッグ表示

printk

- *printf* と同一機能のカーネル版
  ただし, 浮動小数点機能なし(そもそも使えない)

printk の出力確認

- コンソール(のどこか)
- *dmesg* コマンド
- */var/log/syslog* (*syslog.conf* 次第)
- */proc/kmsg* (*syslog* と取り合うことあり)

おすすめ

- コンソール上で動作確認
- *kterm* で *root* で *nice --20 tail -f /proc/kmsg*

ドライバの構成例

(a) メモリランダムアクセス型

- *open()*
  マイナー番号による種類分け, 管理情報設定
- *release()*
  後処理
- *read() / write() / lseek()*
  メモリの読み書き, 操作位置(*file->f_pos*)の処理

- *ioctl()*
  必要なら動作設定など
- *mmap()*
  *mmap()*システムコールに必要なら対応

インターフェイス社製メモリンクドライバ

ドライバの構成例

(b) ストリーム(FIFO)型

- *open()*
  マイナー番号による種類分け, 管理情報設定
- *release()*
  後処理
- *read() / write() / lseek()*
  ストリームの読み書き, バッファの処理
  必要ならブロック(準備できるまで待機)
- *select()*
  読み書きの準備待ち

- *ioctl()*
  必要なら動作設定など

プロセススケジューリングプロファイラ

ドライバの構成例

(c) 状態入出力型(I/Oポートなど)

- *open()*
  マイナー番号による種類分け, 管理情報設定
- *release()*
  後処理
- *ioctl()*
  多品種少量アクセスの実装に有効
- *select()*
  ハードの状態待ち(ポーリングなど)
  割り込み待機
- *read() / write()*
  大きめのデータ領域の読み書き
  *f_pos*を使用せず、毎回先頭から読み書き

アドテックシステムサイエンス社製PIOボードドライバ

ドライバの構成例

(d) 伝言板

- *open()*
  マイナー番号による種類分け, 管理情報設定
- *release()*
  後処理
- *read() / write()*
  伝言領域への書き込み （ *file->f_pos*は使用せず ）
- *select()*
  伝言板の内容更新を読んでいる
  他のプロセスに通知

2脚ロボット制御プロセス間通信用ドライバ

開発中の失敗事例

とんだ × 多数

- 原因多数，これといった特定の対策なし

rmmod 後とんだ

- 登録物(ドライバ，割り込み，タイマ)の解放忘れ？

rmmod できなくなった → リブートするはめに

- 原因 ：モジュール参照カウントが0になってない
- 対策 ：未使用のマイナー番号を使って強制的に
  MOD_INC/DEC_USE_COUNT をする仕掛けをしておく

別のモジュールをinsmodしたらrmmodできなくなった

- 原因 ：モジュール内変数/関数が一部グローバル？
  ドライバソースを流用したら，insmod して混じった
- 対策 ：原則として module 内の変数・関数は static

参考文献

・Linux カーネルソース

linux/fs.h　linux/sched.h　などヘッダファイル

kernel/sched.c　fs/select.c　fs/poll.c　fs/open.c などカーネル中枢

drivers/char/mem.c　drivers/net/3c59x.c などドライバ

・「LINUXデバイスドライバ」

O'REILLY　ALESSANDRO RUBINI 著　山崎康宏・山崎邦子共訳

「プログラミング言語C」にあたるバイブルです

・「LINUXリアルタイム制御/計測開発ガイドブック」

秀和システム　船木陸議・羅正華共著

*RT-Linux*の本ですがそれ以外にもためになります

関連情報

・http://www.mechatronics.mech.tohoku.ac.jp/~kumagai/linux/
Linuxでロボット・ハード・制御
非RT-Linuxによるロボット制御
制御屋さんのための Linux デバイスドライバ製作入門
今回の講演のために最後の春休みをつぶして(...。)
リニューアルした情報ページ
本講演でほとんどふれなかった実践面を提供します
多数の2.0/2.2共通サンプルソースプログラムを掲載

・http:// 同上 /~kumagai/research/library/drivers/drivers.html
旧版
サポート範囲は大きく変わりませんが，実在ドライバ
１本を対象に上から下まで解説形式