ロボティクス・メカトロニクス講演会'00 チュートリアル

Linux v.s. RT Linux v.s. ART-Linux

汎用 Linux によるロボットの制御

– RT-Linux も ART-Linux も要らない！ –

– 普通の Linuxで 手軽に ここまで できます –

東北大学大学院 工学研究科

熊 谷 正 朗

kumagai@emura.mech.tohoku.ac.jp

第1章

概　　要

1.1　はじめに

Linux でロボットやハードウェアの制御が可能かという問いの答は Yes でもあり, No でもある. 当然ながら, 対象の規模や求める制御の性能に依存するためである. 他のリアルタイム OS に関しても同様である. 異なる点は, Yes と断言できる範囲の広さである. 特別な対策の施されていない素の Linux は, 他と比較すれば性能は低い. しかしながら, 歩行ロボットを動作させる程度であれば, 今時のパソコン＋ Linux でも十分な性能が得られる. おそらく, 各所で研究されているロボットの大部分は Linux で制御可能ではないだろうか. 本講演では, 実践面・性能面を主体に原理をふくめて, Linux を制御に用いる手法について解説する.

本手法では Linux をそのままつかう. そのため, “互換性がある” と宣言するまでもなく, 日常的に論文作成をするような環境がそのまま制御環境に切り替わる. 安定性も Linux そのままであり, 巷で販売されている各種パッケージや雑誌の付録をいれただけで制御が可能になる. 実用上はカーネルを 1 個所変更した方がよい部分があるが, それも, “0” を一つ足すだけで済む程度である. そのくらい手軽な方法である. 当然, 手軽さには代償がある. RT-Linux[(1)(2)], ART-Linux[(4)(5)] などに比較し, 時間的な厳密さの保証がない. それでも多くの事例には十分ではないか, と考えられる性能は発揮する. 本手法の特徴は “手軽に 安全に 十分な性能を” である. “万能・完璧” は求めてはいない.

本講演には “RT-Linux も ART-Linux も要らない！” などと物騒なサブタイトルがついているが, もちろん本当に要らないわけではない. 本業の研究を始める前に十分なシステム構築の時間が確保できるなら, ぜひともこれら Linux 系リアルタイム OS や, その他のリアルタイム OS[(6)] の導入を検討して頂きたい. システムが大規模化・固定化するのであればなおさらである. しかしながら, 準備時間が無い場合や, すこし動作を確認したいだけの場合, これまで MS-DOS で十分であったという場合は, これから述べる手法は速効性・手軽さという点で強力である. メカトロ屋の本業はあくまでハードを動かすことであり, 不要な手間は最大限避けたいとお考えの方には, 本手法をお試し頂きたい.

1.2　本講演の対象

本講演で提案する手法は以下のような方々・場合に適していると思われる.

- これまで Linux(UNIX) は使用してきたが, 制御に使うことに興味をお持ちの方
- これまで MS-DOS で倒立振子などを制御してきた方
- リアルタイム OS は, 導入や運用が面倒だとお思いの方
- 制御の担当者のプログラミングスキルがまちまちであるにも関わらず, 共同利用する場合 (巻き添えを食いたくない・食わせたくない方)
- コンピュータの環境構築や習得に手間をかけたくない場合

1.3　ロボットの制御

単にロボット (ハードウェア) の制御と言った場合, その対象は非常に広範囲にわたる. モータ一つの回転角制御から, モータ数十個のヒューマノイドロボットやさらに多数のロボットからなる分散システムまで, 規模は様々である. しかしながら, 研究室における実際の制御対象の多くは小規模なものであり, 一般的には以下のような機能が制御システムに要求されると言える.

- コンピュータに各種情報を取り込み, 演算し, 出力できること
- 一定周期で制御則を実行可能であること

高度な判断機能や画像処理などを併用する場合でも, もっとも核心となる部分は, 一定時間間隔で何らかの制御則に基づいて入出力を行っており, 高度な機能はその上位で, 比較的ゆっくりと行われることが多いのではないだろうか. これらの観点から, ロボットの制御を行うには, 大抵の場合は上の 2 つの動作が十分な能力で可能であれば良いと考えている. 本文書ではまず, Linux による一定周期の実行手法について解説し, 次に, Linux によるハードウェアの操作手法について述べる. それぞれ独立した内容であるため, 最終的に RT-Linux で制御する予定でもハードウェアの事前テストする場合や, ART-Linux を使用する場合にも第 3 章は参考になるかと期待している.

第2章

Linuxによる一定周期実行

RTLinux や ART-Linux が 普通の Linux と異なる点は, 処理の実行周期の高精度化 (期待通りのタイミングで実行) や, 優先度の厳密化 (処理の優先度をユーザが確定できる, 優先度固定), 優先度逆転への対応などの対策を有していることである. これらの機能は, 実行周期がクリティカルな制御や, 短い周期での制御, 複雑な大規模リアルタイム制御系を構築する際には, ユーザの期待通りの動作を保証してくれると期待される.

一方, Linux には, 当然これらの対策はとられていない (一部はあるが). しかしながら, このことは “制御に使えない” ことは意味しない. 実験してみると確認できるが, 多くの制御には十分使用可能な性能を有している. さすがに 1 [ms] を切るような短周期での制御 (例えばモータの電流フィードバックループ) には不適切であると考えられるが, 倒立振子や歩行ロボットの制御等には十分な性能であると考えている.

本章では, Linux のみで, いかなる手法を用いれば周期実行が達成可能か, そしてその周期精度がどの程度であるかについて述べる. なお, 本章では分かりやすくするため, 実験から理論を導くように話を進めるが, 実際にはカーネルのソースを解析して, 有効性を導いた手法である.

2.1 一定周期実行の実現

Linux そのものには, 一定周期で実行する, という便利な機能はない. 用意されているものは, ある時間プロセスを停止 (休眠) させる, という機能である. そこで, 単純に以下のようなプログラムを作成してみる (普通にコンパイル, 普通に実行). ただし, 制御に費やす時間は長くないものとする.

```
while(1)
  {
     DoControl();          // 制御等の実行
     usleep(15000);        // マイクロ秒単位の休眠
  }
```

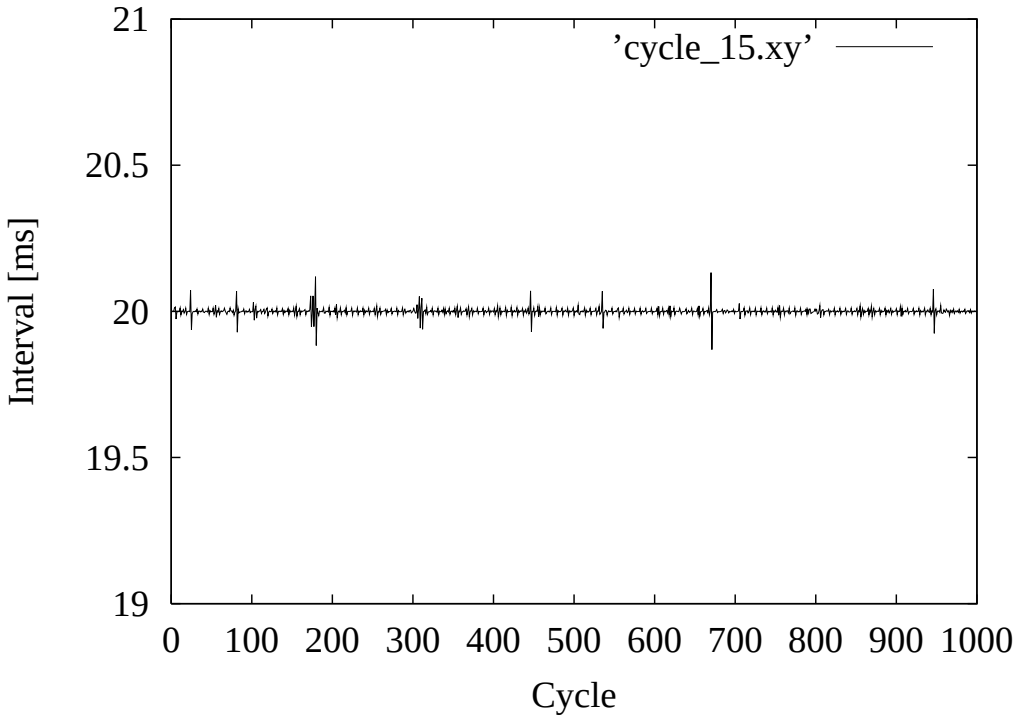

図 2.1　15 [ms] の休眠を指定して, 1000 回周期実行した周期例

ここでは固定時間の休眠としたが, 制御処理に要する時間が変動する場合は, 当然処理時間から休眠時間を概算することになる. このプログラムは一見, 動作しそうでもあるし, とても使い物にならないようにも見える. MS-DOS のような, CPU はすべてそのプログラムのもの, という環境下では制御則の実行時間が測定でき, 休眠時間を厳密に指定可能であれば, 問題なく動作する. しかしながら, Linux は複数のプログラム (プロセス) が CPU を分けあって動作している環境であり, 果たしてこのような単純な手法で想定通り動作するかは, はなはだ疑問である.

ところが, 実際にはこれが今回紹介する手法のすべてなのである. この手法による周期の実測例を **図 2.1**に示す. 指定した休眠時間とは異なるが, この例では 20 [ms] 付近の一定周期を刻んでいる. また, この例では見られないが, まれに大幅に周期が延びることがある. 休眠時間を変更して幾つかの条件を試すと

- 周期の最低は 20 [ms] である
- 周期は 10 [ms] 単位になる

ことに容易に気付く (Alpha 系の Linux の場合のみは 2,1 [ms] 程度のはずである). すなわち, ごく普通の Linux を使用したにも関わらず, 20 [ms] 以上, 10 [ms] 単位の周期であれば, 簡単に周期的な実行が可能であるといえる.

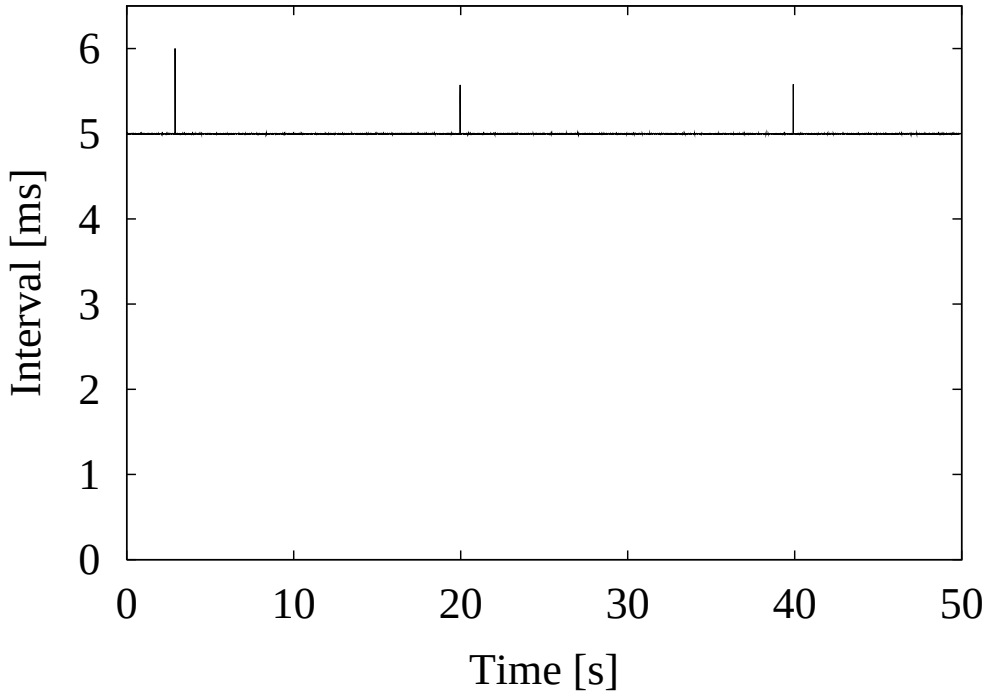

図 2.2　5 [ms] を目標として 10000 周期 実行した例

最低 20 [ms] で 10 [ms] 単位という制約は, 制御を行うには不便である. せめて, もう一桁細かな周期を指定したい. この目的のためには, 残念ながら, カーネルの変更と再構築が必要である. 変更すべき点は */usr/src/linux/include/asm/param.h* の定数 *HZ* である. さしあたって 100 を 1000 に変更する. このあとは, 各々の環境の指示にしたがってカーネル及びモジュールの再構築を行い, lilo 等に設定を行って再起動する. 一度でもカーネルを構築したことがあれば, ほとんど待つだけの作業である.

さて, この新しいカーネルで再起動した場合, 外見上は変化はみられない. 強いて言えば, *top* コマンドを実行したときに合計が 1000%になるなどである. この環境で同様に周期実行を試みる. 例として, 目標を 5 [ms] としたものを**図 2.2**に示す. 10000 回に数回の割合で遅延が確認されるものの, 全般に 5 [ms] で精度の良い周期が達成されている.

変更した数値は カーネルで時間関係の処理を行うために使用されているタイマ割り込みの頻度である (文字通り周波数). 標準では 100 [Hz] (=10 [ms]) であるが, これを 1[kHz](=1 [ms]) に変更した (機能自体にはなんら変更を加えていないが区別のため 1k-Linux と以下では記載). この変更によって, カーネルの処理量が増加するため, コンピュータ全体の性能は若干落ちるが, 制御に十分使用可能なものとなる. 基本的に周期のみの変更のため, 一部の設計の悪いドライバ (HZ=100 と仮定して実装) に支障が出る可能性も有り得るが, 他のカーネル用のパッチなどとの相性は非常に良いため, 多数販売・配布されている日本語ディストリビューションをそのまま使用可能である.

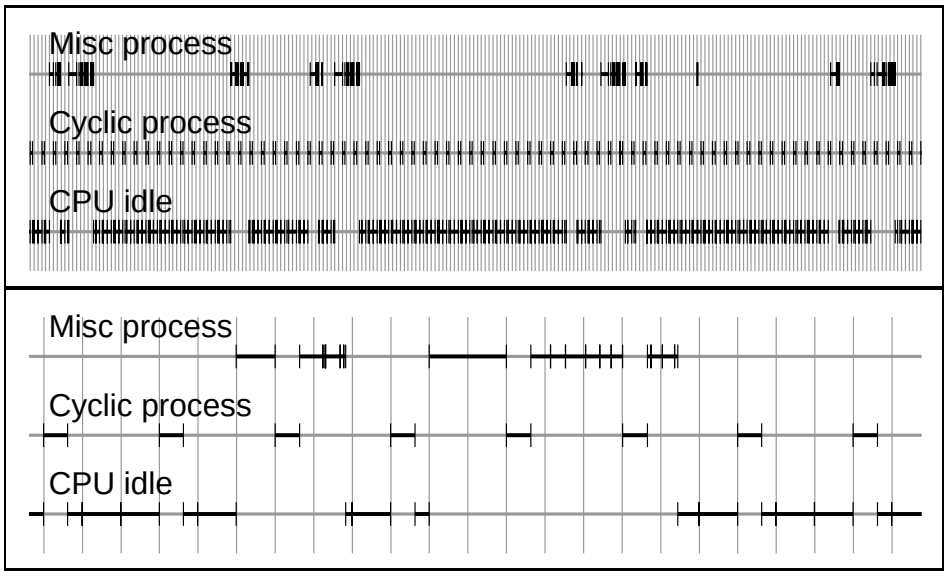

図 2.3　プロセスの実行タイミング (下図は上図の先頭部を拡大)

2.2　動作原理

実験的には動作が確認されたが, 実用上は原理的な裏付けが把握できることが望ましい. 本節では動作原理について触れる. 細かい原理は気にせず, 実践手法に興味のある方は本節は飛ばして次節に進んで頂きたい.

前節の結果より, 本手法にはカーネルのタイマ割り込みが大きく関わることが予想される. そこで, 周期実行プロセスが実際にいつ実行されているかを調べてみる. **図 2.3** に 1k-Linux において, 周期目標を 3 [ms] としてダミーの演算を行わせた例について示す (実測). 図は, ある時刻に CPU がどのプロセスを処理しているかを示すものである. 図において 横軸は時間軸であり, 'H' で表示された部分が CPU が処理している期間である (マルチタスクを実現するために, CPU は時間を区切って各プロセスを処理する). 縦淡線はタイマ割り込みのタイミングを示す. 下段は CPU が休止状態, 中段は目的の周期実行プロセス, 上段はその他のプロセスの処理である. 目的のプロセスはタイマ割り込み 3 回に 1 回 (=3 [ms]) 周期で, 常にタイマ割り込みの直後に実行されている. このため, 基本的にはタイマ割り込みの精度に近い精度で周期実行され, そのばらつきはカーネル内部の処理量の変化分である. 近年のコンピュータの性能をもってすれば, ほぼ一定周期で実行されることになる.

では, なぜ, タイマ割り込みの直後にプロセスが実行されるか, その原理について順次述べていく.

2.2.1 プロセスへの CPU 割り当て

Linux のようなマルチタスクの OS では, 細かい時間で区切って, 複数のプロセスを CPU で交互に実行し (CPU を与える), 一見複数のプロセスが同時に実行されているようにしている. このプロセスには主として 3 つの状態がある (他にもいくつかあり, 参考文献(2) に詳しい).

CPU で実行中 実際に, 該当プロセスが CPU で実行されている. この状態はさらに 2 つに分けられ, ユーザ空間で実行 (プログラムの大部分) される期間と, カーネル空間で実行 (システムコール等のカーネルでの処理) される期間である.

実行可能状態 プロセスが CPU の割り当てを要求している状態. カーネルはこの状態のプロセスをひとつ選んで CPU を割り当てる.

休眠状態 この状態にいる間はプロセスは CPU を要求しない. 外部からの情報到着や休眠時間経過などによって実行可能状態に戻る.

一般に Linux(UNIX) では非常に多くのプロセスが存在しているが (参考:$ps\ -aux$), その大部分は休眠状態でユーザのキー入力やネットワークからの接続などを待機している.

プロセスが 1 つだけ実行可能状態にある場合は, それに CPU を与えればよいが, 当然, 同時に複数のプロセスが実行可能状態にある場合が存在する. このとき, Linux カーネルは, ある条件にしたがって評価し, 1 プロセスを選択し, CPU を割り当てる. この動作をスケジューリングという. この結果, 現在実行中のプロセスより, 別のプロセスを実行するほうが妥当, と判断された場合, 現在実行中のプロセスは CPU を剥奪され, 実行が中断する (プリエンプション).

選択の条件としては, 各プロセスの「持ち時間」にあたる数値を比較する. 原則として持ち時間が長いプロセスに CPU が割り当てられる. この持ち時間は厳密に計算されるわけではなく, 実行中にタイマ割り込みをうけると, 1 減少する. 実行可能状態のプロセスが 2 つ存在すると, タイマ割り込みの度に交互に数値を減らされながら, 実行されていくことになる (厳密には性能向上のための細工あり). すべての実行可能状態のプロセス (すなわち休眠状態のものは除く) の持ち時間が 0 になったとき, この持ち時間の再計算が行われる. その際にはプロセスごとに設定されている優先度に基づいて計算され, この優先度 1 段階あたり 10 [ms] 分の時間がプロセスに与えられる (標準で下から 20 段目=200[ms]). この計算は同時に休眠状態のプロセスにも施される. ただし, 無制限に増加することを防ぐため, 再計算を繰り返した場合は上記基準の倍に収束するよう実装されている.

このようなルールの上で, 大部分を休眠し時々実行可能状態になるようなプロセスは, 実行可能状態になりさえすれば持ち時間を多く持つため, 直後のスケジューリングで CPU を割り当てられることが期待される.

2.2.2　タイマ割り込みと周期実行

Linuxはタイマ割り込みを使用して, 種々の時間に関する処理を行っている. 時刻を司る変数の処理の他, 復帰時刻を指定して休眠しているプロセスを調べ, 指定時刻になっていれば実行可能状態に遷移させる. また, 上述のように, 実行中プロセスの持ち時間の減少も行う.

前節で示した, 周期実行の手法は, 制御処理終了後に時間指定の休眠を行う. その結果, 指定時間経過後のタイマ割り込みの時点で実行可能状態になり, CPUの割り当てを待つ. 上述のスケジューリングの原理より, このプロセスは休眠が多いため, 持ち時間が多いことが期待されており, スケジューリングさえ行われれば, CPUを得ると想像がつく.

実際, タイマ割り込みの直後にはスケジューリングが行われる. スケジューリングが行われるのは, ドライバなどから意図的に要求される場合のほか, 割り込みやシステムコールの終了間際 (実行中プロセスがユーザ空間に帰る直前) である. これはタイマ割り込みにもあてはまる.

ゆえに “タイマ割り込み発生” → “周期実行プロセス実行可能状態に遷移” → “スケジューリング” → “周期実行プロセスCPU取得” という一連の動作がおこり, 比較的高精度にプロセスが周期実行されることになる.

ただし,

- タイマ割り込みがシステムコールの実行中に発生した場合, その処理が終了してからスケジューリングするため, 負荷の大きいシステムコールを実行していると遅延が生じる
- 同時に複数のプロセスが実行可能状態に遷移した場合に, 持ち時間で負けると, 多い方にCPUを取られてしまい, 周期が大幅に遅延する

という問題がある. これはLinuxをそのまま使う以上, 避けられない問題点であるが, 頻度からすると普通はそれほど多くなく, また, 後者の解決のために優先度を上げておく手法もある. これについては次節で述べる.

2.3　周期実行の実装技術

2.3.1　時刻の実測

周期はほぼ正確である, という仮定の元に制御プログラムを作成しても, 本手法ではほとんど問題は生じないと考えられる (特にフィードバック制御). しかし, より厳密さを求めたり, 制御周期を検証する場合には, 現在時刻を取得可能であることが望ましい.

また, 処理時間の変動する制御則の場合, 休眠時間を調整しなければならず, この場合にも演算時間の測定が必要である. 時刻の計測には大きく 2 つの方法がある.

第 1 の方法はシステムコール *gettimeofday()* を使用することである. これは現在時刻を, 秒単位およびマイクロ秒単位の 2 つの数値 (構造体) で与えてくれる. 精度的にはマイクロ秒近いとされ, コンピュータに依存しない値が得られるため使いやすい反面, システムコール呼出の分だけ時間を要する.

第 2 の方法は Pentium 以降の CPU に搭載されたタイムスタンプカウンタを利用することである. このカウンタは 64 ビットで, CPU リセット時にゼロにクリアされ, 以降, CPU のクロック毎にインクリメントされる. そのため, CPU クロックの精度を有する時刻数値として使用できる[8]. 実際, Linux のカーネルでも, 可能ならこれを利用するようになっている. 使用にはインラインアセンブルを必要とする. 具体的には

```
unsigned long long GetTick(void)
{
  unsigned int h,l;
  /* read Pentium time-stamp counter */
  __asm__(".byte 0x0f,0x31" : "=a" (l),"=d" (h));
  return ((unsigned long long int)h<<32)|l;
}
```

とする (命令をコードで直接埋め込み, *gcc −O* でコンパイル). *long long* 型は見慣れないかも知れないが, 64bit の整数型である (printf での型指定は ”%Ld”). 命令一つで済むため, 実行には時間を要せず精度も良いが, CPU のクロック数に依存するため, 注意が必要である. 実時間への変換係数はおおむね CPU のクロックの公称値に等しいが, わずかに異なる (とはいえ 1%も違いはない).

両者とも, 手軽に時間を得ることが可能であるが, 汎用性を考慮するなら前者, CPU が Pentium 以降で実行環境を固定するなら後者を推奨する.

2.3.2 複数処理の周期実行

制御を行う際, 時として複数の独立処理を周期的に行いたい場合がある. このような場合, 本手法では, “公約数をもつ周期を設定する” とそれぞれの周期精度が確保できる. 逆に互いに素な周期を設定すると一方が乱れる. これを示したのが **図 2.4**である. 上は 2 プロセスの周期が 3,6 [ms] で公約数 3 をもつ. 下は 4,5 [ms] で互いに素である. 前者では 2 プロセスが噛み合う形で独立に動作しているが, 後者では一方の周期が乱れる. これは原理からすれば当然のことであり, 同時に実行可能状態になる場合にはいずれかが待たされる (図ではプロセス (2)). 同時に複数のプロセスを周期実行させる場合には注意が必要である (このことは, 本手法に限らず一般論である).

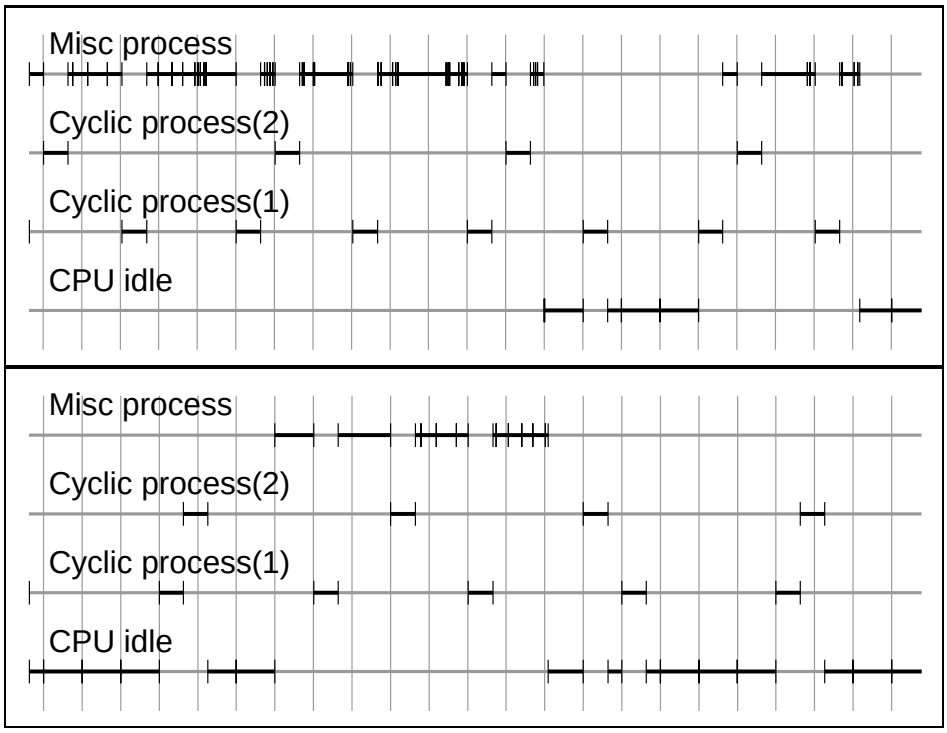

図 2.4　2プロセスの周期実行 (上 3,6 [ms] 周期　下 4,5 [ms] 周期)

もし, 一方が乱れてもかまわないが任意の周期で組み合わせたい, という場合は, より重要なプロセスに対して, 次に述べる手法で優先度を上げるなどの措置を講じる必要がある.

2.3.3　より確実な CPU 取得

原理のところで述べたように, 最終的に CPU がプロセスに割り当てられるかは持ち時間に依存する. 休眠期間が長いプロセスの持ち時間は一般に多いが, 目的のプロセスの他にも休眠プロセスが多いことを考えると, 他に CPU を取られる危険性も存在する. これを回避するためには, 初めから持ち時間を多くするなどの手法があり, 以下に述べる.

優先度の操作

Linux に限らず, UNIX 全般に *nice* というコマンド, *nice() setpriority()* というシステムコールが存在する. これらはプロセスの優先度を操作するためのものであり, ±20 の範囲で増減・設定する. この値が大きいほど “nice” であり, 20 を設定すると, 他に実行可能なプロセスがない状態でのみ動作するようなプロセスとなる. 逆に −20 を設定すると,

CPU を得やすくなる. 具体的に Linux では持ち時間再計算の度に 優先度 20 で最小の実行時間 (タイマ割り込み周期 1 回分) で, 優先度 1 あたり 10 [ms] の実行時間を得る (全実行可能プロセスの持ち時間 0 で再計算). たとえば, 優先度 −20(最優先) のプロセスと 0(標準) のプロセスを同時に連続実行させると, おおむね 400[ms](=10 [ms]×40):200[ms] の割合で CPU を使用することになる (最優先にしたからといって独占的に使用するわけではない).

休眠している時間の長いプロセスの持ち時間は, ここで計算される持ち時間の 2 倍に漸近するため, 優先度を上げておくことは CPU を確実に得るためには効果的である. ただし, いずれの設定手法も, 負の値は root 権限がなければ設定できないので注意が必要である.

スケジューリングポリシーの変更

ここまで述べてきたスケジューリングの方法は “平和な” スケジューリングである. 全プロセスが CPU を穏便に分けあって実行するため, システムとして身動きが取れなくなることはない. それに対して, Linux にはスケジューリングで必ず CPU を得る機能も用意されている. これは *sched_setscheduler()* によって設定するが, 優先度が通常のプロセスより常に上になる. その結果, もし誤ってこのようなプロセスを無限ループにしてしまった場合, X もシェルもすべて動作しなくなるため (これらも一介のプロセスに過ぎない), 停止することすら出来なくなる. そのため, 使用には注意が必要である (デュアルプロセサの場合は暴走プロセスが 1 つのみなら問題ない).

優先度の低いダミープロセスの実行

優先度の操作は周期実行プロセスそのものの優先度を高めることを念頭においている. これに加えて, ダミーのプロセスを実行することで持ち時間を増加させる方法もある. 優先度が最低 (nice 20) のプロセスは 1 回の持ち時間再計算あたり, 1 タイマ割り込み周期しか持ち時間を得られない. 逆に言えば, このプロセスのみが実行可能状態にあるような環境では, しばしば再計算が行われる. その結果, 休眠の多いプロセスの持ち時間は上限まで高くなる. 1 回の制御演算が 1 タイマ割り込み周期以内に終了する場合には効果はほとんどないが, 制御演算に時間を要し, 途中でタイマ割り込みを受け, 徐々に持ち時間が減少するような場合には, 頻繁な再計算の効果がある. 他に何もなければ, ダミーの処理を入れれば良いが, 画像処理などの CPU を連続して使用するような処理を併用する場合には, このプロセスに同様の効果を持たせることが可能である.

補足

それぞれの使用法を以下にまとめる.

```
------------------------------------------------------------------------
# nice --20 ./program
  ./program の優先度を 20 上げて実行. --20 は '-' + '-20' である
------------------------------------------------------------------------
nice(-20);  nice システムコール              <unistd.h>
setpriority(PRIO_PROCESS, getpid(), -20);   <sys/time.h><sys/resource.h>
  setpriority システムコールは汎用であり, プロセス単位ではこのように指定
------------------------------------------------------------------------
struct sched_param sp;
sp.sched_priority=99;    // 1(小)-99(大)
sched_setscheduler(0,SCHED_FIFO,&sp);       <sched.h>
------------------------------------------------------------------------
```

2.4　まとめ

本章では, ロボットなどを制御する際に必須と考えられる, 一定時間周期でのプログラムの実行の手法について述べた. 周期にわずかなばらつきがあるほか, 稀に周期に大きめの遅延が生じる. そのため, 実行周期に厳密さが要求されるような目的 (高度な制御理論の検証など) には不向きであると考えられる. しかし, 多くのフィードバック制御には十分使用に足ると考えている. 必要なら, 周期時間を計測して, パラメータを修正することも可能である.

最後に周期実行手法の要点をまとめておく.

- ループの中に制御演算と休眠を入れる.
- 優先度を上げて実行する.
- 20 [ms] より短い間隔で周期を設定したい場合は *linux/include/asm/param.h* の定数 *HZ* の値を変更してカーネルを再構築する.
- 制御演算処理は, なるべく 1 タイマ割り込み周期に収まるようにする. または, *HZ* を変更する際にはそうなるような値を選択する.

以上の手法により, 今日のパソコンをもってすれば, 十分な精度の周期実行が可能となる.

第3章

Linuxによるハードウェアの操作

本章ではLinuxの通常のプロセスからハードウェアを操作する手法について述べる. 基本的に, 一般向けのマルチタスクOSでは, ハードウェアに対する直接的なアクセスは困難である. これは特定のプロセスの身勝手によって, システムが利用しているハードウェアが操作された場合, OSとしての動作に支障をきたしかねないため制限されていることによる. また, メモリが仮想化されていることにより, 各プロセスが参照しているアドレス (ポインタ) がハードウェア上のアドレスと異なるため, 直接特定のメモリ領域にアクセスすることは不可能である.

Linux の場合, I/Oポートおよびメモリに関して, rootの権限は要するものの, 簡単な手順によってアクセスすることが可能である. 以下にI/Oポートおよびメモリについて述べ, 最後にプロセスで割り込みをある程度利用可能にする手法について述べる.

3.1 I/Oポートのアクセス (x86)

上述のように, 原則としてハードウェアへのアクセスは禁止されている. そのため, I/Oポートをアクセスする場合には, アクセス許可を得る必要がある. 一旦アクセス許可を得た後は, I/Oポートに関しては完全に自由に利用可能となる. なお, 本節は Intel 86系のCPUのみに適用される (カーネルソースを見る限り, そもそもI/Oポートというものが他アーキテクチャにないようである).

3.1.1 I/Oポートアクセスの許可

まず, 許可を得るには2つの関数 (システムコール) がある.

- *int ioperm(unsigned long from, unsigned long num, int turn_on);*
  I/Oアドレス *from* から, *num* 個のアドレスを操作する. *turn_on* に1を設定することで, 該当範囲への読み書きが許可される.
  この関数は特定の領域のみ許可が下りるため, プログラムミスによってシステム

に悪影響を与えにくい. しかし, 対応アドレス 0x3ff までのみであるため, それ以上については, 次の *iopl()* を使用する必要がある.

- *int iopl(int level);*
  I/O ポートへのアクセスを一括して許可する. アクセスを許可するには *level* に 3 を与える (通常は 0).

安全性を考えると前者が推奨されるが, 近年の PCI バス用の I/O ボードは一般に 0x3ff 以内のアドレスにはならないため, プログラムの先頭に, ただ *iopl(3);* と書くことにすれば良いかと思われる. なお, これらの関数の使用には, ヘッダファイル *unistd.h*, *sys/io.h* が必要となる (詳しくは man iopl, man ioperm されたい).

また, これらの関数の実行には root の権限が必要である. root になった (*su*) 上でプログラムを実行するか, root の所有にして (*chown root* 〜), setuid ビットを付加する (*chmod +s* 〜) 必要がある. 後者についてはしばしばセキュリティの面から危険性が指摘されるが, 制御実験用のコンピュータは一般に, 頻繁に root を使用することになるため, 気にする必要はないと考えられる. そもそも, セキュリティに不馴れな者がしばしば root になるような実験用のコンピュータは, 外部のネットワークから切り放す (電気的 or ファイヤウォール) べきである.

3.1.2 I/O ポートの読み書き

一旦, 許可を得た後は, I/O ポートの読み書きは自由に行うことが可能である.

- *unsigned char inb(unsigned short port);*
  unsigned short inw(unsigned short port);
  unsigned int inl(unsigned short port);
  port のアドレスから 1(inb), 2(inw), 4(inl) バイト読み込む.

- *void outb(unsigned char value, unsigned short port);*
  void outw(unsigned short value, unsigned short port);
  void outl(unsigned int value, unsigned short port);
  port のアドレスに 1(outb), 2(outw), 4(outl) バイトの値 *value* を書き込む.

これらの関数はヘッダファイル *asm/io.h* において定義されており, コンパイル時には *gcc −O* と最適化オプションを使用する必要がある. 注意点としては, MS-DOS 系のコンパイラとは *out?*の引数順序が異なることが挙げられる程度である. なお, 許可を得ずにこれらの関数を実行すると Segmentation Fault (SIGSEGV) を起こす.

3.2 物理メモリへのアクセス

多くの入出力ボードは I/O ポートへのアクセスで操作可能であるが, 一部にメモリ領域の使用を必要とするボードがある (共有メモリボードなど). その場合には, デバイス */dev/mem* を使用する.

このデバイスは, *read()/write()* 関数によって読み書きするとそれが物理的なメモリアドレスへの読み書きとなる. 読み書き位置の指定には, 一般的なファイル同様 *lseek();* を用いる. そのため, 操作手順としては

- *open("/dev/mem", O_RDONLY(O_RDWD));* によりデバイスを開く
- *lseek(fd, address, SEEK_SET);* により, アドレスを指定
- *read/write(fd, buf, count);* により, 読み書き

となる. 読み書きするとその分アドレスがずれてしまうため, 同アドレスを再参照する場合には *lseek();* を行う必要がある.

Linux 本体を含め, すべての使用中のメモリにアクセス可能であり, アドレスを誤ることはシステムに重大な影響を及ぼす可能性が高い. そのため, メモリへのアクセスには十分な注意が必要である.

そのほか, *mmap()* というシステムコールを利用すると, *read/write()* を使用せず, メモリをメモリとして直接アクセスすることが可能となる.

3.3 PCI バスハードウェアの情報収集

PCI バスの特徴の一つに I/O アドレス等の自動設定が挙げられる. これは, 多くの利用者にとっては非常に便利な機能であるが, ハードウェアのアクセスを試みる立場からは, アドレスを固定できないため多少厄介である. PCI バス上のハードウェアを自前で操作したい場合は, まず, アドレスを得なければならない.

この目的のためには */proc/pci* が便利である. *cat /proc/pci* とすると PCI バス上のデバイスの情報一覧が出力される.

```
% cat /proc/pci
  PCI devices found:
    Bus  0, device  16, function  0:
      Multimedia video controller: Intel SAA7116 (rev 0).
        Medium devsel.  IRQ 9.  Master Capable.  Latency=64.
               :
    Bus  0, device  15, function  0:
      Bridge: Unknown vendor Unknown device (rev 1).
```

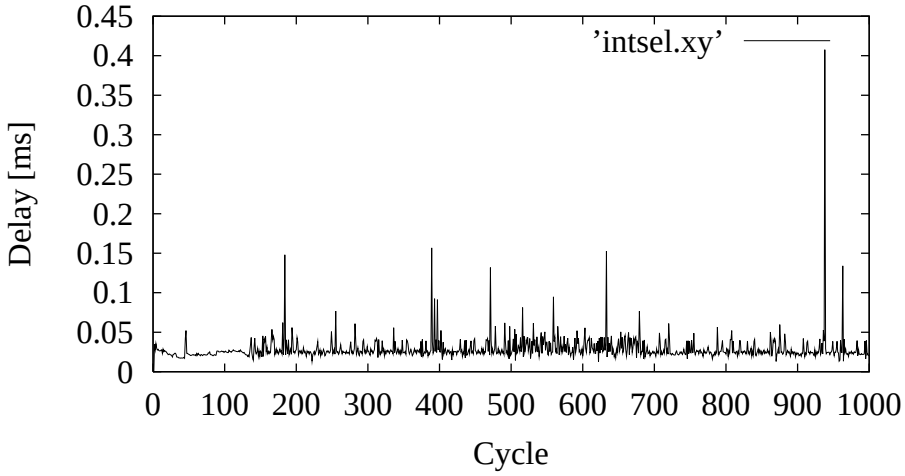

図 3.1　カーネル内部における割り込み検出からプロセス通知までの遅延時間

```
Vendor id=136c. Device id=9054.
Medium devsel.  Fast back-to-back capable.  IRQ 10.
Non-prefetchable 32 bit memory at 0xec002000.
I/O at 0x1400.
I/O at 0x14f0.
    :
```

この中には, 追加した拡張ボードの他, マザーボード上の IDE などのデバイスも含まれる. 目的のボードを探すには, メーカ, デバイス名を頼りにすることになるが, 多くの拡張ボードは Linux で登録されていない. この場合は “Vendor id”, “Device id” を頼りにする. 前者はメーカを, 後者はメーカで決定したハードに固有の番号である. これらの情報はボードのマニュアルに記載されていることが期待される. 無い場合には http://www.pcisig.com/membership/vendors/index.html にて前者は調べることができるが, 最終的には消去法で探すことになろう.

次に, 情報を読み取る. この例で示す Vendor=136c, Device=9054 のボードはアドテックシステムサイエンス社の 48bit パラレル入出力ボードである. マニュアルには連続 16 バイトのアドレスを使うことが記載されているが, 0xec002000 番地から始まるメモリ, 0x1400, 14f0 それぞれから始まる I/O ポート, 割り込み IRQ10 が割り当てられていることを確認できる. I/O ポートが 2 系統あり, どちらが利用すべきアドレスかが不明であったが, 多少操作を試みて, 0x14f0 が目的のアドレスであることが確認できた. なお, 後日メーカに問い合わせたところ, 他の 2 者は, ボードの管理情報にアクセスするための物であるとの回答を得ている.

3.4 割り込みの利用

ハードウェアの操作を行う場合, 割り込みを使用したい場合がある. この場合, 一般的にはデバイスドライバを作成することになる.

しかしながら, 割り込みにさほど応答性が要求されず, 割り込みがあったことが分かれば良い, という程度であれば, プロセスで割り込みをうけることも不可能ではない. これには簡単な割り込みを受けるだけのデバイスドライバを使用し, プロセスは割り込みのあったことをこのドライバに select() システムコール等を用いて通知してもらう[10]. その応答例を**図 3.1**に示す. これは, 366[MHz] の Celeron を 2 個搭載したコンピュータにおいて, パラレルポートの割り込みを使用して実験した物である (1k-Linux 2.2.10, 負荷状態としては X 端末化しており, 現在この原稿を執筆中, 程度). 数十から数百 [μs] で応答していることが確認できる. 原理的には前に述べた周期実行のタイマ割り込みが, 別の割り込みに変ったとの見方もできる. この場合もやはり応答時間は保証されてはいないが, 試験的な用途には問題ない品質と思われる. ただし, 割り込みに必ず即時応答しなければならないような用途では, 完璧を期すにはデバイスドライバを実装べきであろう. 今回は触れないが, デバイスドライバを作成することは難しいことではなく, C によるプログラム経験があれば十分作成可能である[10]. ただ, 作成過程に要する注意と危険性が, 通常のプログラムのそれより大きいだけである.

3.5 まとめ

本章では簡単であるが, Linux の通常のプロセスからハードウェアにアクセスする手法について述べた. I/O ポートに関しては基本的に問題なく, メモリに関しても今回は触れなかったが mmap を併用すれば直接的に操作可能である[10]. また, PCI の情報も容易に得られ, 割り込みすらある程度利用可能であるため, MS-DOS などに比較し, 不自由はしないと思われる. 制御のみならず, ハードウェアのテストなどを行う際に参考になれば幸いである.

第4章

おわりに

以上のように, 素の Linux を用いて制御を行うことは, 可能である. 確かに実行周期に保証はなく, まれに周期をはずしたりすることがあるが, 実験上これらが問題になることは, ほとんどないと思われる.

本手法の理念は “手軽に 安全に 十分な性能を” である. 不要に完璧な性能は求めない. もちろん, リアルタイム OS を使いこなせるのであれば, それに越したことはない. しかし, それほど複雑なシステムではなく, クリティカルな性能が不要であるなら, 導入の手間やプログラム作成時に要される知識等がほとんど不要で安全性・安定性の高い本手法は, 研究・実験用としては有効な選択肢の一つであると考えている.

このような最も身近な OS による方法がこれまで広く利用されてこなかったのには致命的な問題があるのではないか, との疑問もあろう. しかし, 少なくとも, 筆者らが 1 年間, 画像処理などを含む, 2 脚歩行ロボットのマルチプロセス制御に用いて問題が見受けられなかった実績と, 頂いていてる他の複数の実績のご連絡が, その疑問への答になると思われる.

なお, 本原稿では紙面の関係上, 解説部分が多く, 具体的な実例を省略したところがあり, 実際のソースリストなども割愛しているため, 実践性が不足している. より実践向きの情報は WEB ページにて公開している.

http://www.mechatronics.mech.tohoku.ac.jp/~kumagai/linux/

ここでは, 本手法の解説と, おそらく必要はないと思うが, デバイスドライバの作成法について解説している.

参 考 文 献

(1) “Real-Time Linux”, http://www.rtlinux.org/

(2) 船木陸議, 羅正華: “LINUX リアルタイム計測/制御 開発ガイドブック”, 秀和システム (1999)

(3) “日本機械学会 ロボティクス・メカトロニクス講演会’99 チュートリアルセミナー資料 「普及型多足歩行ロボットの機構・制御系・歩容制御入門」 – RT-Linux を用いたリアルタイムシステムの構築 –”, 日本機械学会 (1999)

(4) 石綿 陽一, 松井 俊浩, 国吉 康夫: “高度な実時間処理機能を持つＬｉｎｕｘの開発”, 第１６回日本ロボット学会学術講演会予稿集, 355/356(1998)

(5) 石綿 陽一: “ART-Linux 誕生の経緯と使い方”, “リアルタイム制御を実現する ART-Linux の設計と実装” 柴田智広: “ART-Linux によるリアルタイム処理への適用と応用” インターフェース 1999 年 11 月号, CQ 出版社 (1999)

(6) 加賀美聡: “ロボット研究のための PC/AT 互換機上のリアルタイム OS”, 日本ロボット学会誌 Vol.16 No.8, 8/13(1998)

(7) 熊谷正朗: “汎用 Linux によるロボット制御手法の提案 (第 1/2 報)”, 第 17 回 日本ロボット学会学術講演会予稿集, 847/850(1999)

(8) “Intel Architectrue Softoware Developer’s Manual Volume 3: System Programming Guide” Order Number 243192, 14-14, Intel Corporation(1997) (http://www.intel.com/より PDF ダウンロード可能)

(9) 熊谷正朗: “非 RT-Linux によるロボット制御”, http://www.mechatronics.mech.tohoku.ac.jp/~kumagai/linux/, (2000)

(10) 熊谷正朗: “制御屋さんのための Linux デバイスドライバ製作入門”, http://www.mechatronics.mech.tohoku.ac.jp/~kumagai/linux/, (2000)