第3章

無線センサノード向け
ハードリアルタイムオペレーティングシステム

3.1 はじめに

無線センサネットワークは小型，数が膨大，低消費電力という特徴を持っているため，今までに取得することができなかった粒度で空間の情報を取得することが可能になる．そのため，無線センサネットワークはビルオートメーションや自然科学，農業，軍事，品質管理などさまざまな分野への応用が期待されている．このような無線センサネットワークの多様な分野への応用を促進するためには，多様なアプリケーションや無線通信プロトコルを簡単に開発するためにオペレーティングシステムなどの基盤技術の整備が必須である．

現在，無線センサネットワークのオペレーティングシステムとしてTinyOS[9]が標準として扱われている．筆者らは2002年から無線センサネットワークの研究を進めており，そのとき既に標準になりつつあったTinyOSをわれわれの研究に用いることを検討した．しかしながら，TinyOSがevent modelを用いているためにプログラムが書き辛いこと，ハードリアルタイム性をサポートしていないことの2点の理由から筆者らの研究の要求に合致しなかったため，独自にオペレーティングシステムを開発することにした．

このような背景から開発されたオペレーティングシステムが本章で述べるPAVENET OSである．PAVENET OSは，thread modelを用いることでユーザに対してプログラムの書きやすさとハードリアルタイム処理のサポートの2つをユーザに対して提供する．PAVENET OSではpre-emptiveとco-operativeのハイブリッドのスケジューラを具備している．pre-emptiveのスケジューラではCPUの動的な優先度割り込みの機能とハードウェアによるコンテキストスイッチの機能を利用することで少ないオーバヘッドでハードリアルタイム処理を実現する．co-operativeのスケジューラでは，`os_yield`や`sleep`などユーザが明示的にCPUを開放する関数を実行することで排他制御やコンテキストスイッチの負荷を軽減する．さらに，無線センサノードにおけるタスク間のデータの交換が無線通信における各層のパケットの交換時に多発することに着目し，オペレーティングシステムとして無線通信プロトコルの階層化の機能を提供して排他制御をAPIに隠蔽することで各層のモジュール性を実現する．

このような特徴を持つPAVENET OSをTinyOSと比較評価を行った．PAVENET OS上で作られたプログラムはTinyOSと同程度の計算資源で実現できる．例えば，TinyOSのサンプルプログラムであるBlinkと同じ機能をPAVENET OSで実現した場合にはTinyOSではデータメモリが44byte，プログラムメモリが1428byte必要であるのに対し，PAVENET OSではデータメモリの使用量は63byte，プログラムメモリが1183byteで実現できる．また，パフォーマンスも計算資源と同様にTinyOSと同程度の性能を実現できる．一方で，PAVENET OSはTinyOSと異なり，ハードリアルタイム処理を実現する機構を提供する．そのため，TinyOSでは

実現できないような，無線通信を行いながら正確な 100Hz を刻むといったハードリアルタイム処理が実現できる．

このような PAVENET OS を用いて筆者らは，センサとアクチュエータを連携させるためのデバイス連携フレームワーク [10] や，空間に高密度に配置された無線センサノードを用いて行う実地震モニタリングシステム [49]，ソーラパネルから供給される電力のみで動作する無電源無線センサネットワーク [50] などの研究を行っている．

本章ではまず，3.2 において TinyOS の使い辛い点とその使い辛い点の原因を示し，本研究のモチベーションを明らかにする．それを受け，3.3 では PAVENET OS の設計と実装について述べる．3.4 では，PAVENET OS と TinyOS の比較評価について述べる．3.5 で関連研究との違いを明らかにし，最後に 3.6 でまとめとする．

3.2 Why TinyOS Is A Bad Idea

3.2.1 TinyOSの問題点

無線センサネットワークは，無線センサノードを空間に散布し，無線センサノード同士が通信を行いながら膨大な量の空間情報を取得するための技術である．無線センサノードはセンサからセンサ情報を取得，解析，センサノード同士でホップバイホップで基地局（シンクノード）までルーティング，などの処理を行う．さらに無線センサネットワークではアプリケーションに応じて通信に求められる要件が大幅に変わってくるため，未だMACプロトコルに関する試行錯誤が続いている[51, 52, 44, 45, 43, 42, 53, 41]．つまり，無線センサノードではソフトウェアでアプリケーションから通信プロトコルまでの機能を効率よく開発し，評価するための使いやすいオペレーティングシステムが求められている．

このような背景の中で，本研究のスタート地点は「TinyOSで本当にいいのだろうか?」という疑問であった．筆者らは2002年から無線センサネットワークの研究を進めてきた．当時すでにカリフォルニア大学のバークレー校において開発されたMICA mote[7]上で動作するnesC[8]で作成されたTinyOS[9]がセンサネットワークの分野の標準として扱われていた．筆者らも当初，研究の基盤としてTinyOSを使用することを検討したが，検討していく中で「なぜTinyOSを皆使っているのだろうか?」という疑問が生まれた．その理由は2つある．

1つ目の理由はTinyOSではプログラムが非常に書き辛いということである．TinyOS上で1秒おきにLEDを点滅させるプログラムを以下に示す．

```
[Blink.nc]
configuration Blink {
}implementation {
    components Main, BlinkM, SingleTimer, LedsC;
    Main.StdControl  -> BlinkM.StdControl;
    Main.StdControl  -> SingleTimer.StdControl;
    BlinkM.Timer  -> SingleTimer.Timer;
    BlinkM.Leds  -> LedsC;
}

[BlinkM.nc]
module BlinkM {
    provides {
        interface StdControl;
    }
    uses {
        interface Timer;
        interface Leds;
    }
}
implementation {
    command result_t StdControl.init() {
```

```
        call Leds.init();
        return SUCCESS;
    }
    command result_t StdControl.start() {
        return call Timer.start(TIMER_REPEAT, 1000) ;
    }
    command result_t StdControl.stop() {
        return call Timer.stop();
    }
    event result_t Timer.fired() {
        call Leds.redToggle();
        return SUCCESS;
    }
}
```

TinyOS では LED を点滅させるだけで，このような非常に冗長なコードをかかなければならない．

2つ目の理由は TinyOS ではハードリアルタイム処理が実現できないことである．TinyOS では [9] においてセンサネットワークではソフトリアルタイム処理ができれば十分に対応できると述べている．しかしながら，筆者らの経験ではソフトリアルタイム処理だけでは十分ではない．筆者らはこれまで無線センサネットワークにおける無線通信プロトコルからアプリケーションまでを統合的な観点から研究を進めてきた [50, 54, 55, 49, 56, 57, 10]．その中で，たしかにソフトリアルタイム処理だけでも実現できるアプリケーションも存在した．その一方で，MAC プロトコルを TDMA で実装しようとしたり，正確な 100 Hz のサンプリングで地震のゆれを計測するといったアプリケーションを実現しようとする場合にはソフトリアルタイム処理では十分ではなくハードリアルタイム処理が必要となる．例えば実行周期とデッドラインが 26 μs，計算時間が 12.5 μs の無線通信物理層のタスクと，加速度を正確な 100 Hz のサンプリング周期で取るために実行周期が 10 ms，デッドラインと計算時間が 2.2 μs のタスクを同時にこなさなければならないという状況は容易に発生する．TinyOS は，このようなタスクを効率的に実行するような仕組みを持ち合わせていない．

以上のような背景により，筆者らは

- プログラムの書きやすさ
- ハードリアルタイム処理のサポート

の2つを満たすオペレーティングシステムを実現することを目指す．

3.2.2 Threads v.s. Events

3.2.1 に挙げた TinyOS の 2 つの使い辛い点は，TinyOS が event model で構築されていることに起因する．

システムを構築するのに event model を用いるか，thread model を用いるかは長い間議論されてきた [58, 59, 60, 9, 61]．本論文では thread model を「固定的な数の大きなタスクを複数の実行ストリームによって実行するモデル」，event model を「たくさんある非常に小さいタスクを 1 つの実行ストリームで実行するモデル」と定義する．1979 年に発表された [58] では event model で構築されたシステムは thread model でも構築可能であり，性能や必要とされる計算資源には差が無いことが示された．しかしながら，1996 年に発表された [59] では，thread model よりも event model の方がよいという主張がされている．[59] では，thread model を用いるとタスクが pre-emption されるため，コンテキストスイッチや同期のオーバヘッド，デッドロックなどの問題が発生するので event model を用いた方がよい，と結論付けられている．しかしながら，[59] の主張はある 1 つの誤った前提を置いている．確かに既存の POSIX thread や Win32 thread は pre-emption を前提とした time-sliced multithreading であるが，2004 年に発表された [60] で述べられているように，thread model を用いた場合でもタスクを pre-emption せずに co-operative でスイッチすることで event model を用いた場合と同等のシステムとして扱うことができる．また，event model を用いた場合ではタスクが細かく分割されるので TinyOS のソースコードで示したように制御フローが把握し辛くなる．それに対して thread model を用いると制御フローが明確になるというメリットが存在する．例えば LED を 1 秒毎に点滅させるアプリケーションを thread model を用いると

```
void thread(void)
{
  while(1){
    toggle_led();
    sleep(1);
  }
}
```

のように記述することができる．先ほどの TinyOS のコードと見比べても書きやすさや制御フローの把握のしやすさは thread model を用いた方がよいことは明らかである．また，TinyOS[9] や Contiki[61] によると，thread model を用いた場合には各スレッド毎にスタックを用意しなければならないので必要とされる計算資源が多いと主張されている．しかしながら，各スレッドにスタックを用意しなくてもよいような仕組みを実現できれば event model を用いた場合と同程度の省資源性を実現できる．以上の議論を踏まえて，筆者らは「プログラムの書きやすさ」を実現するために thread model を用いてオペレーティングシステムを構築する．

thread model を用いる場合に考えなければならないのは pre-emption が必要あるのか，無いのか，である．結論から述べるとハードリアルタイム処理を実現したい場合には pre-emption は必須である．しかしながら，pre-emption はタスクスイッチのオーバヘッドが非常に大きい．タスクスイッチの主なオーバヘッドとしては，コンテキストの切り替え（CPU の状態の保存と復元）とタスクの優先度の判定の 2 つである．また，pre-emption する場合には共有データに対して排他制御を行わなければならない．特に無線センサネットワークでは無線通信の各層の間でデータを共有するため，スレッド間での排他制御の仕組みは必須である．

3.3 PAVENET OS

4.2での議論を踏まえて，筆者らは無線センサノード向けのオペレーティングシステムであるPAVENET OSの設計と実装を行った．PAVENET OSでは，thread modelを用いることでユーザに対してプログラムの書きやすさを提供する．また，リアルタイム性が必要となるスレッドをpre-emptiveで実行する．このとき，pre-emptive multithreadingの実現にCPUの機能を積極的に利用することでコンテキストスイッチ時のオーバヘッドを削減する．さらに，リアルタイム性の必要無いスレッドをco-operative multithreadingで実行することで，イベントを用いた場合と同様な特徴を実現しつつもユーザに対してプログラムの書きやすさを提供する．

また，無線センサネットワークにおいて共有メモリにアクセスする場合に問題になってくるのは通信レイヤ間でのパケットの受け渡しの時である．これに向けて，各層の間のインタフェース内で排他制御する仕組みをオペレーティングシステムが提供することでハードリアルタイムタスクを処理する場合でもユーザが排他制御を意識せずに無線プロトコルを開発できる仕組みを提供する．

3.3.1 ハードリアルタイムタスクスケジューラ

タスク	計算時間 (C)	デッドライン (D)	周期 (T)
無線 L1	12.5 μs	26 μs	26μs
無線 L2	10 μs	12 μs	6.7 ms
シリアル通信	10μs	69 μs	69 μs
地震モニタリング	2.2 μs	2.2 μs	10 ms

表 3.1: 無線センサノードの処理

われわれの経験によると，無線センサノードにおけるタスクは

計算時間 $\leq$ デッドライン $\leq$ 周期

の特性を持っているものが多く存在する．表 3.1に無線センサノードのタスクの種類とその特性を示す．無線L1はMICA Mote[7]やSmart-Its[62]などの無線センサノードで用いられているChipcon社のCC1000[63]の物理層の処理をする場合の値である．CC1000では無線で1 bit送受信する度にCPUに対して割り込みが発生するため，次の1 bit送受信するまでに終わらせなければならないような処理が頻繁に発生する．無線L2はMACプロトコルとしてTDMAを実装した時の値である．TDMAは各ノードで同期をとり，自分に与えられたタイミングのみで通信

を行うので正確なハードリアルタイム処理が重要である．ガードタイムを多めに取るなどしてデッドラインをもう少し長くすることも可能であるが，その場合には消費電力や無線帯域の利用効率が低下する．シリアル通信は 115.2 kbps で通信を行ったときの受信処理の値である．シリアル通信では 1 byte 受信する度に割り込みが発生し，次の 1 byte を受信する前に受け取った 1 byte を処理する必要がある．地震モニタリングは，100 Hz で加速度の取得をしたときの値である．地震のモニタリングのように実際の地震工学などに無線センサネットワーク技術を応用することを考えた場合にはサンプリング周期の正確さが非常に重要になってくるため [49, 64]，ハードリアルタイム処理は必須である．

PAVENET OS ではこれらのハードリアルタイム処理を少ないオーバヘッドで実現するために Microchip 社の CPU である PIC18[65] の動的な優先度割り込みを使用して deadline-monotonic scheduling[66] を実現する．deadline-monotonic scheduling は 1 つのプロセッサの場合に最適な優先度割り当てが可能であることが知られている [66]．

PIC18 はタイマ割り込み，ポートチェンジ割り込み，外部割込み，シリアル送信割り込み，シリアル受信割り込みなどさまざまな割り込みを持つ．PIC18 ではこれら全ての割り込みを高優先度割り込みか低優先度割り込みかの 2 種類から選択することができる．低優先度割り込みベクタは 0018h 番地であり，高優先度割り込みベクタは 0008h 番地である．また，高優先度割り込みは低優先度割り込み実行中でも割り込むことが可能である．さらに，PIC18 は高優先度の割り込みを高速に行う機能を持っており，割り込み時のコンテキストの保存と復元をハードウェアで実行する．この PIC18 のハードウェアによる動的な割り込み優先度判定の機能と，コンテキストの保存と復元のハードウェア処理の機能を利用することで，少ないオーバヘッドでハードリアルタイム処理を実現する．

PIC18 の各割り込みはレジスタに割り込み優先度ビット，割り込み有効ビット，割り込みフラグビットの 3 つのビットを持っている．たとえば PIC18 の持つタイマ 0 の割り込みは優先度ビットが TMR0IP，有効ビットが TMR0IE，フラグビットが TMR0IF である．割り込み優先度ビットにはその割り込みが高優先度 (1) なのか低優先度 (0) なのかが設定される．割り込み有効ビットにはその割り込みが現在有効なのか (1) 無効なのか (0) が設定される．割り込みフラグビットは，割り込み有効ビットが 1 に設定されている場合に割り込みフラグビットが 1 に設定されると割り込み優先度ビットに応じたレベルの割り込み処理が実行される．

筆者らは，このような PIC18 の動的な優先度割り込みの機能を最大限に利用することで少ないオーバヘッドでハードリアルタイム処理を実現する．まず，各割り込みには

```
void (*task_timer0)(void); //タイマ 0
void (*task_timer1)(void); //タイマ 1
```

```
void (*task_int1)(void);    //外部割込み 1
void (*task_int2)(void);    //外部割込み 2
void (*task_rc)(void);      //シリアル受信割り込み
```

のように関数ポインタが用意されている．ハードリアルタイム用のタスクを追加するための API では

```
void add_rtask(uint8 isr_type,
               uint8 priority,
               void (*func)(void))
{
    switch(isr_type){
    case ISR_TMR0:
        if(priority == ISR_HIGH)
            TMR0IP = 1;
        else if(priority == ISR_LOW)
            TMR0IP = 0;
        TMR0IE = 1;
        task_tmr0 = func;
        break;
    case ISR_TMR1:
        :
        :
```

のように，各割り込みに対して CPU の割り込みレベルを設定し，関数を割り当てる．高優先度と低優先度の割り込みベクタは

```
0008h: call isr_high
000Ah: nop
     :
     :
0018h: call isr_low
001Ah: nop
```

のように記述されており，isr_high は

```
void isr_high(void)
{
    if(TMR0IP && TMR0IE && TMR0IF){
        TMR0IF = 0;
        task_timer0();
    }
    if(TMR1IP && TMR1IE && TMR1IF){
        TMR1IF = 0;
        task_timer1();
    }
    if(INT1IP && INT1IE && INT1IF){
        INT1IF = 0;
        task_int1();
    }
        :
        :
```

のように，isr_low は

```
void isr_low(void)
{
    if((TMR0IP == 0) && TMR0IE && TMR0IF){
        TMR0IF = 0;
        task_timer0();
```

```
}
if((TMR1IP == 0) && TMR1IE && TMR1IF){
    TMR1IF = 0;
    task_timer1();
}
if((INT1IP == 0) && INT1IE && INT1IF){
    INT1IF = 0;
    task_int1();
}
    :
    :
```

のように記述されている．つまり，PAVENET OS では同じ優先度のタスクを複数使用することを許している．

このような PAVENET OS のリアルタイムスケジューラのスケジューリング可能性の十分条件は deadline monotonic scheduling[67] のスケジューリング可能性のテストで調べることができる．[67] で用いられているスケジューリング可能性のテストを以下に示す．

> プロセス τ_i が周期 T_i，計算時間 C_i，デッドライン D_i，の特性を持っているとする．τ_1 が最も優先度の高いタスクであり，τ_n が最も低い優先度のタスクである．このとき
>
> $$\forall i : 1 \leq i \leq n : \frac{C_i}{D_i} + \frac{I_i}{D_i} \leq 1 \tag{3.1}$$
>
> を満たしているとき τ_i はスケジューリング可能である．I_i は他のプロセスによる干渉であり，
>
> $$I_i = \sum_{j=1}^{i-1} \left\lceil \frac{D_i}{T_j} \right\rceil C_j \tag{3.2}$$

で表される．

式 (3.1) (3.2) は n 個の優先度を持ち，1 つの優先度に割り当てるタスクは 1 つまでだという前提である．しかしながら PAVENET OS では 2 つの優先度しか持たず，かつ 1 つの優先度に対して 1 つ以上のタスクを割り当てることが可能であるので以下のように式 (3.1) (3.2) を次のように変形する．

まず，n 個の高優先度のタスクがあった場合，高優先度タスク τ_i は

$$\frac{C_i}{D_i} + \frac{I_i}{D_i} \leq 1$$

の時スケジューリング可能である．このとき，

$$\begin{aligned} I_i &= \left(\sum_{j=1}^{n} \left\lceil \frac{D_i}{T_j} \right\rceil C_j\right) - \left\lceil \frac{D_i}{T_i} \right\rceil C_i \\ &= \left(\sum_{j=1}^{n} \left\lceil \frac{D_i}{T_j} \right\rceil C_j\right) - C_i \end{aligned}$$

つまり，高優先度タスク τ_i は

$$\begin{aligned} \frac{C_i}{D_i} + \frac{I_i}{D_i} &= \frac{C_i}{D_i} + \frac{\left(\sum_{j=1}^{n} \left\lceil \frac{D_i}{T_j} \right\rceil C_j\right) - C_i}{D_i} \\ &= \sum_{j=1}^{n} \left\lceil \frac{D_i}{T_j} \right\rceil \frac{C_j}{D_i} \leq 1 \end{aligned} \tag{3.3}$$

の時，スケジューリング可能であると言える．

次に，低優先度のタスク τ_k について考える．n 個の高優先度のタスク，m 個の低優先度のタスクがあった場合，低優先度タスク τ_k は

$$\frac{C_k}{D_k} + \frac{I_k}{D_k} \leq 1$$

の時スケジューリング可能である．このとき，

$$\begin{aligned} I_k &= \left(\sum_{l=1}^{n+m} \left\lceil \frac{D_k}{T_l} \right\rceil C_l\right) - \left\lceil \frac{D_k}{T_k} \right\rceil C_k \\ &= \left(\sum_{l=1}^{n+m} \left\lceil \frac{D_k}{T_l} \right\rceil C_l\right) - C_k \end{aligned}$$

つまり，低優先度タスク τ_k は

$$\begin{aligned} \frac{C_k}{D_k} + \frac{I_k}{D_k} &= \frac{C_k}{D_k} + \frac{\left(\sum_{l=1}^{n+m} \left\lceil \frac{D_k}{T_l} \right\rceil C_l\right) - C_k}{D_k} \\ &= \sum_{l=1}^{n+m} \left\lceil \frac{D_k}{T_l} \right\rceil \frac{C_l}{D_k} \leq 1 \end{aligned} \tag{3.4}$$

の時，スケジューリング可能である．

3.3.2 ベストエフォートタスクスケジューラ

PAVENET OSはハードリアルタイム性の必要な処理は3.3.1に示したpre-emptive multithreadingで実現される．それ以外の処理は，ベストエフォートタスクスケジューラによって実行される．無線センサノードにおけるルーティングプロトコルの処理や，フラッシュメモリに対する遅延書き込みの処理，センサ情報の問い合わせに対して返事を返す処理などはベストエフォートで十分に処理可能である．

ベストエフォートタスクスケジューラはできるだけ単純で軽量な仕組みで実現するために，multithreadingの実現にいくつかの制限を設ける．まず，1つめの制限はタスクのスイッチをco-operativeで行うことである．co-operativeでタスクスイッチをすることで共有資源に対するアクセスや，CPUのコンテキストの保存と復元が不要になり，プログラムカウンタを管理するだけでスレッドを実現できる．2つ目の制限は，タスクをスイッチさせるシステムコールを呼ぶことのできる場所の制限である．PAVENET OSではスレッドから呼ばれた関数の中からはタスクをスイッチさせるシステムコールを呼べないようにしている．コールスタックを保存することでスレッドから呼ばれた関数の中でもタスクをスイッチさせることは可能であるが，新たにコールスタックを保存するための計算資源が増加するのでこのような制限を設けた．3つ目の制限は1つのコードからは1つのスレッドしか起動できないことである．この制限はPAVENET OSがコンパイル時にスレッドが使用するメモリを確保することに起因する．あらかじめコンパイラが使用するメモリを決めておくことでスレッドに対して動的にスタックを割り当てる必要が無くなり，少ない計算資源でスレッドを実現できる．

名前	サイズ	内容
tid	8 bit	スレッドID
state	8 bit	スレッドの状態
pc	16 bit	プログラムカウンタ
sleep_time	8 bit	待ち状態の残り時間

表 3.2: タスク制御ブロック

ベストエフォートタスクスケジューラのTCB（タスク制御ブロック）を表3.2に示す．tidはスレッドのIDである．スレッド作成時にシステムがスレッドに対して割り当てる．stateはスレッドの状態である．スレッドはデッド状態，実行状態，スリープ状態，待ち状態の4つの状態を持つ．タスク操作関数には表3.3の7つが存在する．pcはプログラムカウンタである．現在実行中のスレッドのプログラムカウンタが記録されている．sleep_timeはタスクがスリープ状態から実行状

態に遷移する時間である．PAVENET OS では 100 ms に 1 つ増加する jiffies という変数を具備しており，12.7 秒までのスリープをサポートしている．12.7 秒以上のスリープはユーザがプログラム内で時刻を記録し，自分で処理する必要がある．

関数名	処理	実行後の状態
add_task(*funcname*)	タスクを TCB に追加	実行状態
os_yield()	OS に制御を渡す	実行状態
sleep(*time*)	*time* 秒スリープ	スリープ状態
sig_wait()	シグナルを待機	待ち状態
suspend_task(*pid*)	*pid* を待ち状態に	実行状態
signal_task(*pid*)	*pid* を実行状態に	実行状態
kill_task(*pid*)	*pid* をデッド状態に	実行状態

表 3.3: タスク制御関数

表 3.3 に PAVENET OS が提供するタスク制御関数を示す．os_yield，sleep，sig_wait はタスクをスイッチさせるための関数である．os_yield のソースコードを以下に示す．

```
void os_yield(void)
{
    pcounters[current_task] = TOS;
    asm("pop");
}
```

TOS はプログラムカウンタを意味する．PAVENET OS では前述したようにさまざまな制限を加えているので，このような非常に単純な関数でタスクをスイッチすることができる．

タスクスケジューラのソースコードを以下に示す．

```
uint8 task_schedule(void)
{
    uint8 i;
    for(i = 0; i < task_num; i++){
        if(tcb[i].state == TASK_SLEEP){
            if(tcb[i].sleep_time == jiffies)
                tcb[i].state = TASK_RUN;
        }
        if(tcb[i].state == TASK_RUN)
            exec_task(i);
    }
}
```

このように PAVENET OS のベストエフォートタスクスケジューラは非常に単純な構成になっている．

3.3.3 無線プロトコルスタック

PAVENET OS では，ハードリアルタイム処理を実現するために pre-emptive multithreading を用いているため，スレッド間での共有データの排他制御が重要になってくる．筆者らは，無線センサノードにおける共有データのアクセスが通信レイヤ間のデータの受け渡し時に頻繁に発生することに着目し，オペレーティングシステムとして用意した通信レイヤ間の API 内に排他制御を隠蔽することで，ユーザが他の層のタスクを意識しなくても MAC プロトコルやルーティングプロトコルを開発可能な環境を提供する．さらに各レイヤの独立性を実現することでアプリケーションに応じてさまざまなプロトコルを組み合わせて使用することができる．

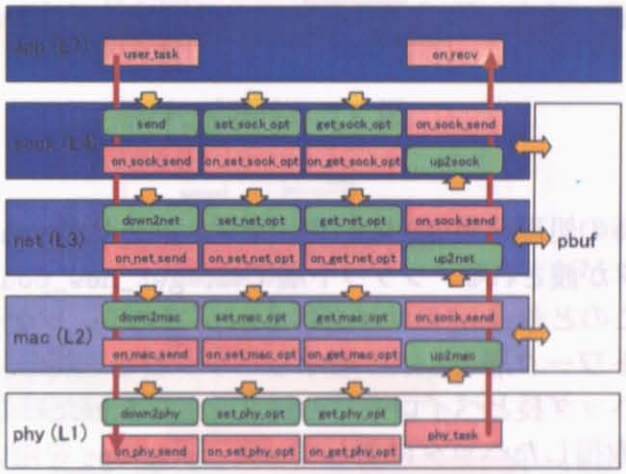

図 3.1: 無線プロトコルスタック

図 3.1 に PAVENET OS の無線プロトコルスタックを示す．PAVENET OS では通信層として物理層，MAC 層，ネットワーク層，ソケット層を提供する．また，レイヤ間のパケットの受け渡しを効率化するために BSD mbuf に似た pbuf という仕組みを提供する．

pbuf はある識別子に対してバッファを割り当てており，レイヤ間で識別子の受け渡しだけで済むように実装されている．pbuf 用の API を以下に示す．

```
uint8 get_new_pbuf(void);
byte *get_pbuf_next(uint8 index, uint8 size);
byte *get_pbuf_head(uint8 index);
uint8 release_pbuf(uint8 index);
uint8 get_pbuf_size(uint8 index);
```

これらのAPIの使われ方をパケット送信時と受信時における各層での処理を例に説明する．

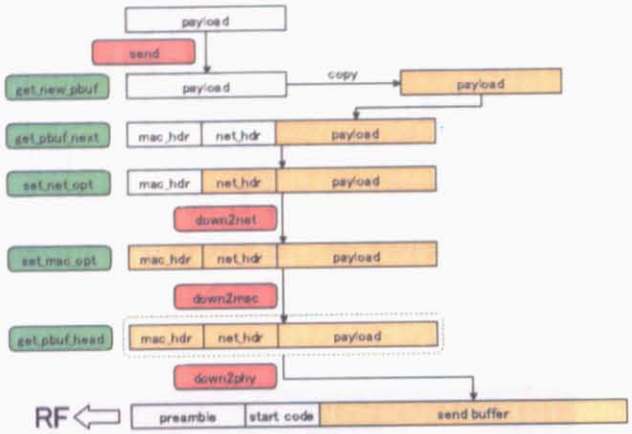

図 3.2: 送信時の処理

まず，送信時の処理を図 3.2 に示す．まず，ユーザが send を呼ぶとソケット層に対してデータが渡される．ソケット層では，get_new_pbuf を用いて新しい pbuf を作成する．このとき pbuf のサイズは 0 byte である．ソケット層は get_net_opt を用いてネットワーク層と MAC 層からヘッダ長を取得し，get_pbuf_next を用いて取得したヘッダ長とペイロードの長さを足した分だけの pbuf の領域を拡大する．そして，取得したヘッダに対して set_sock_opt を用いて宛先アドレスなどの書き込みを行い，down2net を用いてネットワーク層の送信キューに対して pbuf の識別子を書き込む．ネットワーク層の処理は co-operative multithreading で実現されているため，ソケット層からネットワーク層に渡すときの排他制御は必要無い．ネットワーク層では set_mac_opt を用いて pbuf に宛先アドレスなどの設定を行い，MAC 層の送信キューに対して pbuf の識別子を書き込む．MAC 層の処理はハードリアルタイムが必要となる場合がありうるので，識別子の書き込み時には down2mac 内で割り込み禁止などの排他制御を行う．MAC 層では，pbuf 識別子からパケットデータを抽出し，パケットデータを物理層の送信バッファに対して書き込む．物理層では送信バッファを 1 つだけ持っており，その送信バッファが使用中の場合には送信バッファに対する書き込みは禁止される．この場合には MAC 層に対してはビジーが返るので，MAC 層はビジーが終了するのを待つことで排他制御を行う．物理層は MAC 層から送信バッファにデータが書き込まれるとそのデータにプリアンブルとスタートコードを付与し，1 bit ずつハードリアルタイム処理で送信する．

次に受信時の処理を図 3.3 に示す．まず，物理層では受信バッファを 1 つ持ってい

る．物理層はハードリアルタイム処理でプリアンブルとスタートコードを検出すると受信バッファに対して復号した受信データを 1 bit ずつ書き込んでいく．受信バッファが満たされると MAC 層に対して受信データを渡す．物理層から MAC 層へのデータの受け渡しは受け渡し用の API である up2mac 内で排他制御で実現される．物理層から受信データを受け取った MAC 層は，受信データに対して get_new_pbuf と get_pbuf_next を用いて pbuf を割り当てる．そして割り当てた pbuf の識別子を up2net を介してネットワーク層の受信キューに対して書き込む．この時，up2net 内で受信キューへのアクセスの排他制御を行う．ネットワーク層では，受信キューから pbuf 識別子を受け取ってソケット層に対して識別子を渡し，ソケット層では受け取った識別子を用いて pbuf からペイロードを抜き出してペイロードのみを on_recv を用いてアプリケーション層に対して渡す．

このように，pbuf とレイヤ間の API 内の排他制御により，各層のモジュール性と開発のしやすさが提供される．

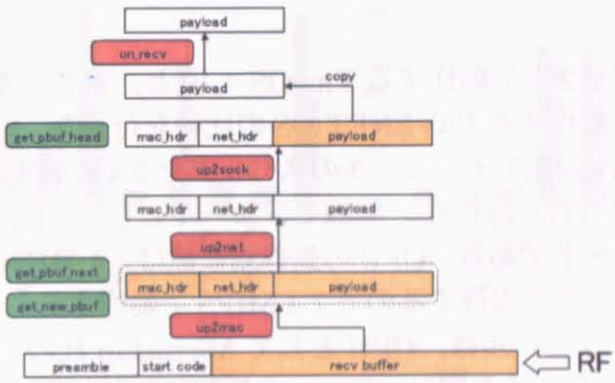

図 3.3: 受信時の処理

3.4 評価

	MICA2	PAVENET モジュール
動作周波数	7.4MHz	20MHz
処理速度	7.4MIPS	5MIPS
無線周波数	315MHz	315MHz
通信方式	FSK	FSK
電源	DC3V	DC3V
電流 (受信時)	30mA	30mA
電流 (sleep)	30μA	0.3μA
通信速度	19.2 kbps	38.4 kbps

表 3.4: 比較環境

筆者らは MICA2 上で動作する TinyOS 1.10 と，筆者らが開発した PAVENET モジュール上で動作する PAVENET OS の比較を行った．

MICA2 と PAVENET モジュールのスペックを 3.4 に示す．MICA2 は CPU として ATMEL 社の ATmega128L，無線モジュールとして Chipcon 社の CC1000 を具備したセンサノードである．CPU の動作周波数は 7.4 MHz であり，ATmega128L は 1 命令 1 クロックで実行するので 7.4 MIPS で動作する．

PAVENET モジュールは，CPU として Microchip 社の PIC18LF4620，無線モジュールとして MICA2 と同じ CC1000 を具備したセンサノードである．動作周波数は 20MHz であるが，PIC18 は 1 命令を 4 クロックで実行するので 5MIPS で動作する．無線の通信速度は普段は 38.4 kbps で通信を行っているが，MICA2 との公平性を保つために 19.2 kbps に速度を落として比較した．

3.4.1 プログラムサイズ

モジュール	RAM (byte)	ROM (byte)
Task Schedular	47	1536
Wireless Protocol Stack	628	930
合計	675	2466

表 3.5: プログラムサイズ

PAVENET OSの基本的な評価として，まずコア部分のメモリサイズを表4.4に示す．PAVENET OSではタスクスケジューラはデータメモリが47 byte，プログラムメモリが1536 byteである．これは作成可能なスレッドの数が最大5個の場合であり，スレッドの数に応じてスレッド1つあたりデータメモリが4 byte増減する．thread modelは，大きなタスクをなるべく少ない数で実現するため，event modelに比べてタスクスケジューラで必要とされる計算資源が少なくて済むという特徴を持っている．無線プロトコルスタックはRAMが675 byteと比較的多い．これは，物理層の送信バッファ，受信バッファ，各レイヤで共通に使用するpbuf，各レイヤが持つ送信キューや受信キューをシステムとして用意していることに起因する．TinyOSは何もモジュールを接続しなかった場合にはRAMが19 byte，ROMが473 byteである．

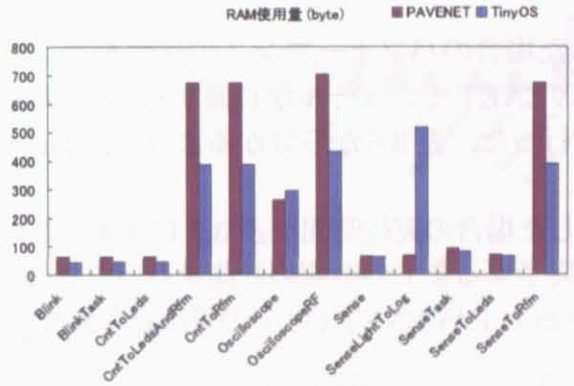

図 3.4: RAM使用量の比較

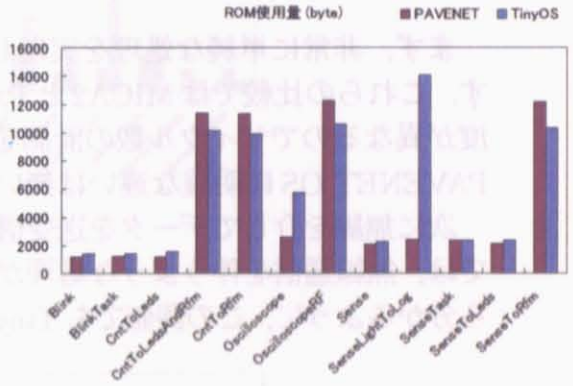

図 3.5: ROM使用量の比較

次に，TinyOSで用意されているプログラムと同じものをPAVENET OSで実装した場合のRAM使用量のグラフとROM使用量のグラフをそれぞれ図 3.4, 図 3.5に示す．グラフから分かるように，BlinkやCntToLedsなどの簡単な機能を実現した場合には必要とされるメモリのサイズに大きな差異は見られない．RAMサイズは若干TinyOSの方が省資源であり，ROMサイズは若干PAVENET OSの方が省資源である．しかしながら，無線通信を行う場合にはPAVENET OSの方がデータメモリ，プログラムメモリ共に大きく上回る．これはPAVENET OSがシステムとして無線プロトコルスタックを用意しており，バッファの部分がメモリを多く消費しているからである．SenseLightToLogのメモリ消費量がTinyOSが多く消費しているのは，PAVENET OSでは最低限の機能しか実装しなかったからである．たとえば，TinyOSではシリアルからログの取得周期などの変更を行うことのできる機能を具備しているが，これらの機能は今回の評価では実装しなかった．

3.4.2 実行性能

次に TinyOS で用意されているサンプルプログラムと同じものを PAVENET OS で実装して比較評価を行った．計測はそれぞれ 100 回の計測を行い，平均を出した．

タスク	TinyOS (cycle)	PAVENET OS (cycle)
Blink	19.5	8.5
BlinkTask	123.5	134.5
CntToLeds	155.5	147.0

表 3.6: 基本性能の比較

まず，非常に単純な処理を実現した場合のパフォーマンスの比較を表 3.6 に示す．これらの比較では MICA2 と PAVENET モジュールが具備する CPU の処理速度が異なるのでサイクル数の計測を行った．表 3.6 から分かるように，TinyOS と PAVENET OS に明確な違いは無い．

次に無線を介してデータを送受信した場合の実行時間を表 3.7 に示す．この比較では，無線通信を伴うような処理が終了するまでの時間の計測を行った．表 3.7 から分かるように，この評価でも TinyOS と PAVENET OS での差は見られなかった．

タスク	TinyOS (ms)	PAVENET OS (ms)
CntToLedsAndRfm	17.2	16.9
CntToRfm	17.1	17.0
SenseToRfm	17.4	17.1

表 3.7: 無線通信の比較

このように，TinyOS と PAVENET OS では処理性能による大きな違いは見られない．

3.4.3 コード量

図 3.6 に TinyOS のサンプルプログラムと同じ機能を PAVENET OS に移植した場合のコード量を示す．グラフから分かるように，同じ機能でも TinyOS と PAVENET OS ではユーザが書かなければならないソースコードの行数が異なる．

TinyOS ではモジュールの接続とモジュールの機能の記述のそれぞれ configuration file と implementation file に分かれている．そのため，既存のモジュールを組み合わせるだけで構築可能な `CntToLeds` のようなサービスは configuration file を記述するだけで開発できるのでユーザが書かなければならないソースコードの行数は少なくて済む．それに対して PAVENET OS ではユーザが処理を記述するとコンパイラが自動的に必要とされるモジュールを選択するので新しい機能を実現したい場合には少ない行数でプログラムを書くことができる．

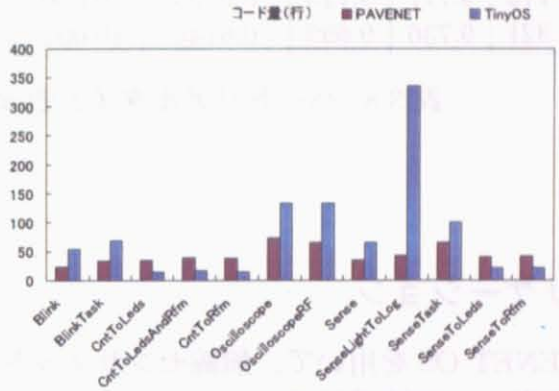

図 3.6: コード量

3.4.4 ハードリアルタイム処理

Tiny OS と PAVENET OS のハードリアルタイム処理の比較を行った．表 3.8 に TinyOS と PAVENET OS でそれぞれ正確な 100Hz を生成した場合の比較結果を示す．正確な 100Hz を生成するためには正確な 10ms 毎に処理を行う必要がある．

TinyOS ではタスクを 100Hz でセンサからの取得するというタスクのみを実装した場合にでも 100Hz を作ることができなかった上に，毎回のセンサの値を取得する間隔にばらつきがあった．さらに，無線通信を行いながら 100Hz のセンサの値の取得を行った場合では毎回のセンサの値を取得する間隔のばらつきが大きくなった．

PAVENET OS では，正確な 100Hz でのセンサの値の取得ができた．また，無線通信を行った場合はごくまれにセンサのデータを取得する間隔が 10.001ms になることがあった．これは無線通信の処理の中でデータの受信を完了して上位層にデータを渡す場合に PAVENET OS の無線プロトコルスタックの中でクリティカルセクションを経由することがあることに起因する．しかしながら，このクリティ

カルセクションを通る時間は非常に小さい上にごくまれにしか起こらないためデッドライン内に処理を完了できている.

以上のことにより，ハードリアルタイム処理の性能ではPAVENET OSの方がTinyOSに比べて優れていることが示された.

	TinyOS				PAVENET OS			
	最大 (ms)	最小 (ms)	平均 (ms)	標準偏差	最大 (ms)	最小 (ms)	平均 (ms)	標準偏差
100Hz	9.713	9.717	9.715	0.0125	10.000	10.000	10.000	0.0000
100Hz + RF	9.321	9.736	9.699	0.6748	10.001	10.000	10.000	0.0000

表 3.8: ハードリアルタイム処理

3.4.5 アプリケーション

筆者らはPAVENET OSを用いて，無線センサネットワークのさまざまなアプリケーションを実装し，無線通信プロトコルなどの研究を行っている.

ANTHは目覚まし時計や椅子，ミュージックプレーヤなど身の回りのさまざまなオブジェクトに組み込まれたセンサやアクチュエータを柔軟に連携させるためのフレームワークである[10]. ANTHはシングルホップの無線センサネットワークを構築するのでネットワーク層のプロトコルは持たない. MAC層では，A-MAC（ANTH-MAC）と呼ばれるセンサとアクチュエータを連携させるのに特化したMACプロトコルを用いることで無線通信における低消費電力化を実現している.

地震モニタリングは，多数のセンサノードを部屋の床や天井に高密度に設置し，各点での地震による加速度の変化を検出することで地震工学などへの応用や，空間の変形を抽出することによる構造ヘルスモニタリングへの応用などを目指している研究である[49][11]. 地震モニタリングでは，正確な100Hzが必要になるためPAVENET OSのハードリアルタイム処理は必須である. また，センサノード間で取得したデータの時系列上のずれを防ぐために各ノード間で1ms以下の同期を取っている. さらに，MACプロトコルがTDMAで通信すると共にネットワーク層でマルチホップネットワークを構築することで低消費電力な通信を実現している.

Solar Biscuitはソーラパネルによって獲得した電力をスーパキャパシタに蓄積し，一瞬だけ動作するという超間欠型の無線センサネットワーク技術である[50]. 主に，農業への応用を目指している. MAC層では，一定周期で間欠動作する. ネットワーク層では，各ノードのMAC層での周期をわずかにずらすことで超間欠動作していてもデータを運べるような仕組みを提供している.

3.5 関連研究

3.5.1 オペレーティングシステム

無線センサネットワークのオペレーティングシステムに関する研究としては，TinyOS[9, 8]，SOS[68]，Contiki[61]，MANTIS[27]，protothreads[16] などが挙げられる．これらの技術の多くはイベントモデルで構築されているためスレッドモデルに比べてプログラムの記述が難しく，かつハードリアルタイム処理をサポートしていない．

TinyOS[9] は現在無線センサネットワークの研究分野で標準的に使用されているOS である．TinyOS は nesC[8] と呼ばれる C 言語に似たイベントドリブンに特化した言語を用いて構築されており，少ないメモリとオーバヘッドで複数のタスクを処理することができることを特徴としている．しかしながら，TinyOS は 3.2.1 で述べたようにプログラムの書き辛さとハードリアルタイム性の欠如の問題を持っている．さらに，ユーザは nesC の記述の仕方を学ばなければならないという問題点も持っている．

SOS[68] は TinyOS と同様に event model を用いて構築されている．TinyOS がモジュール性を持っていないのに対し，SOS は少ないオーバヘッドでプログラムモジュールの動的な追加や削除を行うことができる．また，タスクを C 言語で記述可能という特徴も備えている．しかしながら，SOS は TinyOS と同様に event model であるがゆえにハードリアルタイム処理をサポートしておらず，プログラムも書き辛い．

Contiki[61] は C 言語を用いて event model と thread model の両方を用いて構築されている．Contiki は各スレッドに対してスタックを割り当てることで time-sliced pre-emptive multithreading を実現している．そのため，PAVENET OS に比べてスレッドを実現する時のメモリ使用量が多く，ハードリアルタイム処理も実現できない．

protothreads[16] は event model におけるプログラムの書き辛さを軽減するために考案された仕組みである．event model では各タスクは run-to-completion で実現されるため，ユーザは何かを処理したい場合にその処理を複数のタスクに分割して記述しなければならない．protothreads では，event handler を途中で中断可能な仕組みを取り入れることで event handler の書きやすさを提供している．protothreads を用いることで event model の制御フローの把握し辛さは若干軽減されるものの，本質的な event model のプログラムの書き辛さは解決していない．さらに，ハードリアルタイム処理もサポートしていない．

MANTIS は PAVENET OS と同じく thread model のオペレーティングシステムである．MANTIS は time-sliced pre-emptive multithreading で実現されており，

各スレッドに対してスタックを割り当てなければならない．そのため，必要とされる計算資源がPAVENET OSに比べて多い．また，ハードリアルタイム処理も実現できない．

3.5.2 CPU

PAVENET OSはMicrochip社のPIC18の機能を積極的に利用することでハードリアルタイム処理を少ないオーバヘッドで実現している．PAVENET OSがPIC18で利用している機能は動的な優先度割り込みの機能とハードウェアレベルによるコンテキストスイッチのサポートである．PIC18のこれらの機能を具備しているCPUであればPAVENET OSは実現できる．

Intel 8051ではPIC18と同様に動的な優先度割り込みを提供しているため，PAVENET OSをAtmelやMaxim ICやフィリップスなどが提供するIntel 8051互換のCPU上に実装することができる．しかしその場合はハードリアルタイムタスクへのコンテキストスイッチの機能をソフトウェアで提供する必要がある．

ATMEL社のATmega128LやMSP430は優先度割り込みを提供しているが，優先度が固定的であるため，PAVENET OSは現在のままではこれらのCPU上に実装することはできない．PAVENET OSのco-operative multithreadingはATMEL社のATmega128LやMSP430でも実現可能であるが，ハードリアルタイム性の実現は割り込み毎の優先度チェックやレジスタの退避など全てソフトウェアでやらなければならないため，オーバヘッドが大きくなる．

3.5.3 階層化

無線センサノードにおける通信プロトコルの階層化に関する研究としては[17]が存在する．[17]では，TinyOS上で多様なプロトコルをサポートするためにL2とL3の間にSP(Sensornet Protocol)と呼ばれるプロトコルを提供している．本研究の目的はAPI内で排他制御を隠蔽することで各層の独立性を実現することであるので[17]と目的が異なる．そのため，SPとPAVENET OSの無線プロトコルスタックは共存可能であるので，今後SPをPAVENET OS上に実装することも検討する必要がある．

3.6 むすび

本章では，無線センサノード用のハードリアルタイムオペレーティングシステムであるPAVENET OSを示した．PAVENET OSはTinyOSと同じ程度の計算資源と性能で実現できるうえに，TinyOSが持っていないハードリアルタイム処理とプログラムの書きやすさを提供する．さらに，ハードリアルタイム処理を記述した場合でもオペレーティングシステムとしてレイヤ間の排他制御を隠蔽する機能を提供することでユーザは各層の独立に開発することができる．

我々は今後，PAVENET OSを無線センサネットワークの分野で普及させたいと考えているが，それにはいくつかの敷居がある．まず，TinyOSがすでに広く出回っており，シミュレータなどの開発ツールが充実していることである．これらのツールをPAVENET OSでも開発していく必要がある．また，PAVENET OSがPIC18の機能に特化して設計していることも普及の妨げになると考えられる．PAVENET OSは移植は可能であるが，移植した場合にはPIC18が持っていて移植先のCPUが持っていない機能をソフトウェアで実現する必要が出てくるため，実行時のオーバヘッドが大きくなる．さらに，PIC18用の使いやすいフリーのコンパイラが存在しないことも普及の妨げになると考えられる．現在SDCC(Small Device C Complier)と呼ばれるGPLで提供されているコンパイラが開発されているが，まだ使えるレベルに達していない．筆者らは，われわれの研究の中でPAVENET OSを改良しながらこれらの問題点を解決し，普及させて行きたいと考えている．