NightStar RT Tutorial

Version 4.6

(RedHawk™ Linux®)

concurrent

0898009-150
July 2018

本書は当社製品を利用する社員、顧客、エンドユーザーを対象とします。
本書に含まれる情報は、本書発行時点での正確な情報ですが、予告なく変更されることがあります。
当社は、明示的、暗示的に関わらず本書に含まれる情報に対して保障できかねます。

注意事項：

本書は、Concurrent Real-Time, Inc.より発行された「NightStar RT Tutorial」を日本語に翻訳した資料です。英文と表現が異なる文章については英文の内容が優先されます。

前書き

一般事項

NightStar RT™は、RedHawkを実行しているユーザーがクリティカルなアプリケーションだけでなくオペレーティング・システム・カーネルの実行時の挙動をスケジュール、監視、デバッグ、解析することを可能にします。

NightStar RTはNightTrace™イベント・アナライザー、NightProbe™データ監視ツール、NightView™シンボリック・デバッガー、NightSim™スケジューラー、NightTuneTMシステム/アプリケーション・チューナー、データ監視API、Shmdefine共有メモリ・ユーティリティで構成されます。

マニュアルの範囲

本説明書は、NightStar RT のチュートリアルとなります。

マニュアルの構成

本説明書は NightStar RT のチュートリアルを含む 7 つの章と巻末付録で構成されます。

構文記法

以下の表記法が本書を通して使用されます：

斜体

ユーザーが特定すべき書籍、参照カード、参照項目は、*斜体*にて表記します。特殊用語やコメントも*斜体*にて表記します。

太字

ユーザー入力は**太字**で表記され、指示されたとおりに入力する必要があります。ディレクトリ名、ファイル名、コマンド、オプション、man ページの引用も**太字**形式にて表記します。

`list`

プロンプト、メッセージ、ファイルやプログラムのリストのようなオペレーティング・システムおよびプログラムの出力は `list` 形式にて表記します。キーワードについても `list` 形式にて表記します

<u>強調</u>

特別に注目する必要のある単語または語句は強調形式を使用します。

window

キーボードの並びやプッシュ・ボタン、ラジオ・ボタン、メニュー項目、ラベル、タイトルのようなウィンドウ機能は window 形式で表記します。

[ ]

大括弧はコマンド・オプションやオプションの引数を囲みます。そのようなオプションや引数の指定を望む場合、括弧は入力しないで下さい。

{ }

中括弧はパイプ記号(|)で分離された相互排他的な選択を囲み、1 つの選択を選ぶ必要があります。選択したものと一緒に中括弧もしくはパイプ記号は入力しないで下さい。

. . .

省略は反復される項目が後に続きます。

: :=

本記号は Backus-Naur Form (BNF)として定義されていることを意味します。

参考文献

次の図書を本書の中で参照しています：

0898395	*NightView™ User's Guide*
0898398	*NightTrace™ User's Guide*
0898465	*NightProbe™ User's Guide*
0898480	*NightSim™ User's Guide*
0898515	*NightTune™ User's Guide*

目次

1 章 概要

2 章 パネル

3 章 NightView の利用

4 章 NightTrace の利用

5 章 NightProbe の利用

6 章 NightTune の利用

7 章 NightSim の利用

チュートリアル・ファイル

図

1

概要

NightStar RT™ は、タイム・クリティカルな Linux®アプリケーションを開発するためのデバッグ・ツール一式が統合されています。NightStar RT はお手持ちのアプリケーションに対して最小限の干渉、実行動作の保護、デターミニズムとなるように設計されています。ユーザーは直ぐにそして容易にアプリケーションのデバッグ、監視、解析、チューニングが可能となります。

NightStar RT ツールは次で構成されます :

- NightView™ ソースレベル・デバッガー
- NightTrace™ イベント・アナライザー
- NightProbe™ データ・モニター
- NightTune™ システム/アプリケーション・チューナー
- NightSim™ スケジューラー

本チュートリアルにおいて、これらのツールを広範囲に及ぶ機能性をデモする様々な状況を具体化して 1 つのまとまった実例に統合しています。

はじめに

本チュートリアル内のある作業は、デフォルトでユーザー・アカウントに許可されていない高度なユーザー権限を必要とします。本チュートリアルで示しているように root ユーザーとして実行、もしくは次項の「ユーザー権限の設定」で詳述されているように適切な権限を取得する必要があります。

ユーザー権限の設定

Linux は、ある権限を持った操作を実行するための特権を権限のないユーザーに別の方法で許可する手段を提供します。**pam_capability(8)** (Pluggable Authentication Modul)は、様々な作業で要求されるケーパビリティ(ロールの呼び出し)一式を管理するために使用されます。

Linux システムは NightStar RT が要求とするケーパビリティを提供する nightstar ロールを構成する必要があります。NightStar RT 機能の全ての優位性を得るには、各ユーザーが(少なくとも)以下に示すケーパビリティを使用するように構成する必要があります。

(定義されていない場合は)**/etc/security/capability.conf** を編集し「ROLES」セクションに nightstar ロールを定義して下さい：

```
role nightstar cap_sys_nice cap_ipc_lock
```

更にターゲット・システムの各 NightStar RT ユーザー用にファイルの最後の行に以下を追加して下さい：

```
user  username   nightstar
```

username はユーザーのログイン名となります。

ユーザーが nightstar ロールで定義されていないケーパビリティを必要とする場合、nightstar と必要とする追加のケーパビリティを含む新しいロールを追加し、上記のテキストの nightstar を新しいロール名称に置き換えて下さい。

/etc/security/capability.conf へのログイン名の登録に加えて、**/etc/pam.d** ディレクトリ以下のファイルも有効にするためにケーパビリティを許可する構成にする必要があります。

ケーパビリティを有効にするには、**/etc/pam.d** 以下の選択したファイルの最後に(存在しない場合は)以下の行を追加して下さい：

```
session required pam_capability.so
```

修正するファイルのリストは、システムにアクセスするために使用されるメソッドのリストに依存します。次の表は、システムへのアクセスに用いられる最も一般的なサービスについてユーザーにケーパビリティを許可する推奨構成を示します。

表 1-1. 推奨する/etc/pam.d 構成

/etc/pam.d ファイル	影響するサービス	コメント
remote	telnet rlogin rsh（コマンド無しで使用の場合）	お手持ちのシステムに依存、**remote** ファイルは存在しないかもしれません。 **remote** ファイルは生成しないで下さい。但し存在する場合は編集可能です。
login	local login (e.g. console) telnet* rlogin* rsh*（コマンド無しで使用の場合）	*Linux の一部のバージョンでは、**remote** ファイルの存在は **login** ファイルの範囲をローカル・ログインに制限します。そのような状況では、**login** と共にここに記述されている他のサービスは **remote** 構成ファイルのみに影響します。
password-auth	多くのサービス	**最新の** Linux システムにおいては、本ファイルも修正する必要があります。これが存在する場合、本ファイルに対しても全頁の行を追加を追加して下さい。
rsh	rsh（コマンド無しで使用の場合）	e.g. **rsh system_name a.out**
sshd	ssh	**/etc/ssh/sshd_config** を編集し次の行が存在することを確認する必要があります：**UsePrivilegeSeparation no**
gdm **gdm-password**	gnome セッション	一部のシステムにおいては **gdm-password** が存在する場合は修正する必要があります。
kde	kde セッション	

/etc/pam.d/sshd もしくは**/etc/ssh/sshd_config** を修正する場合、変更を有効にするため **sshd** サービスを再開する必要があります：

```
service sshd restart
```

適切に上記変更を有効にするには、ユーザーはログ・オフしターゲット・システムに再ログ・インする必要があります。

NOTE

許可したケーパビリティを確認するには、次のコマンドを実行して下さい：

```
/usr/sbin/getpcaps $$
```

上記コマンドの出力は現在割り当てられているロールをリストアップします。

チュートリアル・ディレクトリの生成

全ての作業を行うディレクトリを生成することから始めます。ディレクトリを生成しそこに移動して下さい：

- 作業ディレクトリを生成するには **mkdir(1)** コマンドを使用して下さい。

  次のコマンドを使ってディレクトリ名を **tutorial** とします：

  ```
  mkdir tutorial
  ```

- **cd(1)** コマンドを使って新しく生成したディレクトリに移動して下さい：

  ```
  cd tutorial
  ```

様々なツールのためのソース・ファイルおよび構成ファイルは NightStar RT のインストール中に **/usr/lib/NightStar/tutorial** へコピーされます。これらのチュートリアルに関連するファイルを **tutorial** ディレクトリにコピーします。

- 全てのチュートリアル関連ファイルをローカル・ディレクトリにコピーして下さい：

  ```
  cp /usr/lib/NightStar/tutorial/* .
  ```

プログラムのビルド

この例では周期的なマルチスレッド・プログラムを使用し、各サイクルで様々なタスクを実行します。サイクルは設定可能なレートを持ったタイムアウトを使用するメイン・スレッドによって制御されます。

メイン・ソース・ファイル **app.c** の一部を以下に示します：

```
int
main (int argc, char * argv[])
{
    pthread_t thread;
    pthread_attr_t attr;
    struct sembuf trigger = {0, 2, 0};
    nosighup();

    trace_begin ("/tmp/data",NULL);

    sema = semget (IPC_PRIVATE, 1, IPC_CREAT+0666);

    pthread_attr_init(&attr);
    pthread_create (&thread, &attr, watchdog_thread, NULL);

    pthread_attr_init(&attr);
    pthread_create (&thread, &attr, sine_thread, &data[0]);

    pthread_attr_init(&attr);
    pthread_create (&thread, &attr, cosine_thread, &data[1]);
```

```
    pthread_attr_init(&attr);
    pthread_create (&thread, &attr, heap_thread, NULL);

    for (;;) {
        struct timespec delay = { 0, rate } ;
        nanosleep(&delay,NULL);
        work(random() % 1000);
        if (state != hold) semop(sema,&trigger,1);
    }

    trace_end () ;
}
```

プログラムは 4 つのスレッド生成した後に共通のタイムアウトに基づき 2 つのスレッドをそれぞれ周期的にアクティブにするループに入ります。3 つ目および 4 つ目のスレッド(heap_thread および watchdog_thread)は非同期的に実行します。

To build the executable

ローカル・ディレクトリ **tutorial** から以下のコマンドを入力して下さい：

```
cc -g -o app app.c -lntrace_thr -lpthread -lm –lrt
```

NOTE

NightStar RT ツールは、ユーザー・アプリケーション・プログラム・ファイルからシンボル・テーブル情報を読むためにユーザー・アプリケーションが DWARF デバッグ情報を伴ってビルドされていることを必要とします。従って、**-g** コンパイル・オプションを指定します。しかしながら、ツールはシンボルなしでプログラムをデバッグするために機能を減らして使用することも可能です。

2

パネル

NightStar は、サイズ変更可能かつ移動可能なパネルの利用を通してユーザーのニーズを満たすために柔軟に構成するグラフィカル・ユーザー・インターフェースを提供します。

本章ではパネルの移動およびサイズ変更に関わる概念を示します。これは単に参考用に考案されたもので、段階的な指導ガイドではありません。

3-1 ページの「NightView の利用」で習うツールを使用するための初期段階へ進む前に本章を読んで下さい。

パネルの移動

パネル内に List ビューと Graph ビューがそれぞれ含まれている以下の NightProbe を見て下さい：

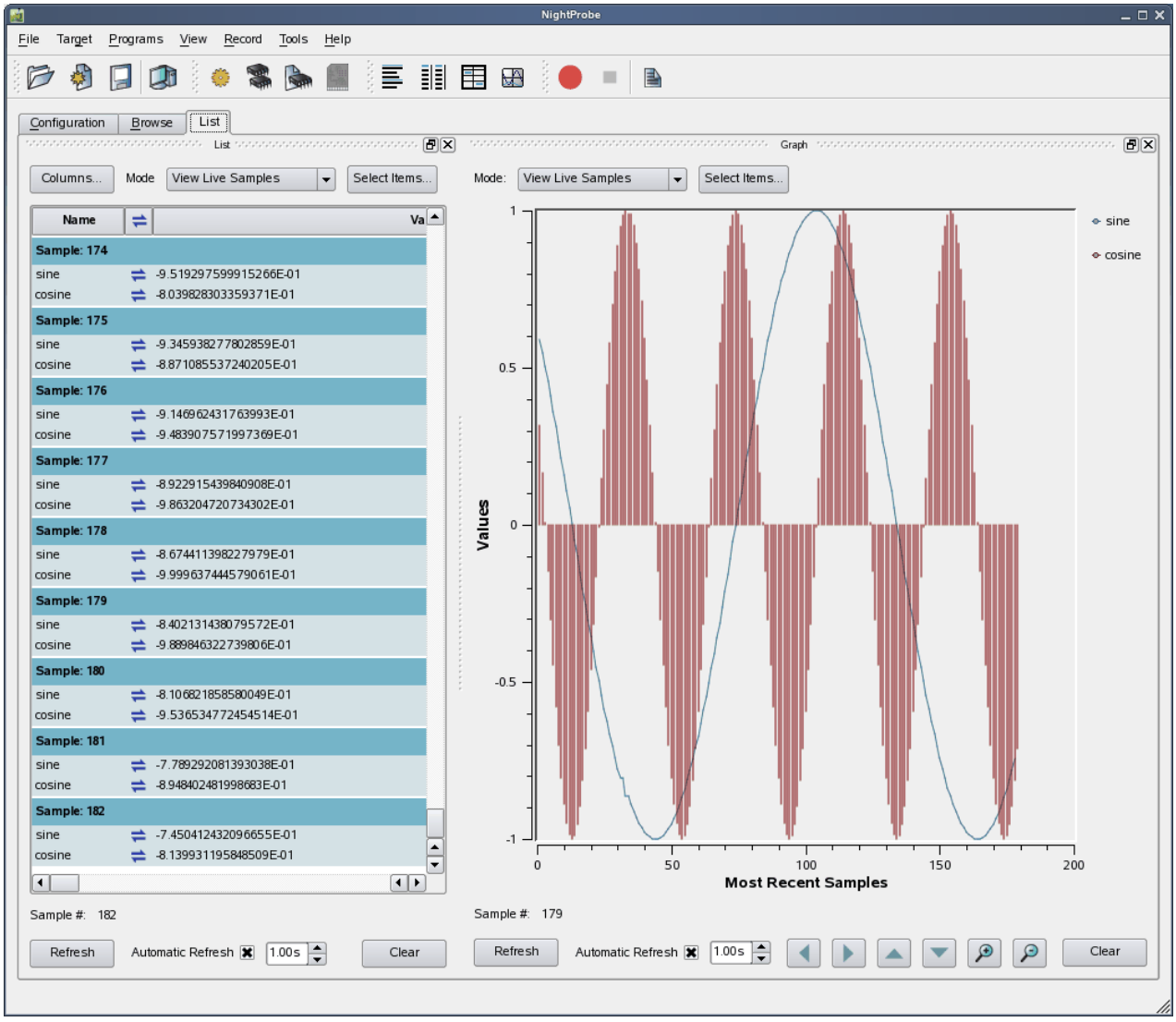


図 2-1. List と Graph パネルが一体になったページの表示

パネルはタイトル・バーを左クリックし、新しい場所へそれをドラッグした後、マウス・ボタンを解放することで移動します。マウス・ボタンを解放した時のパネルの位置によっては、パネルは切り離されたままとなる、もしくはページに再度はめ込まれます。

はめ込まずにページからパネルを切り離すには、パネル右手上部の一番左側にあるコントロール・ボックスをクリックして下さい。

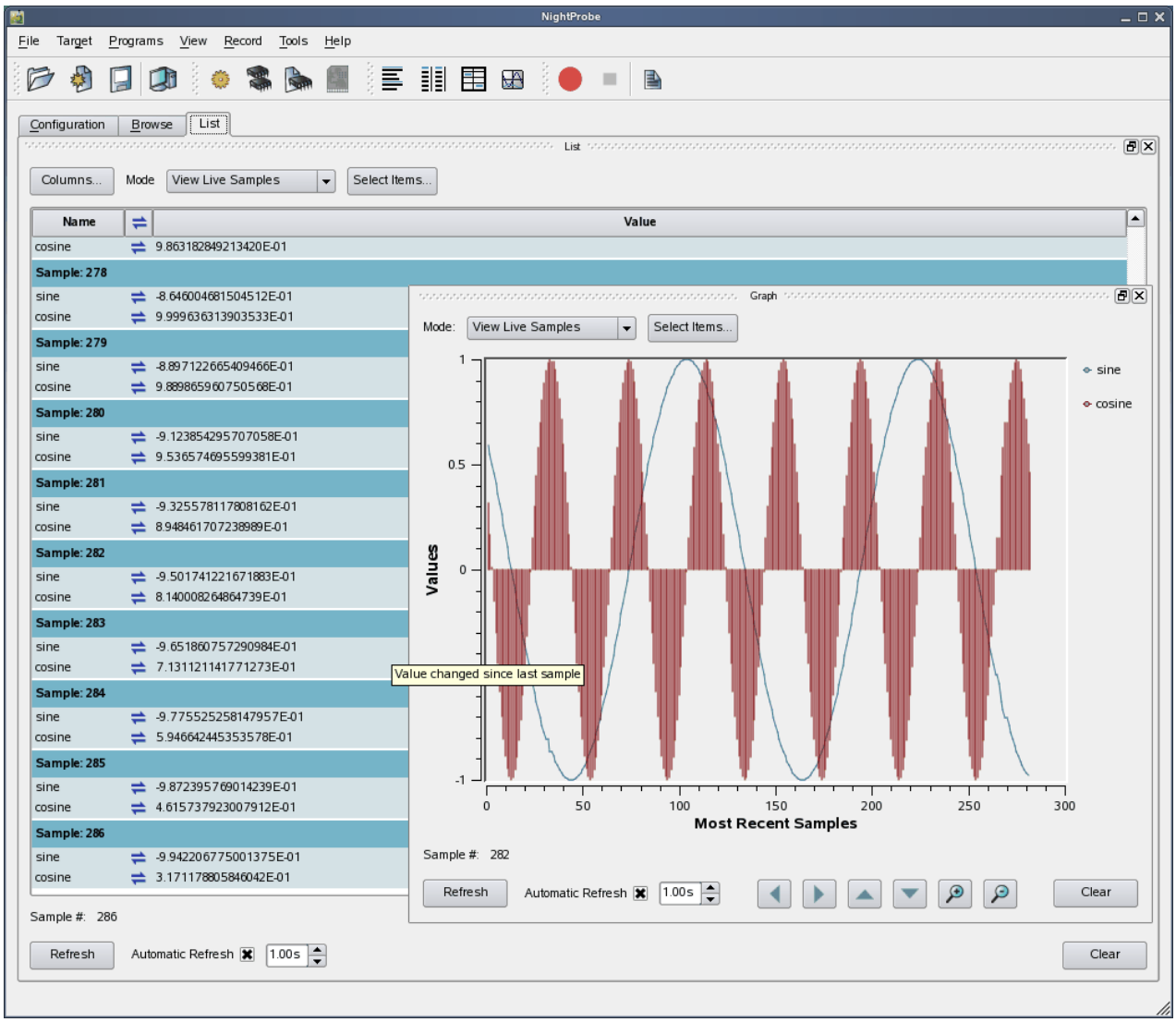


図 2-2. ページから切り離されたパネル

Graph パネルはページから切り離されて自由に動くようになります。

メイン・ウィンドウの境界線の外に移動して解放された場合、パネルはメイン・ウィンドウから切り離されたままとなります。一方、切り離されたモードであっても、メイン・ウィンドウがアイコン化された場合、切り離されたパネルも一緒にアイコン化されます。そのため、切り離されたパネル自体はあまり有用ではありません。切り離しは殆どの場合はパネルを移動して再ドッキングする時に有用です。

ページの新しい場所にパネルを挿入する場合、そのタイトル・バー上を左マウス・ボタンを使いパネルをドラッグしてページの境界に近づくまで移動して下さい。ウィンドウはパネルが挿入される場所を示す空間を生成することで応答します。

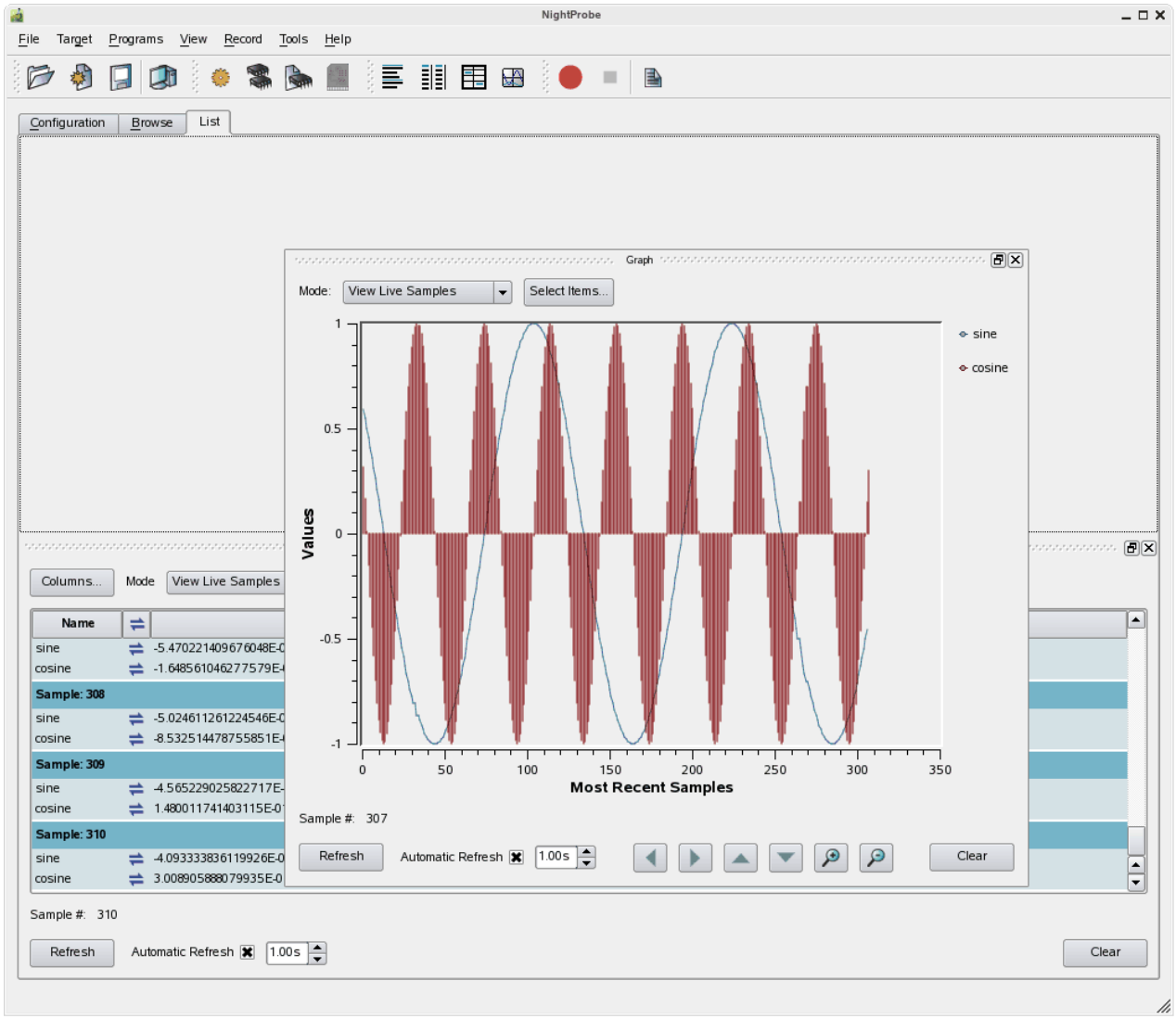


図 2-3. 移動途中のパネル

上図は、ページの水平境界の上部に向かって Graph パネルがドラッグされたため、空間が List パネルの上に生成されていることを示しています。

この時点でマウス・ボタンを解放すると Graph パネルがページ内に挿入され、直近で生成された空間が消費されます。

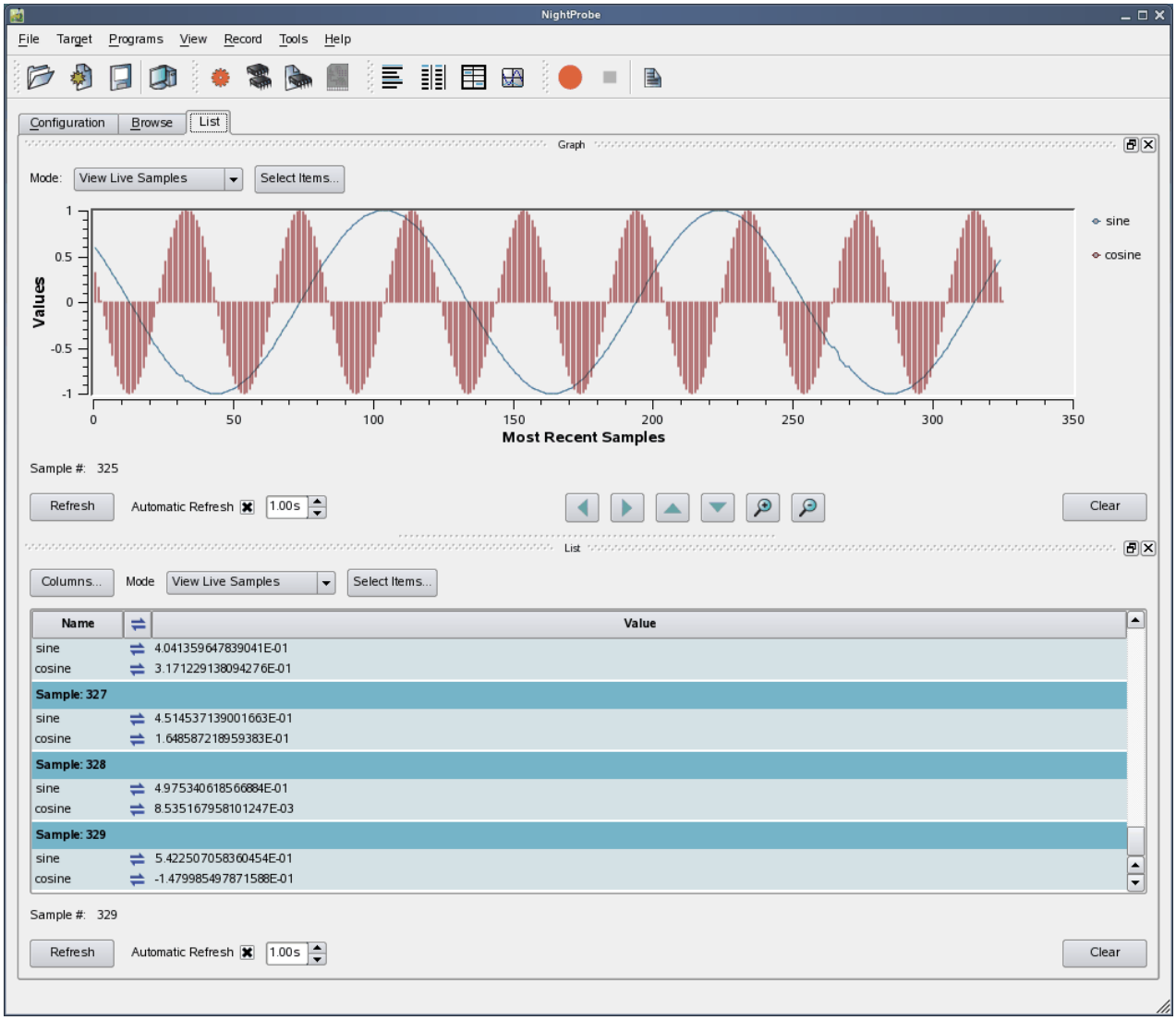


図 2-4. List パネルの上にある Graph パネル

IMPORTANT

ページ内にパネルを移動しようとした際、希望する場所に空間が表示されない場合は、メイン・ウィンドウのサイズを拡大、ドッキングしていないパネルのサイズを縮小、置きたい場所付近の別のドッキングしていないパネルの縁の移動を試して下さい。

デフォルトで、新しいパネルが生成された時にツールは通常ページの右側にパネルを追加します。

次の図においては、Table パネルが Graph と List パネルの右側に追加されました。

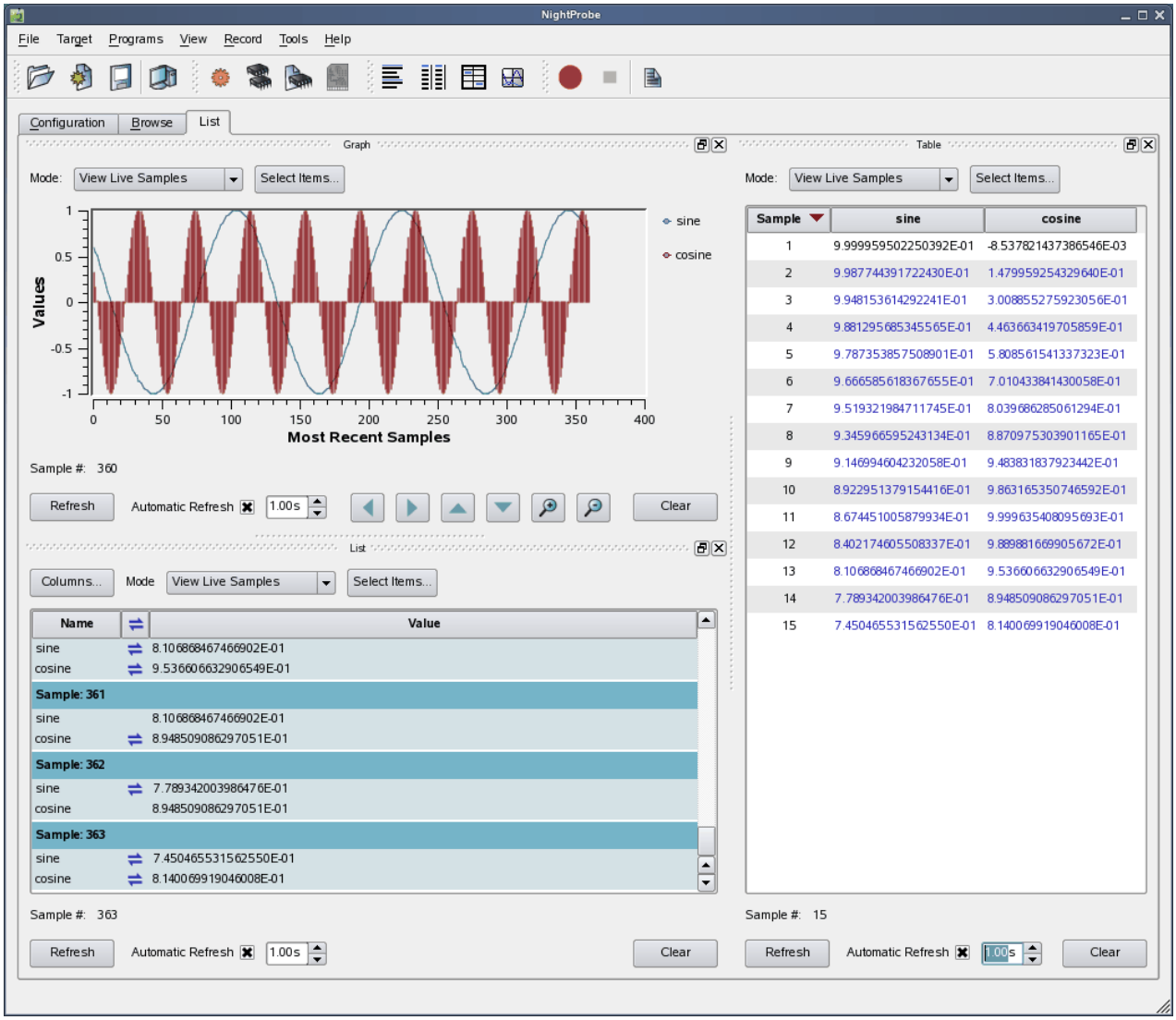


図 2-5. Table ビューが追加されたページ

パネル間のセパレータを左クリックして必要なサイズになるまでドラッグすることでパネルのサイズを変更することが可能となります。

タブ化パネル

グラフィカル・ユーザー・インターフェースのもう一つの特長はタブ化パネルの使用です。タブ化パネルは、2 つ以上のパネルを同じ場所にお互いを積み重ねて置くことで GUI 領域を最大化することを可能にします。その後、タブをクリックすることで最上位にパネルを上昇させることが可能です。

タブ化パネルを生成するには、ページに結合しているままのパネルの下部にタブが表示されるまで他のパネルの水平境界下側にパネルを移動して下さい。

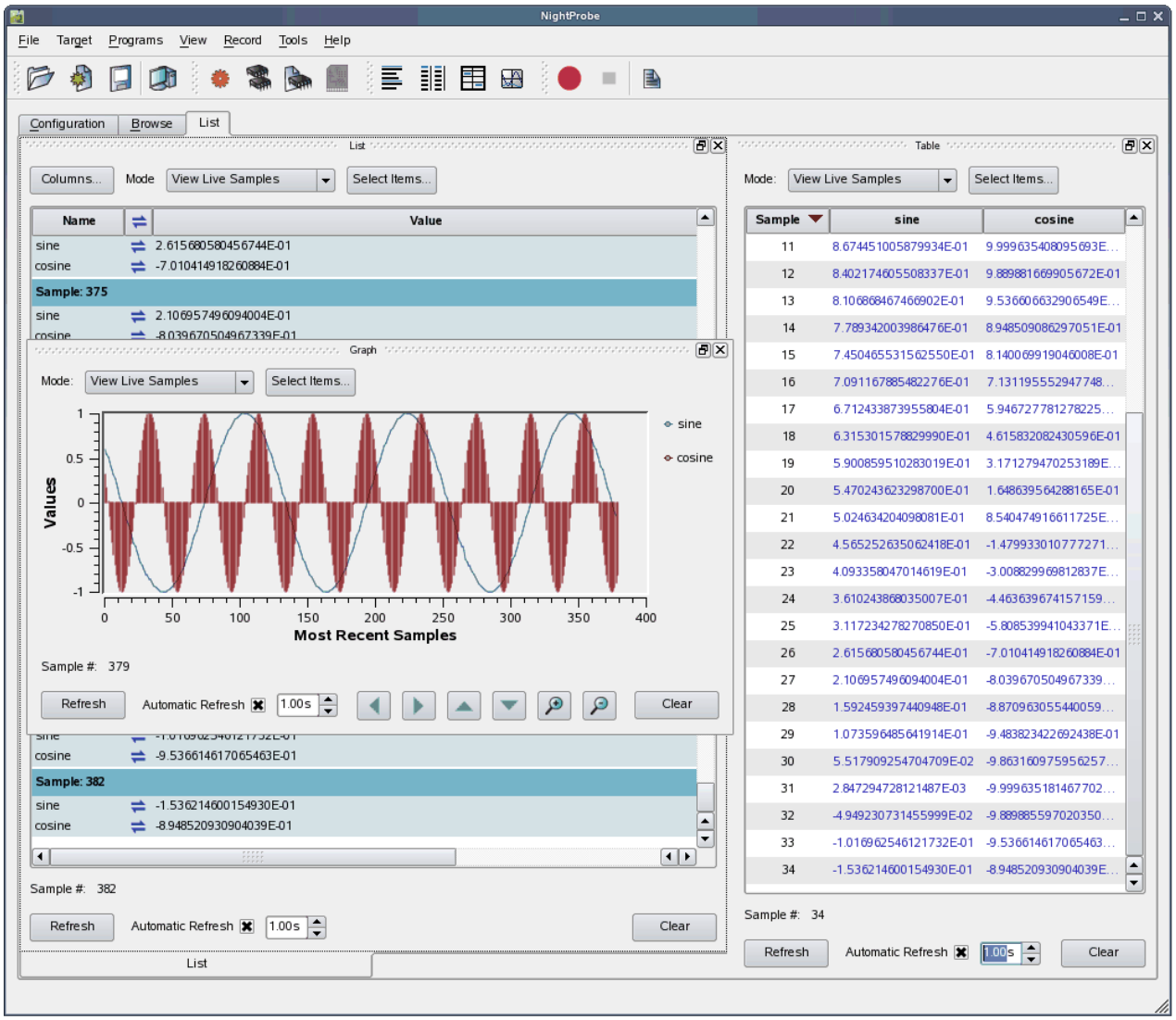
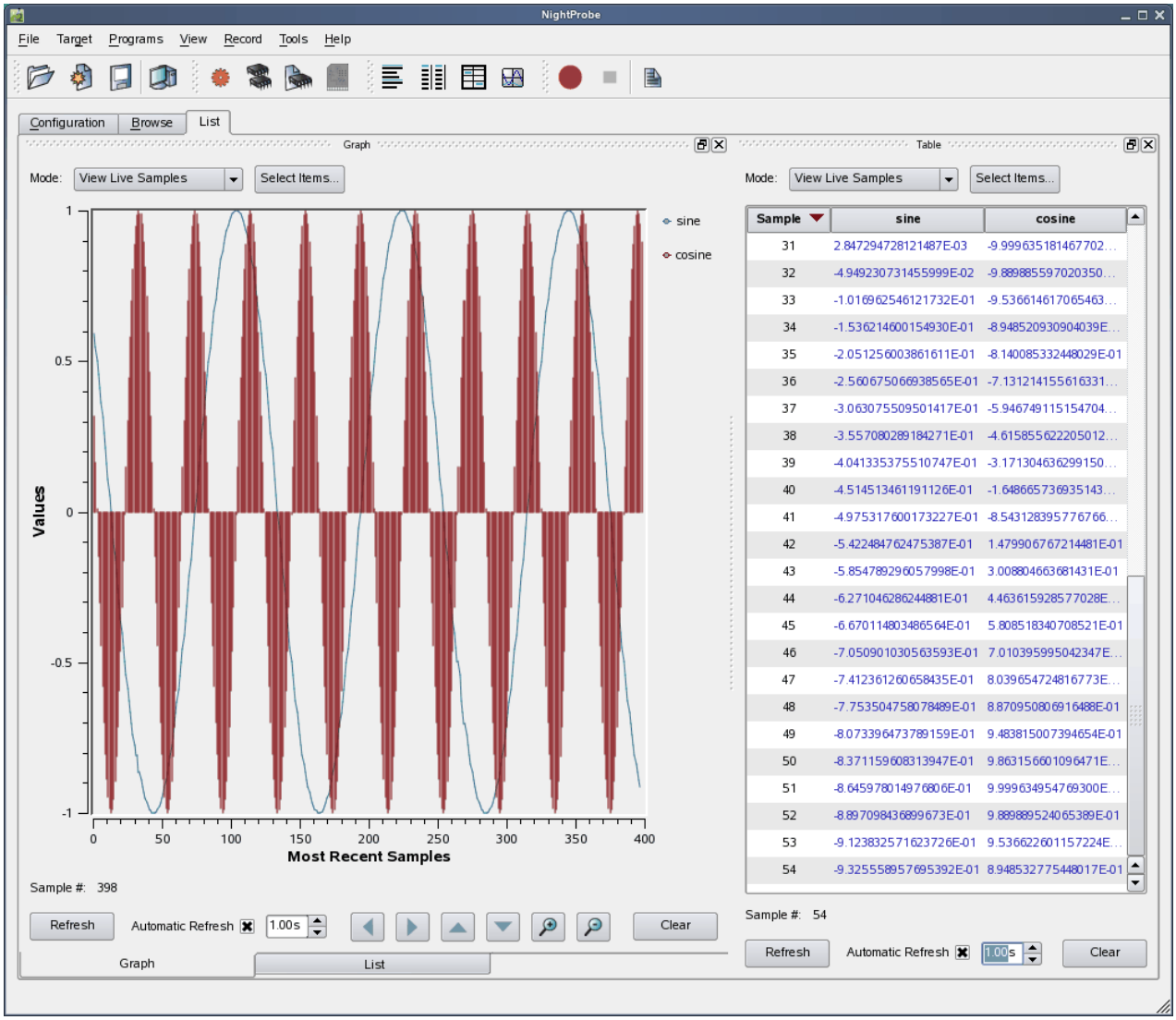


図 2-6. パネルを移動してタブを生成

上図は、Graph パネルを List パネル上の元の位置から List パネルの下部に向かってドラッグしているところです。マウス・ボタンを解放した場合、Graph および List パネルはタブ化されるためにページの同じ領域を消費する事を示すタブが List パネル上に現れます。



IMPORTANT

パネルを他のパネル上に移動するには、望むパネルを他のパネルの上端に移動して下さい。パネルを他のパネルの下端に移動した場合、代わりにタブ化パネルになります。

コンテキスト・メニュー

NightStar ツールはコンテキスト・メニューの使用に非常に依存しています。

コンテキスト・メニューは、マウス・カーソルが興味のある領域もしくはアイテムの上に置かれた時にマウスを右クリックすると現れるメニューです。それらの内容が右クリックした領域もしくは右クリックしたアイテムのコンテキストに多くの場合は依存するため、これらはコンテキスト・メニューと呼ばれます。

不確かな場合、右クリックの操作を試してメニューが利用可能になるかどうかを確認して下さい。

チュートリアルのスクリーン・ショット

本チュートリアルではフル・スクリーン・ショットを見せるため、各メイン・ウィンドウのサイズを大抵はデフォルト設定のままにしています。出力されたページの利用可能な空間内にイメージを合わせるため、大きなウィンドウの表示は圧縮が必要となります(このような圧縮は細部を不明瞭にします)。

しかしながら、このチュートリアルのユーザーであれば、メイン・ウィンドウのサイズを拡大する事を強く推奨します。そうすることで個々のパネルの内容をスクロールする必要なく多くのデータを確認することが可能となります。

本チュートリアル内の多くのケースでは、スクリーンの拡大された領域の一部はメイン・ウィンドウから抜粋され、単独のスクリーン・ショットとして含まれています。これらは各ツールのメイン・ウィンドウ内のパネルが相当します。

3

NightView の利用

NightView は特にタイムクリティカル・アプリケーション用に設計されたグラフィカルなソースレベルでのデバッグおよび監視するためのツールです。NightView は最小限の干渉により複数のプロセッサ上で実行中の複数のプロセスを監視、デバッグ、パッチを当てることが可能です。

NightView は以下を含む標準的なデバッガに求める全ての特長をサポートします：

- ブレークポイント
- 命令文を通り抜けて 1 行ずつ実行
- 関数呼び出しを飛び越えて 1 行ずつ実行
- 完全な抽象表現による解析
- ブレークポイント用の条件および無視カウント
- ハードウェアを支援するアドレス・トラップ(ウォッチポイント)
- アセンブリおよびシンボリック・デバッグ

標準的なデバッグ機能に加えて、NightView は次の機能も提供します：

- アプリケーション速度でイベントポイントの条件付け
- プログラム実行中にプログラムのフローを変えるためにパッチを適用、またはメモリもしくはレジスタを変更する機能
- ホット・パッチおよびイベントポイントの制御
- 同じ速度でデータの監視
- ロード可能なモジュール
- マルチスレッド化プログラムのサポート
- 複数プロセスのデバッグ
- 動的メモリのデバッグ
- 関連するリストを横断
- メモリのセグメントを検索
- 分岐履歴
- スレッド単位のデバッグ(保護、単一スレッド、複数スレッド)
- スマート表示(不明慮または複雑な構造で表示されている箇所をカスタマイズ)

NightView の起動

- 次のコマンドをコマンド・プロンプトで入力して NightView を実行して下さい：

```
nview &
```

またはデスクトップ上のアイコンをダブルクリックして下さい。

NOTE

NightStar ツール用のデスクトップ・アイコンがない場合は、**/usr/lib/NightStar/bin/install_icons** を実行して下さい。

NightView を起動すると NightView メイン・ウィンドウが現れます。

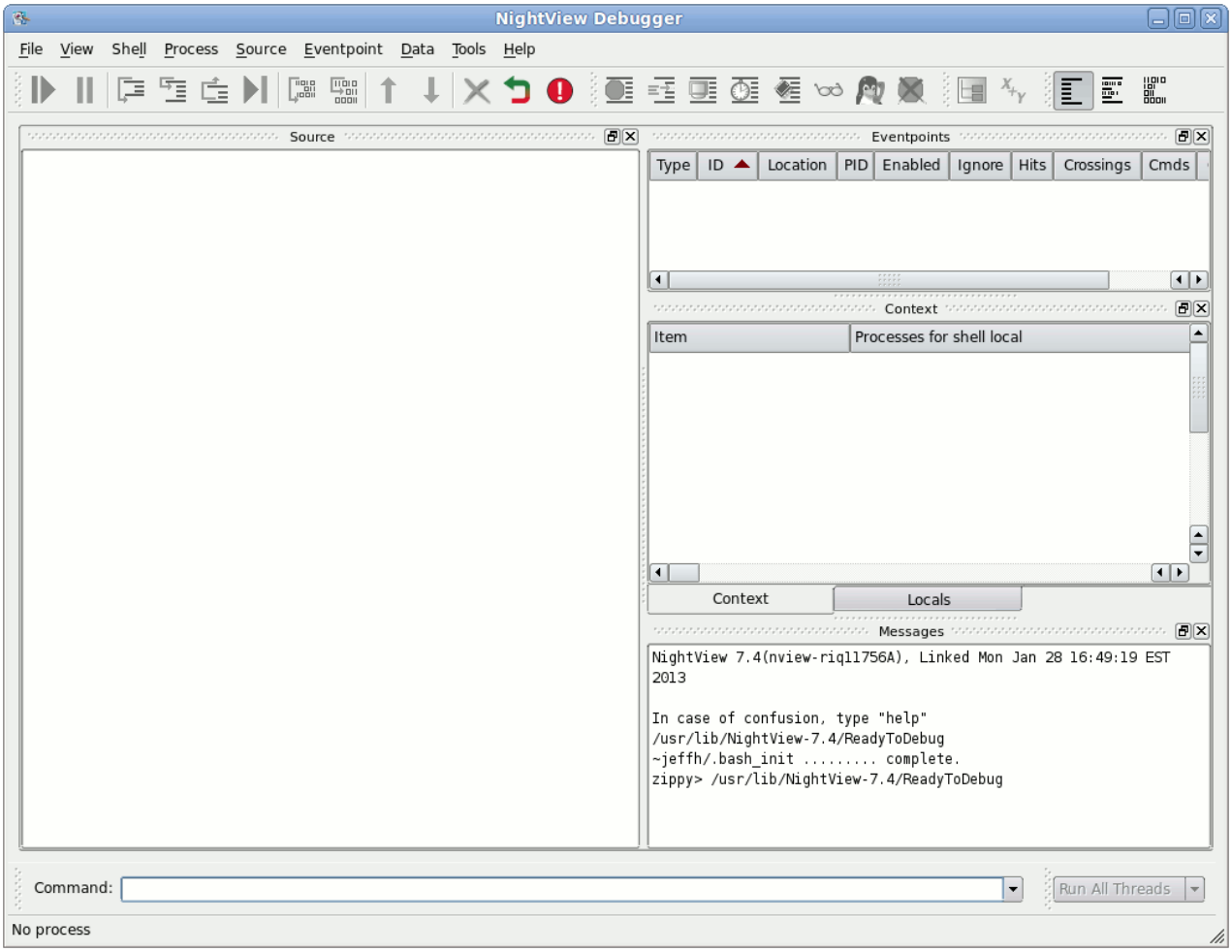


図 3-1. NightView メイン・ウィンドウ

NOTE

NightStar をインストールまたは最新のバージョンにアップグレードしてから NightView を起動するのが初めての場合、初期画面を見ることになるかもしれません。その画面の下部左端のチェックボックスを使ってその後の起動において初期画面を無効にすることが可能です。

画面が現れる場合、続行するには NightView ボタンを押下して下さい。以前に NightStar を使っておりその構成をカスタマイズしている場合、本チュートリアルを使用する間は混乱を回避するためデフォルトの構成をロードしたほうが良いでしょう。File メニューから Load System Default Configuration を選択することでそれが可能です。

本書の例では、シングル・アプリケーションをデバッグします。

NOTE

app プログラムを生成していない場合は、1-4 ページの「プログラムのビルド」を参照して下さい。

- Process メニューから Run...を選択し Run on local ダイアログのテキスト・フィールドに以下を入力して NightView メイン・ウィンドウで本チュートリアル・アプリケーションを起動して下さい：

```
./app
```

- ダイアログを閉じるには OK を押下してプログラムを実行して下さい。

プログラムが生成するどの出力も Messages パネルに表示します。

実行のために **app** プログラムがロードされた時、NightView は動的リンクの初期段階が終了した直後(但し、一部の静的コンストラクター・コードが実行される前)にプログラムを停止し、

ソース・パネルに main 関数のソースコードを表示します。

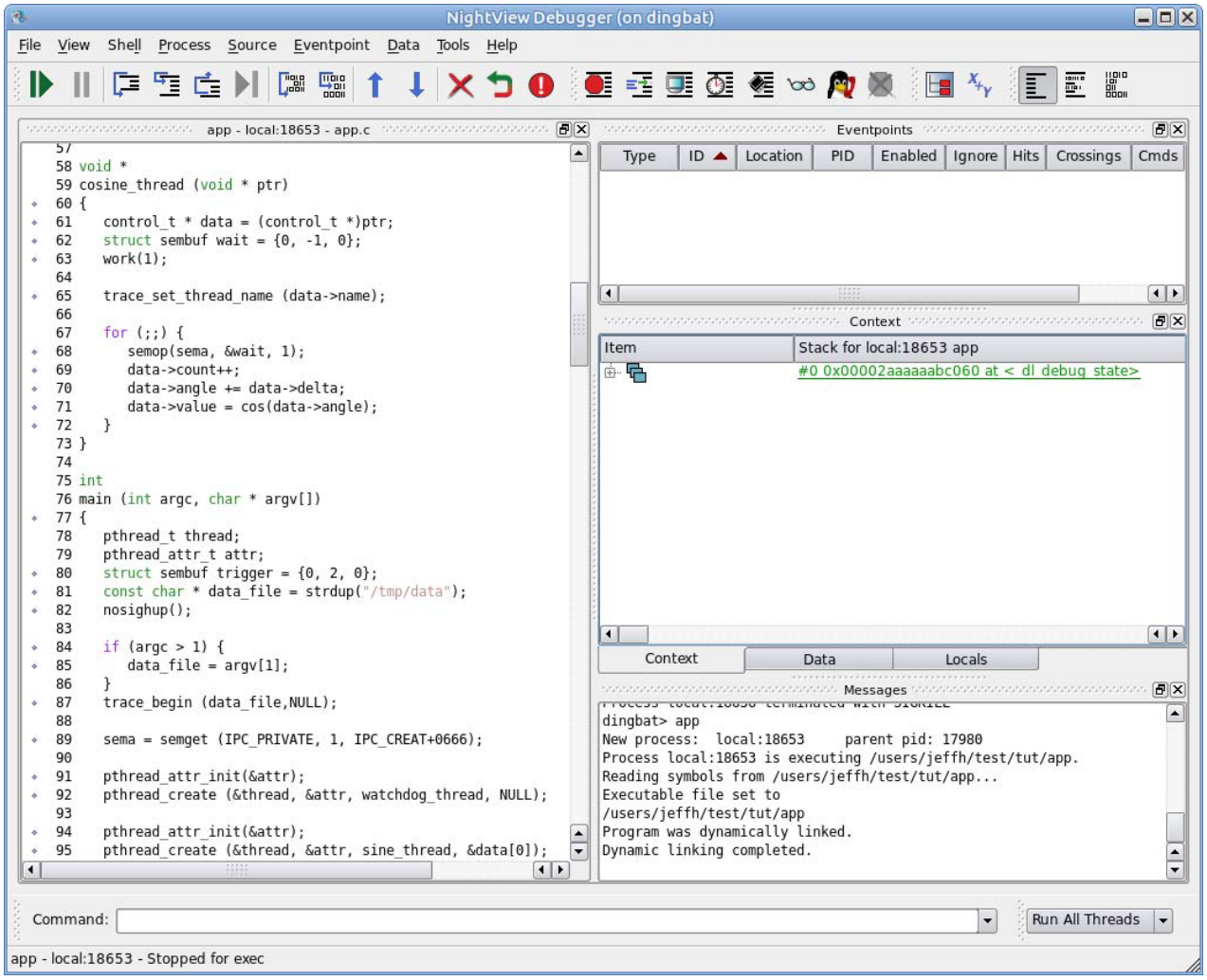


図 3-2. ロードされた app プログラム

NightView は複数プロセスだけでなくシングルおよびマルチスレッドのプログラムのデバッグもサポートします。本チュートリアルでは、複数のスレッドを伴うシングル・プロセスをデバッグします。

複数スレッドのデバッグ

当該アプリケーションは main スレッドと main スレッドに生成される 4 つの追加スレッドで構成されます。

- 以下のコマンドを入力してプロセスを起動して下さい：

```
resume
```

2 番目のスレッド(watchdog_thread)はクリティカルなスレッドで 50ms のデッドラインを守ってスムーズに実行する必要があります。さもなければ機能しなくなります。

NOTE

> watchdog スレッドがメッセージ・パネルにオーバーランのメッセージを断続的に出力していて、かつ特権を持つユーザー(1-2 ページの「ユーザー権限の設定」を参照)または root ユーザーではない場合、それは SCHED_FIFO へスケジューリング・クラスに設定できないためです。特権ユーザーまたは root ユーザーとして実行することを検討して下さい。

このスレッドを protected としてマークするので、他のスレッドが停止した時に NightView はこのスレッドを停止しません。

- Context パネルの watchdog_thread を右クリックして Protected Thread チェックボックスをクリックして下さい。

コンテキスト・パネルは watchdog_thread 行に単語 *protected* を表示することで応答します。実際には、protected 属性は NightView が保持する特別なスレッド属性です。

全ての NightView スレッド属性の値がスレッドのリスト内に表示されます。スレッド属性はユーザーにとって有用となります。詳細については *NightView User's Guide* の Concepts 章にある Thread Tags を参照して下さい。

後述のいくつかのセクションにおいて、プロセスを停止させるブレークポイントや他の手法を使用します。

- 以下のコマンドを実行して 52 行目にブレークポイントを設定して下さい：

```
b 52
```

プロセスはスレッドの 1 つが 52 行目のブレークポイントに到達するまで実行されますが、watchdog スレッドは保護されているので停止せず、全ての非保護スレッドは NightView により停止されます。

- Context パネルを表示するため Context タブをクリックして下さい。
- 緑色に表示されたスレッドを展開して下さい。

- 結果の最終段に現れた walkback リストの最初の項目を展開して下さい。

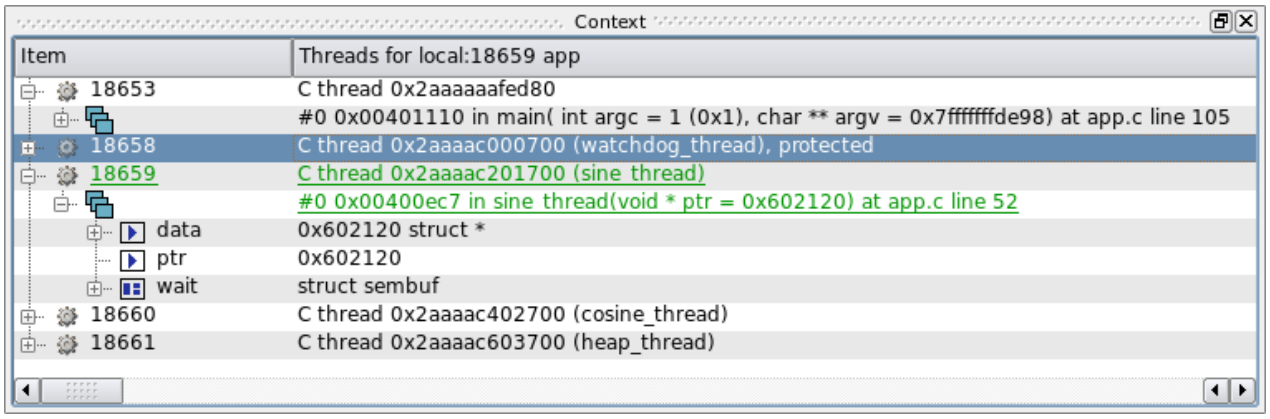


図 3-3. スタック・フレームが展開されたコンテキスト・パネル

walkback リストの個々の Frame を展開するとそのフレームに関する全てのローカル変数を表示します。複合変数および複合変数へのポインタを更に展開することが可能です。

Source パネルに表示されているソースは、プロセスを停止させたスレッドのプログラム・カウンタに関連したものです。丸括弧内のスレッドの開始ルーチンの名称を見ることで停止したスレッドを知ることが可能です。NightView は **pthread_create(2)** に渡された開始ルーチンに基づきスレッドに名称を自動的に割り当てます。更に **set-thread-name** コマンドを使って NightView の内部でスレッドの名称を設定することが可能です。

目的のスレッドをクリックすることで他のスレッドのコンテキストに切り替えることが可能です。スレッドをクリックすると NightView メイン・ウィンドウに表示されるソースはそのスレッドを実行している場所に変わります。

あるいは、**select-context** コマンドを使って Threads ディスプレイに表示されているスレッド名称もしくは **info threads** コマンドの出力から指定することも可能です：

```
info threads /v
select-context name="cosine_thread"
```

スレッド名称がスレッド間で一意ではない場合、常に一意であるスレッド ID を使用することが可能です。スレッド ID はスレッドを表す 16 進数で、スレッド生成中にスレッド・ライブラリにより割り当てわれます。スレッド ID は Context パネル内の各スレッド項目上にある単語「C thread」の直ぐ後に続きます。

- クリックして実行中のスレッド sine_thread() のコンテキストに切り替えてください。

NightView はスレッドを再開および停止させる方法を指定する Run Mode を提供します。デフォルトで Run Mode は Run All Threads です。従って、アプリケーションがブレークポイントに達する、または別の方法で NightView に停止された場合、アプリケーション内の(非 protected の)全てのスレッドも同様に停止し、NightView がスレッドの実行を再開した場合、全スレッドが実行を再開します。

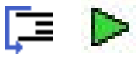
Run Mode を Run One Thread に変更した場合、その後スレッドを再開すると実行されるのは 1 つのみとなります。停止している他の全てのスレッドは停止したままとなります。

ツールバーの 1 つで現在の実行モードを表すオプション・リストを見ることが出来ます。デフォルトで本項目は画面下部の Command 領域の右側にあります。

図 3-4. Run モード・セレクター

- リストをクリックし Run One Thread モードを選択してモードを変更して下さい。

- 緑の PC アイコンが 51 行目(semop()の呼び出し)で停止するまで Next アイコンを複数回クリックして下さい。



- Next アイコンをもう 1 回クリックして下さい。

Next の操作が完了しないことに気付いてください。これは 1 つのスレッドを実行させているだけなので、そのスレッドは semop()の呼び出しでブロックされ、他のスレッド(main スレッド)がそれをブロック解除するのを待ってるためです。

- Next 操作をキャンセルするには Interrupt アイコンを押下して下さい。

NOTE

一部のディストリビューションの一部の glibc のバージョンは **semop(2)** ルーチン用の正確な walkback 情報が欠落する可能性があり、それはスレッドが停止した場所となります。このケースでは、後述する **walkback** と **interest** は本具体例に関して後述するようには反応しません。

同様に一部のシステムは glibc のデバッグ・バージョンがインストールされている可能性があり、その場合は NightView は semop()内部またはそれを呼び出すルーチンのソースコードを表示することが可能です。そうであっても、低レベル(=システム・コール)で停止されない限りは、後述するようにグレーの三角矢印が表示される可能性があります。

ソース・パネルの行番号の前にあるグレーの三角矢印は最上位のスタック・フレームではないスタック・フレームに位置しており、現在のフレームがサブプログラムの呼び出しを実行中であることを表しています。

デフォルトで、NightView は重要ではないフレームを非表示にします。全ルーチンについて(デバッグ情報なしであっても)全てのフレームを見たい場合は、*interest threshold* にキーワード `min` を設定することが可能です：

```
interest threshold min
```

一旦コマンドを実行したら、walkback 情報は全てのフレームを表示し、どのフレームにも位置を合わせて(必要であれば)アセンブリレベルでデバッグすることが可能です。

- 次のコマンドを介して **interest threshold** をゼロにリセットして下さい：

```
interest threshold 0
```

- Run Mode を Run All Threads に変更して下さい。

プロセスの再実行

プログラムをデバッグ中にしばしば、プログラムを再実行したくなることがあります。

NightView は自動的に全てのイベントポイント設定を記憶し、プロセスを再度実行した時にそれらを再び適用します。但し、スレッドの protected ステータスは再適用しません。

しかし、NightView は実行中に protected 属性を設定するために使用できる他のメソッドを持っています。

最初にプロセスを再実行します。

- Process ツールバー内の ReRun アイコンを押下してプログラムを再起動して下さい：

NOTE

あるいは、プロセスを起動するために Command フィールドから直接次のコマンドを実行することも可能です：

```
rerun
```

- 次のコマンドを使いプロセスを再開して下さい：

```
resume
```

ブレークポイントが自動的に再適用されブレークポイントに到達すると `watchdog_thread()`を含む全てのスレッドが停止しますが、これはもはや protected ではないためであることに注意して下さい。

スレッドの実行を開始した時に protected を常に再適用されるようにそのステータスを確保するにはパッチポイントを適用することが可能です。

- 286 行目までスクロール・ダウンしその行をクリックして下さい。
- Eventpoint メニューから Set Patchpoint を選択して下さい。
- Set thread local tag values ラジオボタンをクリックして下さい。

- Thread Tags テキストボックスに「protected=1」と入力して下さい。
- Enable, disable after next hit ラジオボタンをクリックして下さい。

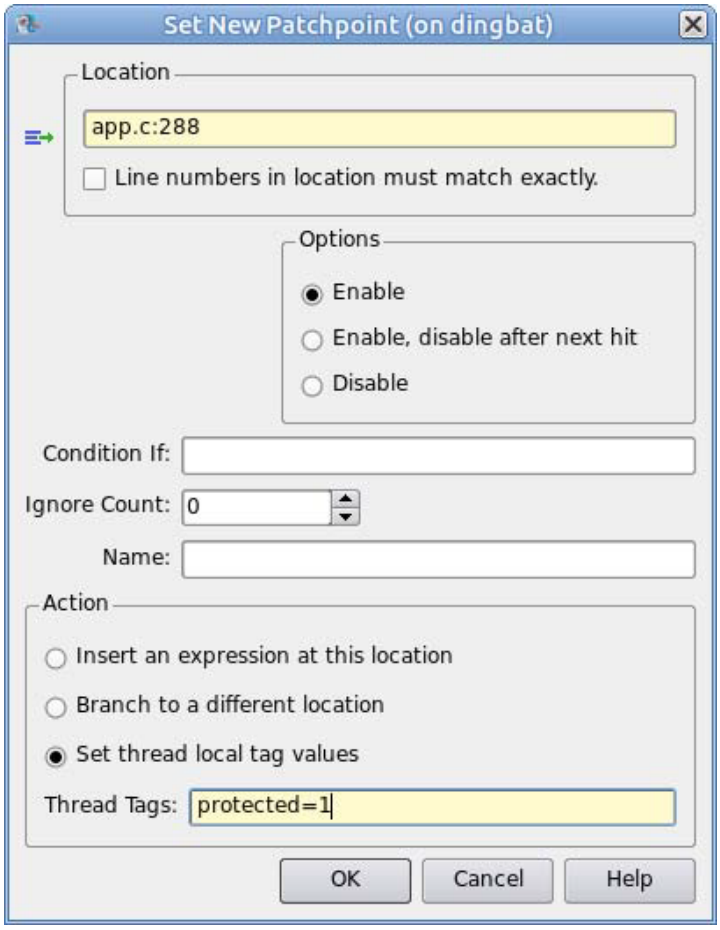


図 3-5. 変更中の Set Patchpoint ダイアログ s

- OK をクリックして下さい。

タスクが生成され 286 行目を実行すると直ぐにそれ自身が protected として設定されます。

- もう一度プロセスを再実行し再開して下さい：

```
rerun
resume
```

プロセスは 52 行目のブレークポイントに達しますが、ウォッチポイントは停止せずコンテキスト・ウィンドウ内にタグ(protected)を表示します。

- Eventpoints パネルのブレークポイント上を右クリックして Delete を選択、もしくは次のコマンドを実行して 52 行目のブレークポイントを削除して下さい：

```
clear app.c:52
```

- プロセスの実行を再開して下さい。

NOTE

NightView の重要な機能の 1 つはプロセスの実行を止める必要なく殆どのデバッグ操作を実行できる能力です。

本チュートリアルのいくつかのセクションでのデバッグ操作はプロセスを止めることなく行われます！

リンク・リストの横断

NightView のデータ表示パネルは変数を(ポインタを介して間接的に)表示および詳細レベルを展開または折り畳むことが可能です。変数は表示を容易にするためツリー内に示されます。

NightView は複雑なデータ構造体を簡単に表示させる 2 つの機能(リンク・リストとフィルタリング)を提供します。

本アプリケーションは各構造体のメンバーを介してリンクされた構造体のリストを生成しています。変数 head はそのリンク・リストの先頭を表しています。

簡単にするため、本セクションを進める前に既存のデータ・パネルを削除します。

- 既存のデータ・パネルを持ち上げた後にパネルのコントロール領域の右上のクローズ・アイコンをクリックして閉じて下さい。
- 次のコマンドを入力して新しいデータ・パネルに変数 head を追加して下さい：

```
data head
```

新しいデータ・パネルが現れてポインター変数 head が含まれます。

- ポインター変数およびそのメンバーのポインターlink、そしてその子供をいくつか展開して下さい。

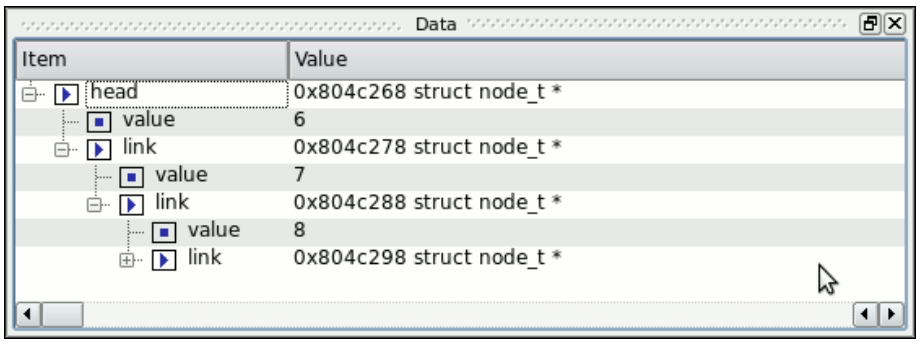


図 3-6. 展開されたリンク・リストへのポインタ

上図が示すようにリンク・リストの各ノードは、リスト内の前ノードの下にネストされます。これは申し分ない演出である一方、極わずかなノードにとどまらず表示をした途端に煩わしくなります。

代案として、ポインターがリンク・リストのメンバーであることを NightView に指定することが可能です。

- head 変数を右クリックして Treat As Pointer To Linked List...を選択して下さい。

小さなダイアログが現れ、リスト内の次のエレメントを定義する構造体のメンバーを指定することが可能です。

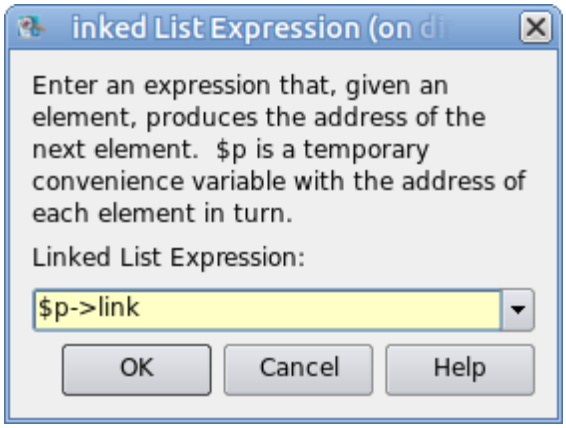


図 3-7. リンク・リスト・コンポーネントを選択するダイアログ

NightView はリスト内のリンクを示すのに相応しい型を持つ全てのメンバーを備えるドロップダウン・リストを自動的に追加します。今回のケースでは、リスト内の次のノードを特定するメンバーを正しく選択しています。

- OK を押下して下さい。

データ・パネル内の head 変数は別の方式を使って直ぐに表示されます。

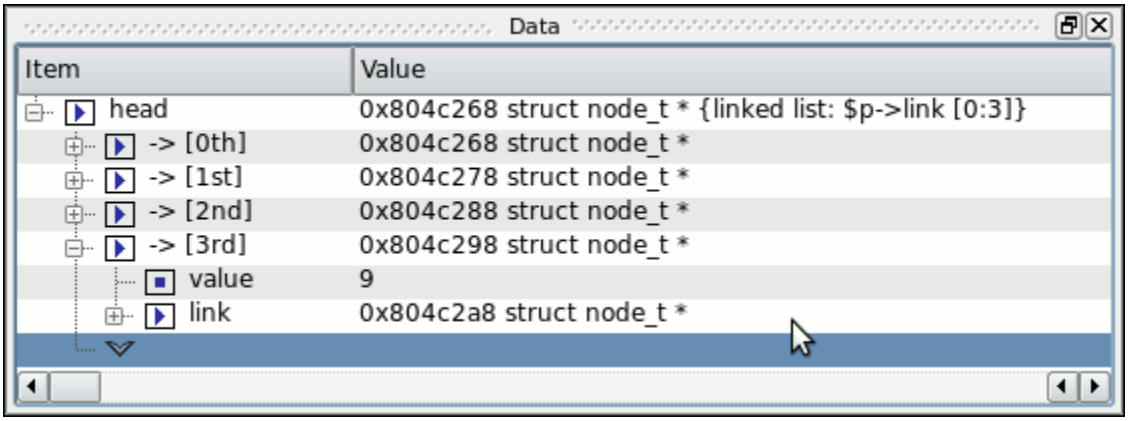


図 3-8. リンク・リストとして表示されたポインタ変数

上図ではリンク・リスト内のいくつかのノードが同じレベルで表示され 0 から始まる番号が付けられています。

- 「3rd」ノードが表示されるまでガード記号(青の三角)を何回かクリックして、上図に一致するように展開して下さい。

NightView は上で選択したリスト内の次の項目が定義されているメンバーが NULL ではない限りリストを展開することが可能です。(リストを展開するためにガード記号をクリックし続けるのとは対照的に)表示するリスト内のノードがいくつなのかを NightView に伝えるにはコンテキスト・メニューを使用することも可能です。

頻繁にリンク・リストを表示する場合はリスト内の特定のノードを指定して下さい。それをするには NightView のフィルター機能を使用します。

- head 変数を右クリックして Filter Elements with a Condition...を選択して下さい。

以下のダイアログが現れ、リスト内に表示するノードを定義する式を入力することが可能です。

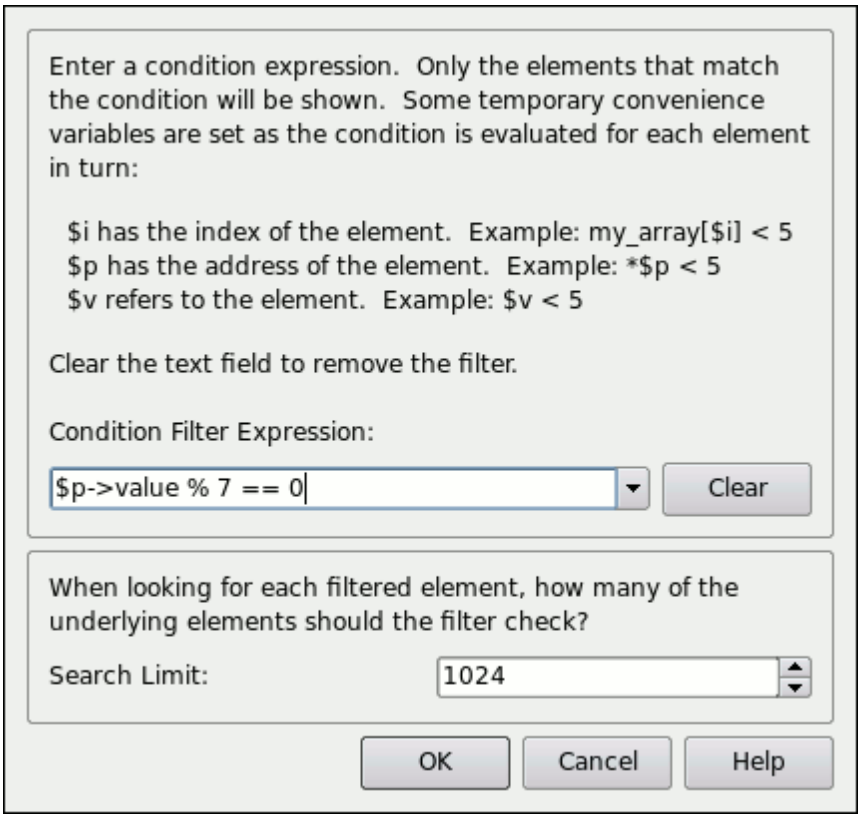


図 3-9. フィルター・ダイアログ

式はフィルターの指定に役立ついくつかの独特な組み込み変数を含めることが可能です。ダイアログ内の文章は次の変数について説明しています：$i, $p, $v

- 上図に示すように Condition Filter Expression フィールドに以下の文字列を入力し、OK を押下して下さい。

```
$p->value % 7 == 0
```

リスト内のメンバーvalue が 7 の倍数のノードだけを表示するように NightView に指定します。

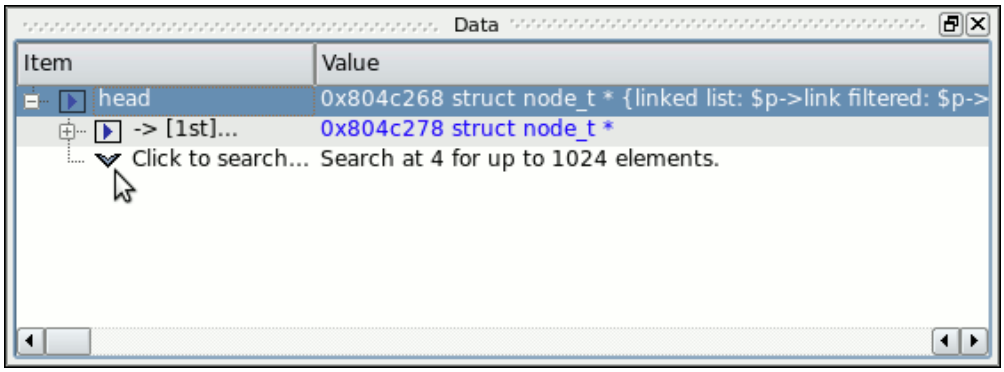


図 3-10. フィルター処理済みリンク・リスト

初めにフィルター条件に一致したリスト内の最初のノード(ノード番号 1)がリストの 2 番目のノード(ノード番号は 0 から始まります)に表示されます。

- ガード記号を 2 回クリックしてフィルター処理されたリストを展開し、3 つのフィルター処理されたエレメントを下の図と同じように全て展開して下さい：

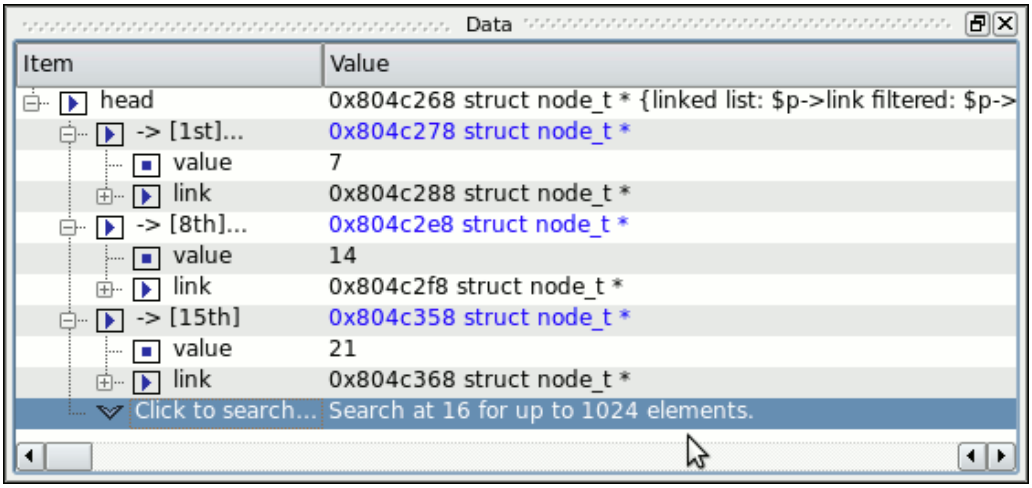


図 3-11. 展開されたフィルター処理済みリンク・リスト

リスト内の表示された全てのノードが 7 の倍数の value メンバーを持っており、前に指定したフィルター式を満足していることを理解して下さい。

リストの次のノードが連続していない場合は省略記号が各ノード番号の後に続き、フィルタリングによって表示されているリストにギャップがあるのを示していることに気付いてください。リンク・リストの先頭の説明フィールドもフィルタリングがアクティブであることを示しています。

リンク・リストだけでなく配列についても NightView のフィルタリング機能が利用することが可能です。実際に特定の値に関してメモリを通して検索するために使用できます。

単にデータ・パネルにポインターの値を追加して、コンテキスト・メニューを使い配列としてそれを扱うよう NightView に指定し、その後フィルターの式を適用して下さい。

モニタポイントの利用

モニターポイントは停止することなくプログラム内の変数の値を監視する手段を提供します。モニターポイントは指定された位置にデバッガーにより挿入されるコードで、それは記述した 1 つ以上の式の値を保存します。保存された値はその後 Monitor パネル内に NightView によって周期的に表示されます。

非同期サンプリングとは異なり、モニターポイントはアプリケーション内の個別の位置の実行に同期しているデータを見ることを可能にします。

- 52 行目を右クリックしてポップアップ・メニューから Set eventpoint を選び、サブ・メニューから Set Monitorpoint...を選択して下さい。

NOTE

あるいは、Set New Monitorpoint ダイアログを起動するために Eventpoint メニューから Set Monitorpoint...オプションを選択、またはツールバーから Set Monitorpoint アイコンをクリックすることが可能です。

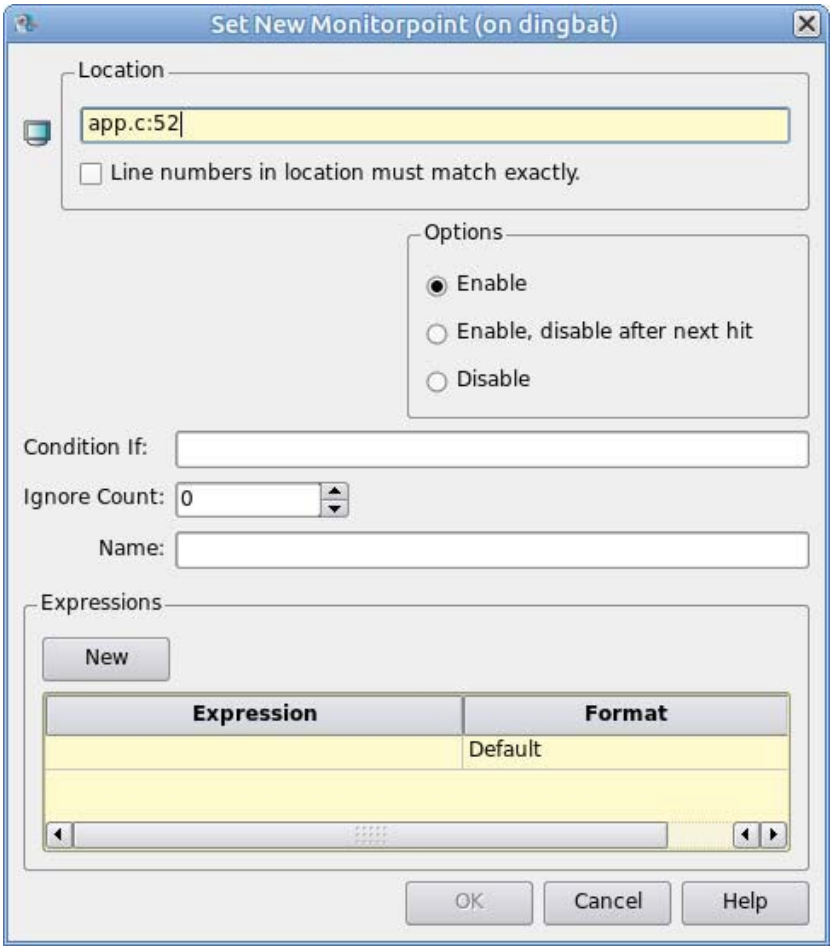


図 3-12. Monitorpoint ダイアログ

- Location テキスト・フィールドに app.c:52 があることを確認し、必要であれば修正して下さい。
- Expression 列見出しの下のテキスト・フィールドに以下を入力して下さい：

```
data->count
```

但し、Enter キーはまだ*押下しないで下さい*。

- Label 列に以下を入力して下さい：

```
sine count
```

- Label 列の下のセルにまだ置かれている間に Tab キーを押下して下さい。これで次の行に展開し式の追加を続けることが可能になります。

NOTE

既にそのセルから離れてしまい 1 行だけ表示されている場合は New ボタンを押下して下さい。

- Expression 列の下の 2 番目の行で以下を入力して下さい：

```
data->value
```

- 以下を入力して Label 列にラベルの値を設定して下さい：

```
sine value
```

- Set New Monitorpoint ダイアログ内の OK ボタンを押下して下さい。

Monitor パネルが上記で入力したコマンドのエントリーを収容して生成されます。

- 同様に前述のモニターポイントにあるように同じコマンドを使い、Label フィールドの sine を cosine に置換して 69 行目にモニターポイントを設定して下さい。

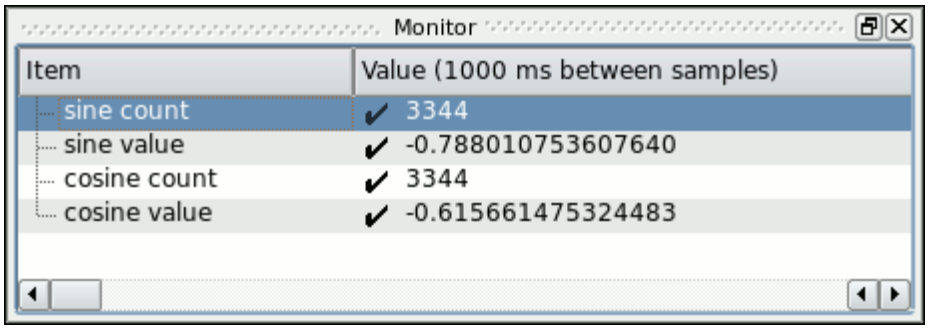


図 3-13. NightView の Monitor パネル

この時点で、Monitor パネルのデータ値は変化します。

52 行目と 69 行目がそれぞれ実行された時は直ぐに値がサンプリングされます。NightView は最新の値一式をユーザーが選択可能な速度で Monitor パネルに表示します。

イベントポイント条件と無視回数の利用

NightView の全てのイベントポイントは任意の*条件*および*無視*属性を持っています、

*条件*はイベントポイントが実行される前に評価するユーザー提供の任意の複雑性論理式です。条件にはユーザー・アプリケーション内の関数呼び出しを含めることが可能です。

同様に*無視*属性は実際に実行する前にイベントポイントを無視する回数のカウントとなります。

条件と無視カウントはコードに適用されたパッチを介してアプリケーション自身により評価されますので、完全なアプリケーション速度で実行されます。他のデバッガーはデバッガーのコンテキスト内部から条件および無視カウントを評価しますので、著しく時間を必要としプログラムの挙動に激しく影響を及ぼす可能性があります。

- 52 行目へのモニターポイントを表している Eventpoint パネルの行の Ignore 列のセルをクリックして下さい。
- 値を 500 に変えて Enter を押下して下さい。

Monitor パネルは値の前に疑問符を表示することでそのモニターポイントの値がサンプリングされていない事を示します。無視カウントがゼロに達した場合、その値は再び更新を開始します。

最後にモニターポイントは簡素な変数だけではなく複雑な式を含めることが可能です。

- NightView メイン・ウィンドウの Command フィールドに以下のコマンドを入力して下さい：

```
monitor app.c:105
    p FunctionCall()
end monitor
```

ユーザー・アプリケーションが毎回 105 行目を横断することにより実行される関数呼び出し FunctionCall()の結果を示す新しい項目が Monitor パネルに追加されます。

パッチポイントの利用

ブレークポイントやモニターポイントとは異なり、パッチポイントはプログラムの挙動を変更することが可能です。

パッチポイントはプログラムの流れの変更、もしくは変数またはマシーン・レジスタの変更が可能です。

最初に当該プログラム内の一部の命令文の周りを分岐するためにパッチポイントを使用します。

NOTE

ソース・ファイル **app.c** が表示されていない場合、次のコマンドを実行して下さい：

```
l app.c:53
```

- NightView メイン・ウィンドウに表示されたソース・ファイルをスクロールして 53 行目を右クリックして下さい。

```
data->angle += data->delta
```

続いてコンテキスト・メニューから Set eventpoint を選択しサブメニューから Set Patchpoint... を選択して下さい。

NOTE

あるいは、Set New Patchpoint ダイアログを起動するために Eventpoint メニューから Set Patchpoint...オプションを選ぶ、またはツールバーから Set Patchpoint アイコンをクリックすることが可能です。

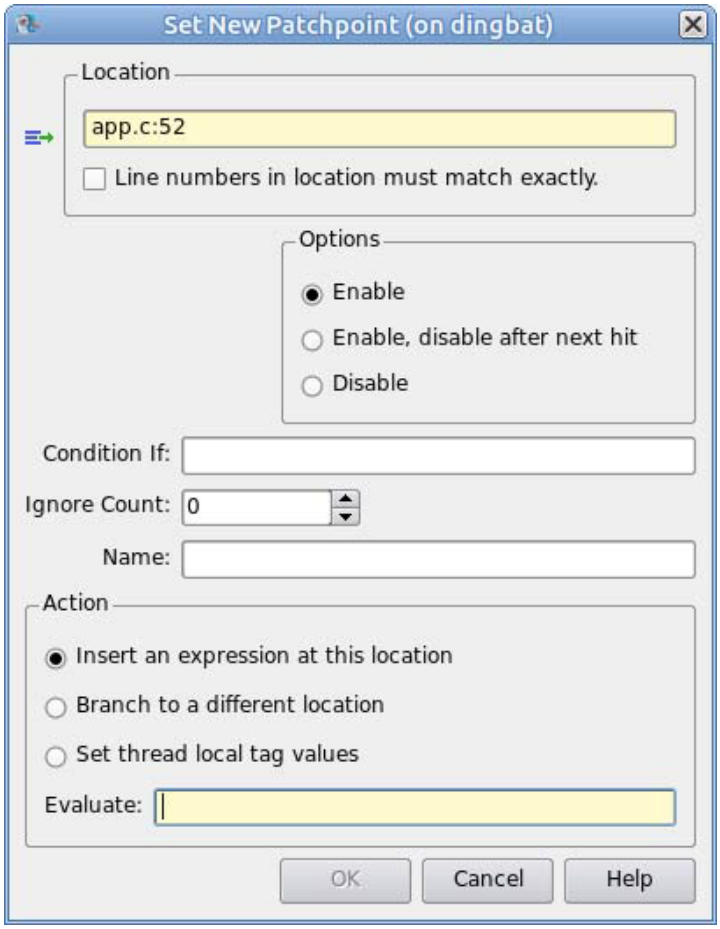


図 3-14. Patchpoint ダイアログ

- Location テキスト領域で文字列が app.c:49 と表示されていることを確認して下さい。
- ダイアログ下部の Branch to a different location ラジオボタンをクリックして下さい。
- Go To:テキスト領域で次を入力して下さい：

```
app.c:54
```

次に OK ボタンを押下して下さい。

これは事実上アプリケーションにその後の sin()の呼び出しの中で使用される angle を更新する 53 行目の実行をスキップさせます。

Monitor パネルの sine value は変化が停止していますが、関連する sine count の値は変化し続けていることに注意して下さい。

あるいは、式または変数の値を変更するためにパッチポイントを使用することも可能です。

- NightView メイン・ウィンドウの Command パネルに以下のコマンドを入力して下さい：

```
patch app.c:52 eval data->count -= 2
```

1 ずつ増加し続けているのが、今は同様に 2 ずつ減少しているのを夫々繰り返すため、sine count の値が減少している事に注意して下さい。

パッチポイントを削除することなく無効にすることが可能です。

- Eventpoints パネルの(単語 Patch で Type 列に表示されている)両方のパッチポイントを選択して右クリックし、ポップアップ・メニューから Disable を選択して下さい。

パッチは無効となり Monitor パネル内に表示される値は元の挙動に戻ります。

動的な関数の追加および置き換え

NightView は動的に新しい関数をデバッグ中のアプリケーションに追加するだけでなく、同様に既存の関数を置き換える機能を提供します。

- NightView の外側の端末セッションにて、本チュートリアルの初期段階でカレント・ディレクトリにコピーした **report.c** ソース・ファイルをコンパイルして下さい：

```
cc -g -c report.c
```

- NightView メイン・ウィンドウの Command パネルに以下のコマンドを使ってプログラムの中に新しいモジュールをロードして下さい：

```
load report.o
```

stdout に情報を出力する簡単な関数を追加しました。この関数は任意の複合体でアプリケーション内のどの変数でも参照することが出来るはずです。唯一の制限は、ロードされたモジュールに存在せずユーザー・アプリケーション内にもないシンボルを関数が参照できないことです。

- 関数 report()のソース・コードを見るには以下のコマンドを実行して下さい：

```
l report.c
```

report()関数は変数の型がそれぞれ char *と double のペアを求めていることが分かります。

- 以下のコマンドを実行してアプリケーション・ソース・ファイルに戻って下さい：

```
l app.c
```

新しく追加した関数を呼び出す新しいパッチポイントをインストールします。

- 以下の式を含めて app.c:68 上にパッチポイントを設定して下さい：

```
report("cos",data->value)
```

report()関数の呼び出しが実行されるとプログラムは NightView メイン・ウィンドウの Messages パネルに **stdout** への出力を生成します。

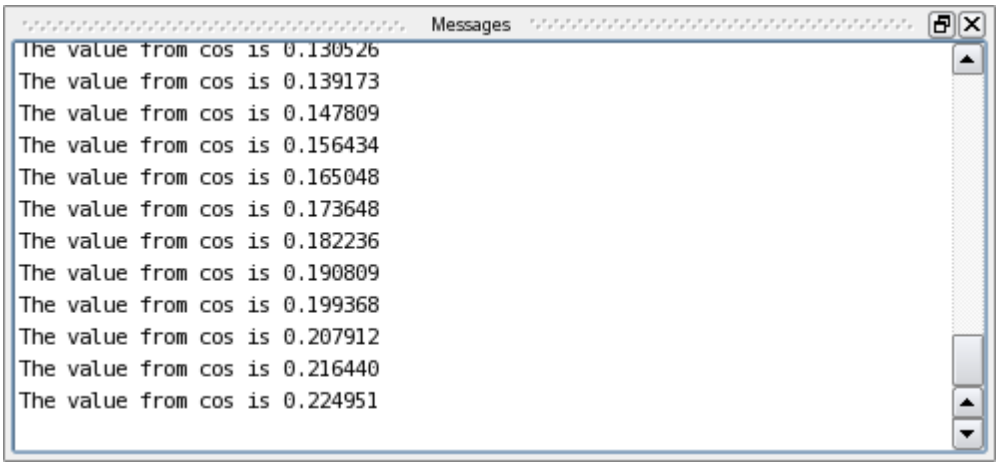


図 3-15. 新たにロードされた関数を呼び出すパッチの結果

- Eventpoints パネルの Enabled チェックボックスをクリアして今しがた追加したパッチポイントを無効にしてください。

最後にアプリケーション内に既に存在している関数を置き換えます。

- NightView の外側の端末セッションにて、本チュートリアルの初期段階でカレント・ディレクトリにコピーしたソース・ファイル **function.c** の内容を表示し、続いて以下のコマンドでコンパイルして下さい：

```
cat function.c
cc -g -c function.c
```

- NightView メイン・ウィンドウの Command パネルに以下のコマンドを入力して置き換え用コードをロードして下さい：

```
load function.o
```

FunctionCall()値に関する Monitor パネルの値はもはやアプリケーションにて計算される値とは関連せず、むしろソース・ファイル **function.c** により単調に数字が増加していることに注意して下さい。

- 以下のコマンドを介して NightView メイン・ウィンドウのソース・パネルを **app.c** ソース・ファイルの 41 行目に戻して下さい：

```
l app.c:41
```

トレースポイントの利用

チュートリアルのこの部分では NightView と NightTrace との統合について取り上げます。

トレースポイントは、基本的にオプションの引数を伴うトレース・イベントを記録する呼び出しを行うためのパッチを適用する特殊なイベントポイントです。

例えアプリケーションが NightTrace API をまだ使用していなくても、NightView は必要なコンポーネントをリンクしてトレース・モジュールをアクティブにすることが可能です。本アプリケーションは既に NightTrace API を使用していますので、それは必要ありません(NightTrace API を使用していないアプリケーション内でのトレースポイントの使用に関する詳細については *NightView User's Guide* の **set-trace** コマンドを参照して下さい)。

sine_thread()と cosine_thread()ルーチンの周期性能の計測に関心があり、またそのサイクル中のデータ値の記録にも関心があると仮定して下さい。

- NightView メイン・ウィンドウに表示されているソース・ファイルをスクロールして 53 行目を右クリックして下さい：

  ```
  data->angle += data->delta
  ```

  続いてポップアップ・メニューから Set eventpoint を選択し、サブメニューから Set Tracepoint...を選択して下さい。

NOTE

あるいは、Eventpoint メニューから Set Tracepoint... を選択してダイアログを起動、またはツールバーの Set Tracepoint アイコンをクリックして Set New Tracepoint ダイアログを起動することが可能です。

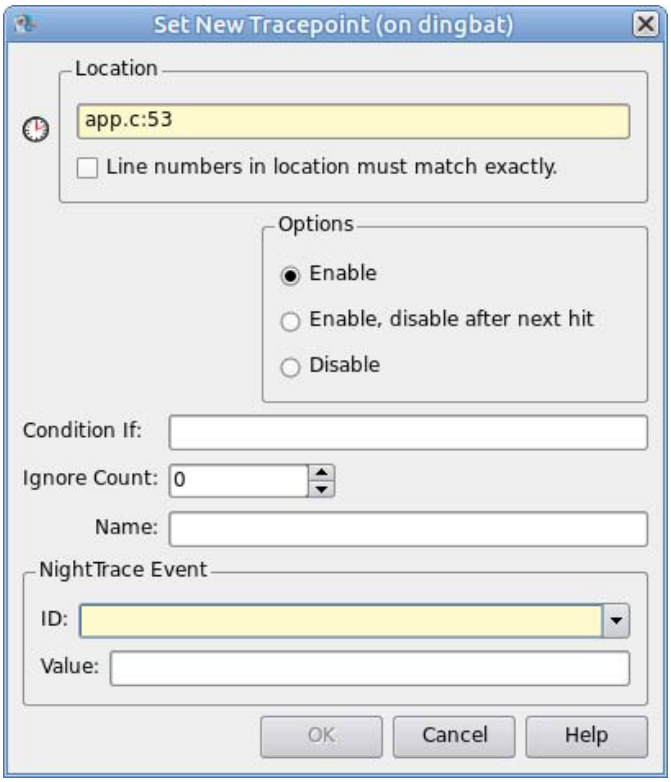


図 3-16. Tracepoint ダイアログ

- Location:テキスト・フィールドに **app.c:53** が表示されていることを確認して下さい。
- NightTrace Event セクションの ID フィールドに以下を入力して下さい：

  1

- OK ボタンを押下して下さい。

同様に追加のトレースポイントを設定しますが、トレースポイントと一緒に記録させる値も指定します。

- **app.c:**51 上にトレースポイントを設定し Event ID に **2** を指定、Value テキスト・フィールドに以下を入力して下さい：

  data->value

- **app.c:**68 上にトレースポイントを設定し Event ID に **3** を指定、Value テキスト・フィールドに以下を入力して下さい：

  data->value

トレース・イベントは NightTrace ツールを使って記録させることが可能となりましたが、まず NightView のヒープ・デバッグ機能を考察し、その後 NightTrace の起動に戻ります。

ヒープ・デバッグ

本セクションでは非常に便利な NightView のヒープ・デバッグ機能について説明します。

しかしながら、NightTrace セクションに今すぐ取り掛かりたい場合は 3-42 ページの「NightTrace の準備」に進んで下さい。今後いつでも本セクションに戻ることは可能です。

動的メモリの問題をデバッグするのは困難かつ極めて時間を消費する可能性があります。単語ヒープは一般的には C 言語の malloc()と free()ユーティリティによって制御される割り当ておよび解放されるメモリの集合を指します。

NightView は、お手持ちのプログラムをコンパイルまたはリンクさせる非標準アロケーターを必要とせず、メモリの割り当て、解放、設定のエラーを監視および検出するための優れた機能を提供します

この強みの一つはデバッグ・アロケーターにスイッチする時は大抵、ブロックを割り当ておよび解放する方法の変更が、探そうとしている正にそのバグを頻繁に隠していることです。

NightView は一般的なヒープ関連エラーを捕らえるのに便利な様々な設定やデバッグ・レベルを用意しています。一部の設定でシステム・アロケーターの挙動、割り当てられたブロックのサイズへの作用、最終的に返されるアドレス値を変更します。

動的メモリのエラーは以下 4 つの方法の 1 つで検出されます：

- ヒープ関数(例：malloc, free, calloc,等)の呼び出しの数を単位として指定された頻度でヒープ全体をチェック
- free または realloc が呼ばれた時に個々の割り当てられたブロックをチェック
- **heappoint** を交差した時にヒープ全体をチェック
- **heapcheck** コマンドが実行された時にヒープ全体をチェック

heapdebug コマンドまたは Debug Heap ウィンドウの頻度の設定はユーティリティ・ルーチンが呼ばれた時にどれくらいの頻度で NightView がヒープ・エラーをチェックすべきかを制御します。頻度を 1 に設定すると NightView は毎回のヒープ操作でヒープ・エラーについてチェックします。

heappoint はプロセスがヒープポイントが挿入されている命令を実行する時に NightView がエラーについてチェックします。ヒープポイントの数は無制限でお手持ちのプログラムに挿入することが可能です。

free または realloc が呼ばれた時の個々のブロックのチェックは自動です。

4 つのメカニズム全てが役立ちます。最初の 3 つのメカニズムがあれば、ヒープ・エラーの検出はデバッガーへのコンテキスト・スイッチなしでプログラム・アプリケーションの速度で実行されます。

ヒープ・デバッグの始動

ヒープ・デバッグの制限は、プログラム内でアロケーションが発生する前にデバッグをアクティブにすることが必要なこと、いくつかのヒープ・デバッグ・コマンドはプロセス全体が停止している場合のみに機能することとなります。アロケーションが既に発生した後にヒープ・デバッグ機能をアクティブにしようとした場合、NightView はその要求を満たすことができないことを知らせます。

- (恐らく以前に本セクションをスキップしているために)app プロセスが現在デバッグ中ではない場合はプロセスをロードして下さい：

```
nview ./app
```

- さもなければ、アプリケーションが NightView で今なお実行中である場合、(本セクションでは何度も停止するために)watchdog_thread の protected 属性を無効にしてリアルタイム処理ループに入ることを防ぐ必要があります。

```
stop /protected
select-context name="watchdog_thread"
set $thr.protected=0
clear app.c:286
```

- watchdog_thread を本質的にアイドル状態にするため以下のコマンドを入力して下さい：

```
patch 293 goto 307
```

- NightView メイン・ウィンドウの Process メニューから Debug Heap...メニュー・オプションを選択して下さい。

Debug Heap ウィンドウが現れます。

- ダイアログ上部の Enable heap debugging チェックボックスを選択して下さい。
- Debugging Level 領域の Medium ボタンを押下して下さい。
- Specify check heap freq テキスト・フィールドを 1 に変更して下さい。

Debug Heap ウィンドウは下図と同じように見えるはずです：

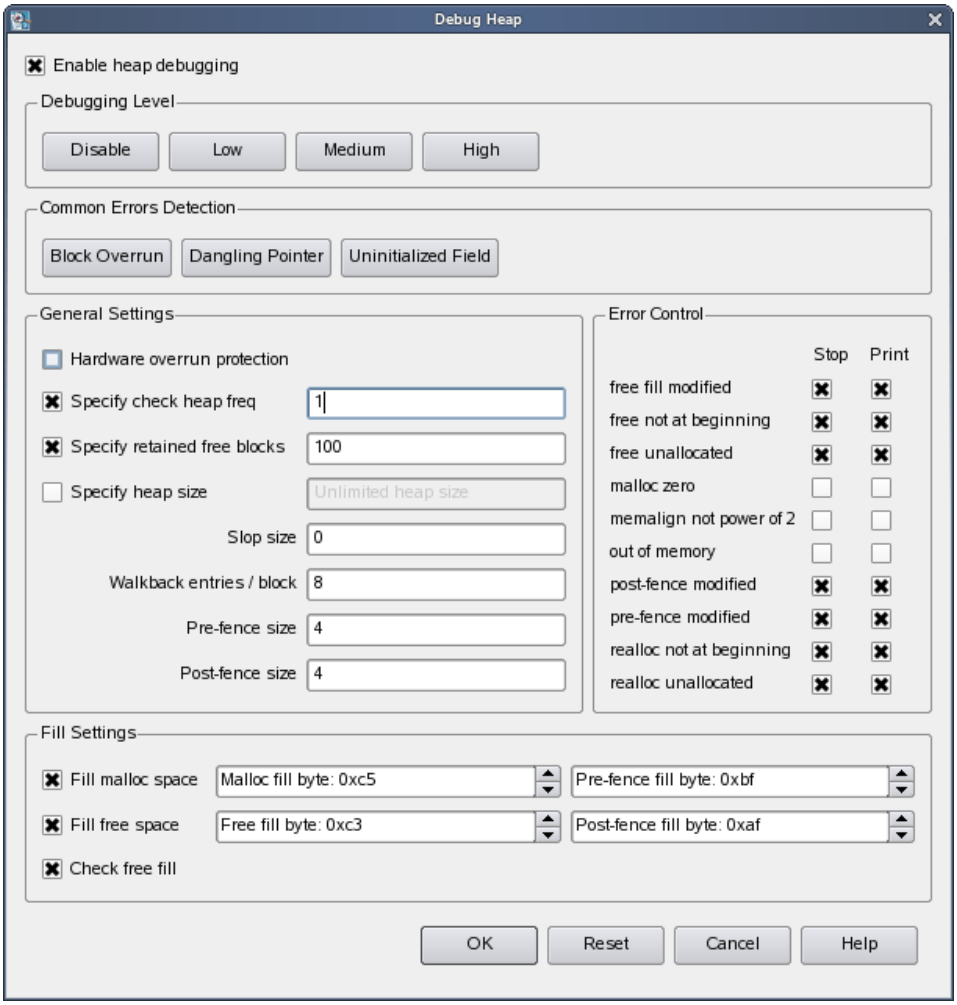


図 3-17. NightView の Debug Heap ダイアログ

- 変更を適用するため OK ボタンを押下してダイアログを閉じて下さい。

これらのオプションはデバッガーにヒープ・デバッグをアクティブにすること、確かなエラーの種類を検出するために解放されたブロックを記憶すること、エラーを検出するために要求されたサイズのメモリの終端を過ぎたところに追加のメモリを割り当てること、どのヒープ・エラーを検出してもプログラムを停止することを指示します。

- 処理の実行を再開するため Resume アイコンをクリックして下さい。

ヒープ・デバッグ・シナリオの設定

main プログラムにより生成された 4 番目のスレッドは heap_thread と呼ばれるルーチンを実行します。

このルーチンは scenario 変数の設定に基づいて様々な動的メモリの操作を繰り返し実行します。これらの操作は動的メモリに関わる一般的なユーザー・エラーを代表するものです。

130 行目にブレークポイントを設定しましょう。

- ソース・ウィンドウの 130 行目にスクロールして下さい：

```
sleep(5);
```

- その行のどこかで右クリックし、ポップアップ・メニューから Set simple breakpoint を選択して下さい。

NOTE

オプションで、Eventpoint メニューから Set Break- point メニュー項目を使用、または NightView メイン・ウィンドウの Command パネルで以下のコマンドを入力のどちらかにより 130 行目にブレークポイントを設定することが可能です。

```
break app.c:130
```

プロセスは heap_thread 内のブレークポイントに達して全てのスレッドが停止します。

シナリオ 1: 解放されたポインタの使用

一般的なエラーは既に解放されたメモリー・ブロックを読み書きすることです。

それを検出する方法は解放されたブロックを記憶し、特定のパターンで解放されたブロックを埋めることを NightView に指示する事です。その後にブロックが読まれた場合、その内容が予想外であるためにアプリケーションはエラーをより速く発見することが可能です。その後にブロックに書かれた場合、NightView はそれを検出することが可能です。

デフォルトで heap_thread は 5 つのシナリオのいずれも実際には実行されません。

- シナリオ 1 を実行させるには、Command フィールドで以下のコマンドを入力して変数 scenario に 1 を設定して下さい：

```
set scenario=1
resume
```

これは 5 秒遅れて以下のコードの断片を実行させます：

```
ptr = alloc_ptr(1024,3);
free_ptr (ptr,2);
memset (ptr, 47, 64);
```

最後の行は既に解放されている動的に割り当てられた空間の使用を示しています。

NightView はユーザーが挿入したヒープポイント、またはこの場合は 170 行目の(**heapdebug** コマンドの **frequency** 設定に基づく)ヒープ操作でこれを検出します。

NightView はヒープ・エラーを検出した時点でプロセスを停止し、以下のような診断結果を出力します：

```
Heap errors in process local:3771:
    free-fill modified in free block (value=0x804a818)
#0 0x8048b6d in heap_thread(void*unused=0) at app.c line 170
```

エラーは解放されたブロック内の場所がブロックが解放された後にプロセスによって変更された事実に言及します。

Data パネルはヒープに関する情報だけでなく様々な他の属性を表示するのに便利です。

- Data メニューから Heap Information を選択して下さい。

  Data パネルが Locals や Context パネルと同じ位置で NightView メイン・ウィンドウに追加されます。新しいタブが Data パネル用に生成されます。

- 新たに追加された Data タブをクリックして下さい(既に現在のタブとして持ち上がっているはずなので、これは恐らくは必要ありません)。
- (必要であれば)下の興味のある項目の全体が見えるように列の見出しの間にある仕切りをクリックし右へドラッグして最初の列のサイズを変更して下さい。
- 現在の **heapdebug** 設定を見るには、Data パネル内の Heap Information の下にある Configuration 項目を展開して下さい。

- ヒープの動きに関する簡易統計を見るには Heap Information の下の項目 Totals を展開して下さい。

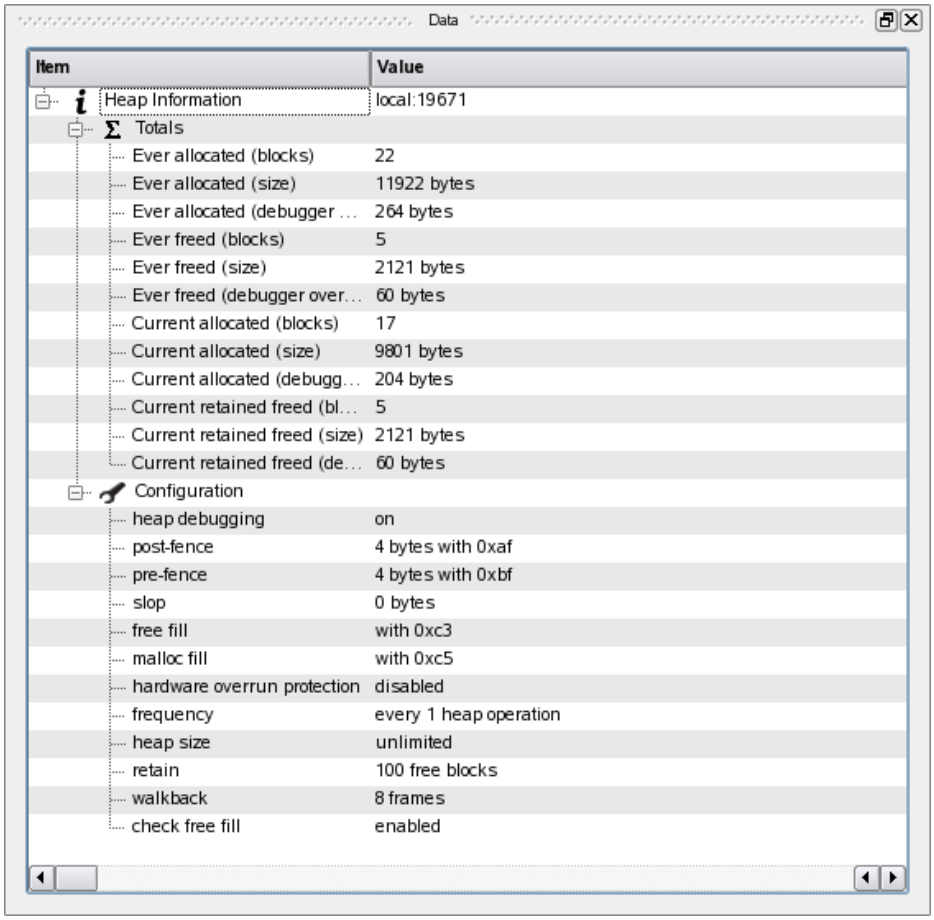


図 3-18. ヒープの総計と構成

NOTE

通常、Data パネル内の全ての情報はいつデバッグ中のプロセスを停止したとしても更新されます。

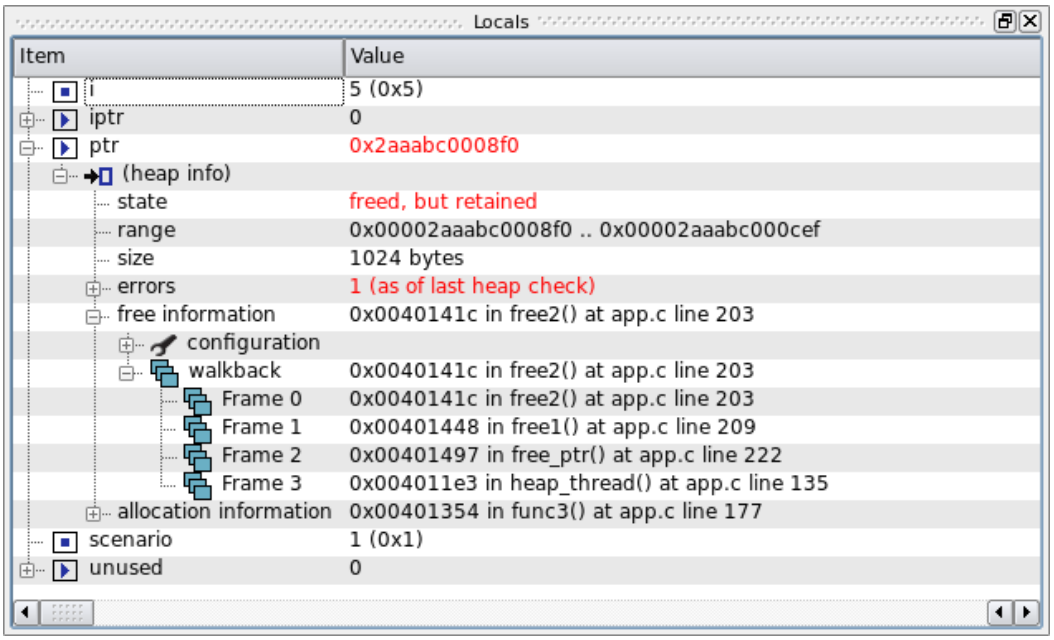
Totals セクションの値はシステムによって異なります。

- Totals と Configuration 項目を折り畳んで下さい。
- Locals タブをクリックして下さい。

Locals パネル内の項目リストは現在表示されているフレームに関連したローカル変数を示しています。変数 ptr の値はもはや有効な(割り当てられた)ヒープ・アドレスを含んでいないために赤で表示されていることに注意して下さい。

ptr 項目を展開すると(heapinfo)項目を表示します。その項目を展開すると次を含む一度参照されたポインターのブロックに関する追加情報を表示します：

- 状態 - freed, but retained
- アドレス範囲
- サイズ
- エラー
- 解放および割り当て情報は、展開された時に割り当てられたおよび解放されたブロックのルーチンに関する walkback 情報を含みます



シナリオ 2: 無効なポインタ値の解放

次の一般的なエラーはポインターを複数回解放する、つまり実際にはヒープ・ブロックを参照していない値を解放することです。

- プロセスを再開して 130 行目のブレークポイントに到達させて下さい：

```
resume
```

- 変数 scenario に 2 を設定して下さい：

```
set scenario=2
resume
```

これは 5 秒遅れて以下のコードの断片を実行させます：

```
ptr = alloc_ptr(1024,3);
free_ptr(ptr,2);
free(ptr);
```

NightView は障害を検出し以下のような診断結果を出力します：

```
Heap error in process local:3771: free called on freed or unallocated block
(value=0x804ac40)
#0 0x8048a78 in heap_thread(void*unused=0) at app.c line 142
```

NOTE

もし glibc のデバッグ情報がシステムにインストールされている場合、NightView は glibc アロケーター内の異なるフレームを示す可能性があります。その場合は、heap_thread 内のフレームが表示されるまで **up** コマンドを入力して下さい。

問題になっているヒープ・ブロックに関する情報を取得する別の方法は、**info memory** コマンドを使用することです。これは Locals パネル内の ptr 項目の下で利用できる情報のテキスト出力を NightView メイン・ウィンドウの Messages パネルに対して提供します。

- Command パネル内で次のコマンドを実行して下さい：

```
info memory ptr
```

NightView は Messages パネル内に次のような出力を提供します：

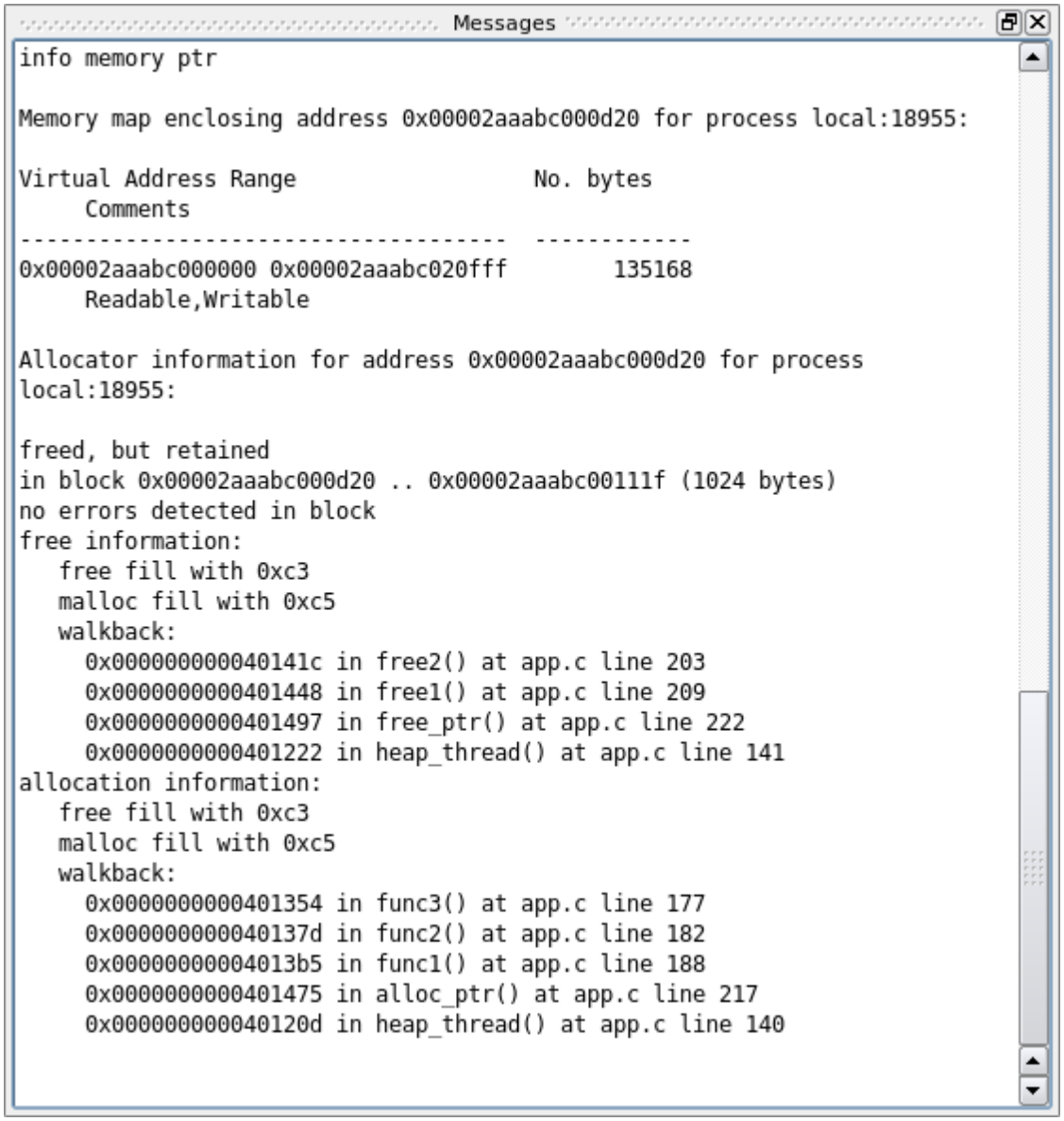


図 3-19. info memory コマンドの出力

上図は本質的にブロック内にエラーがないことを報告していることに注意して下さい。ここでの実際の問題は、2 度目の試みが既に解放されていたブロックを解放したことです。

この場合、実際の解放に関連する walkback 情報は、実際にブロックを解放したコード・セグメントを直ぐに探すことが出来るので便利です。

シナリオ 3: 割り当て済みブロックの端を超えた書き込み

次の一般的なエラーは不十分な空間を割り当てる、つまり割り当てられたブロックの終わりを超えて書き込むことです。

- プロセスを再開して 130 行目のブレークポイントに到達させて下さい：

```
resume
```

- 変数 scenario に 3 を設定して下さい：

```
set scenario=3
resume
```

これは 5 秒遅れて以下のコードの断片を実行させます：

```
ptr = alloc_ptr(strlen(MyString),2);
strcpy (ptr, MyString); // oops -- forgot the zero-byte
```

NightView は障害を検出し以下のような診断結果を出力します：

```
Heap errors in process local:3771:
post-fence modified in block (value=0x804b068)
#0 0x8048b6d in heap_thread(void*unused=0) at app.c line 170
```

Locals パネル内の変数 ptr の説明は無効なステータスを示していないことに注意して下さい。これは ptr が有効なヒープ・ブロックを指し示しているためです。

一方、ptr と errors リストに関する(heapinfo)の情報を展開すると、post-fence(ブロックの最後尾)が変更されたため ptr に参照されているブロックは無効であることを示します。

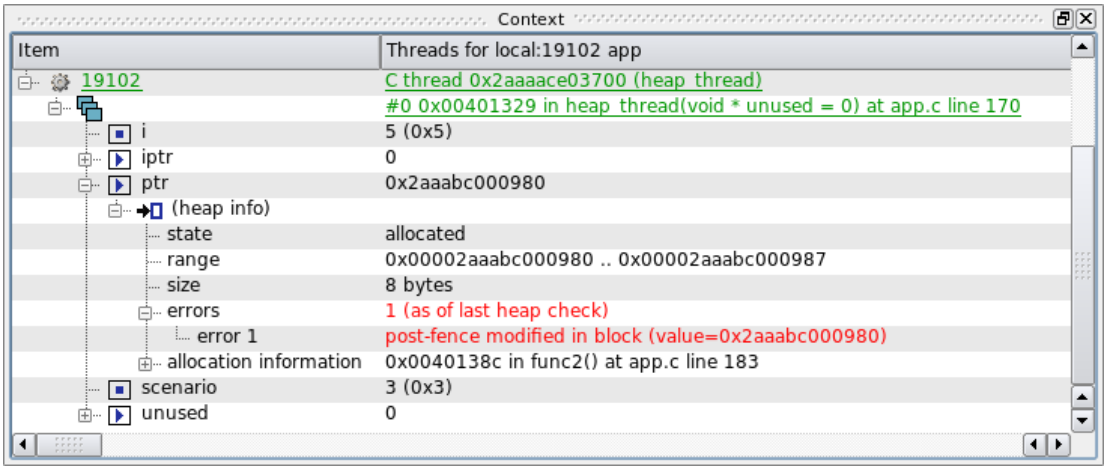


図 3-20. ヒープ・エラーの説明

シナリオ 4: 未初期化ヒープ・ブロックの使用

次の一般的なエラーは動的に割り当てられたメモリを使用する前に初期化するのを忘れることです。コード・セグメントは、malloc()ではなく calloc()の場合のように動的に割り当てられたメモリがゼロに初期化されていると仮定している可能性があります。

- プロセスを再開して 130 行目のブレークポイントに到達させて下さい：

```
resume
```

- SIGSEGV が送られたらいつでも停止することを NightView に伝えて、同様に変数 scenario に 4 を設定して下さい：

```
handle sigsegv stop print pass
set scenario=4
resume
```

これは 5 秒遅れて以下のコードの断片を実行させます：

```
iptr = (int**)alloc_ptr(sizeof(void*),2);
if (*iptr) **iptr = 2778;
```

NightView は障害を検出し以下のような診断結果を出力します：

```
Process local:3771 received SIGSEGV
#0 0x8048ad2 in heap_thread(void*unused=0) at app.c line 153
```

あるヒープ・デバッグ・オプションは、非初期化メモリの使用を簡単に検知するために新たに割り当てられた未初期化空間を特定のパターンで満たすように NightView に指示します。ヒープ・デバッグを有効にした時に使用した Debug Heap ダイアログ内の Fill malloc space フィールドがバイト・パターンを 0xc5 にするように指定しました。

- 非初期化メモリ・ブロックの中身を表示するには以下のコマンドを実行して下さい：

```
x/x iptr
```

SIGSEGV シグナルは致命的なエラーなのでチュートリアルを継続するにはプロセスを再始動する必要があります

- 以下のコマンドを実行して下さい：

```
kill
```

- Process ツールバー内の ReRun アイコンを押下してプログラムを再始動して下さい：

NOTE

あるいは、プロセスを始動するために Command フィールドから直接以下のコマンドを実行することも可能です。

```
rerun
```

NOTE

NightView はプログラムの次の実行を確かめる時に全てのイベントポイントやヒープ制御設定を自動的に再適用します。

シナリオ 5: リークの検出

エラーまたはメモリの不適切な使用を示す可能性がある他の状況はリークです。この事例では、プログラム内のどのポインタにも参照されていない動的に割り当てられたメモリのブロックとしてリークを定義します。

リークの検出は CPU 使用率およびユーザー・アプリケーションへの侵害に対しては***非常に高価***な行為です。そうであるため、リーク検出は明確な要求がユーザーからされた時にのみ実行されます。

- プロセスを再開して 130 行目のブレークポイントに到達させて下さい：

```
resume
```

- 以下のコマンドを実行して下さい：

```
set scenario=5
resume
```

これは 5 秒遅れて以下のコードの断片を実行させます：

```
ptr = alloc_ptr(37,1);
ptr = 0;
```

NightView は前述のように自動的にはリークを検出しません。プロセスは 130 行目のブレークポイントに到達した時に再度停止します。

- この時点で、Data パネルに項目を追加するために Data メニューから Heap Leaks...を選択してリークの報告を明確に要求し、New Leaks ラジオボタンをチェックして、Data Heap Leaks ダイアログ内の OK を押下して下さい。

  本操作は NightView にリークに関してプログラムを解析させて Data パネル内の Leak Sets 項目を表示します。小さなプログラムでは本操作は重要でないように見えるかもしれませんが、より大きなプログラムに関してはかなりの時間を必要とする可能性があります。

- Data タブをクリックして下さい。
- 必要であれば Leak Sets 項目を展開して下さい。

  割り当てられたブロック・サイズと一致するリーク・セット毎に一致する walkback と共に追加項目が表示されます。個々のセットの展開は、各割り当てだけでなく展開可能な個々のリークしたブロックの説明について表示する共通の walkback を提供します。

- サイズが 37 のリーク・セット項目を展開し、続いてそれに関する walkback 項目を展開して下さい。

app.c.の 157 行目の heap_thread()ルーチンによって割り当てられた事を walkback が示している事に注意して下さい。

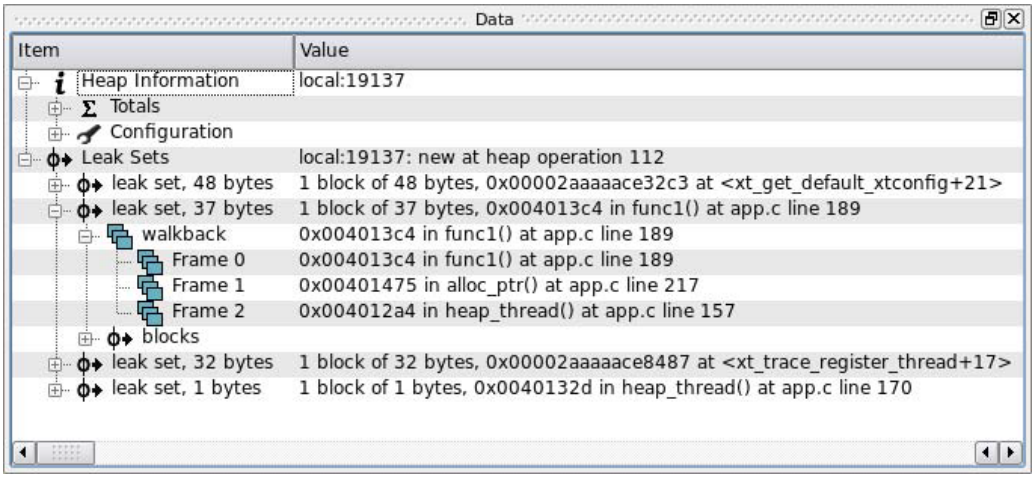


図 3-21. Heap Leaks の表示

NOTE

Leak Sets の表示はお手持ちのシステムによって変わります。上で示したように 37byte のリーク・セットに注意を向けて下さい。

表示された最後のフレームは、システムに pthread デバッグ・ファイルがインストールされているかどうかによりますが、heap_thread()内または pthread ライブラリ(start_thread())内である可能性があります。

NOTE

Data パネル内の殆どの項目とは異なり、プロセスが停止した時に leak sets 項目は自動的に更新されません。記述はプログラム実行中のある時点でのリークのスナップショットであるため、追加のリークが発生したとしても変更されないままとなります。情報を更新するには、(Data メニューから Heap Leaks...を選択して)別のリークの報告を要求して下さい。

Scenario 6: Allocation Reports

NightView は割り当てられた全てのメモリの詳細な報告を提供します。

その報告の作成は CPU 使用率およびユーザー・アプリケーションの実行時間への侵害に対しては***非常に高価***な行為です。そうであるため、割り当ての報告は明確な要求がユーザーからされた時にのみ実行されます。

- 変数 scenario に 6 を設定して下さい：

```
set scenario=6
resume /one
```

これは(**resume** コマンドに対する**/one** パラメータであるため) heap_thread のみを再開し、追加の割り当てを行います。

スレッドは 130 行目のブレークポイントに到達すると再度停止します。

- この時点で、Data メニューから Still Allocated Blocks...を選択して割り当ての報告を明確に要求し、ラジオボタンをクリックした後に Data パネルに項目を追加するため Data Still Allocated Blocks ダイアログ内の OK を押下して下さい。

本操作は NightView にプログラムを解析させ Data パネル内に Still Allocated Sets 項目を表示します。小さなプログラムでは本操作は重要でないように見えるかもしれませんが、より大きなプログラムに関してはかなりの時間を必要とする可能性があります。

- (必要であれば)下の興味のある項目の全体が見えるように列の見出しの間にある仕切りをクリックし右へドラッグして最初の列のサイズを変更して下さい。
- 必要であれば Still Allocated Sets 項目を展開して下さい。割り当てられたブロック・サイズと一致する割り当てのセット毎に一致する walkback と共に追加項目が表示されます。個々のセットの展開は、各割り当てだけでなく展開可能な個々のリークしたブロックの説明について表示する共通の walkback を提供します。
- サイズが 1048576 の allocated set 項目を展開し、続いてそれに関連する walkback 項目を展開して下さい。

  walkback は func3()関数によって割り当てられ、それは **app.c** の 162 行目の heap_thread()ルーチン内で alloc_ptr()の呼び出しにより起動された事を示している事に注意して下さい。

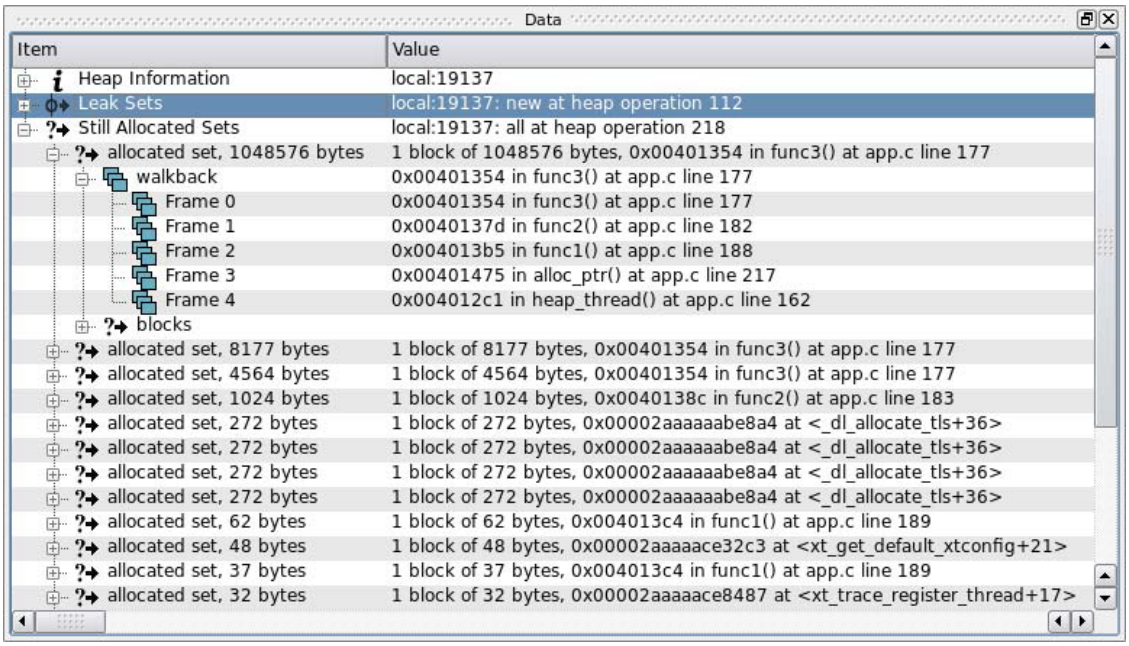


図 3-22. Still Allocated Blocks の表示

NOTE

Still Allocated Sets のデータはシステムにより異なります。上で示したように 1048576 byte の割り当てられたセットに集中して下さい。

NOTE

Data パネル内の殆どの項目とは異なり、プロセスが停止した時に `Still Allocated Sets` 項目は自動的に更新されません。記述はプログラム実行中のある時点でのリークのスナップショットであるため、追加のリークが発生したとしても変更されないままとなります。情報を更新するには、(Data メニューから Still Allocated Blocks...を選択して)別の割り当ての報告を要求して下さい。

ヒープ・デバッグの無効化

- ヒープ・デバッグに関する全てのオーバーヘッドを無効にするには以下のコマンドを実行して下さい：

```
heapdebug off
```

- Eventpoints パネル内のブレークポイントを右クリックして Delete を選択、または以下のコマンドを実行して 130 行目のブレークポイントを削除して下さい：

```
clear app.c:130
```

これでヒープ・デバッグに関するチュートリアルのトピックは終了です。

NightTrace の準備

本セクションは 3-23 ページの「トレースポイントの利用」セクション内の手順が完了していることを前提としています。そうではない場合、そのセクションに戻ってそれらの手順を終えてからここに戻ってきて下さい。

- Tools メニューを開いて NightTrace を選択した後、NightTrace Analyzer を選んで NightTrace を起動して下さい。

チュートリアルの残りのセクションで NightView は使用しませんが、実行ファイルにパッチを当てたトレースポイントを保持する必要があります。NightView からは切り離しますが、実行し続けて全てのパッチポイントとトレースポイントを保持します。

- コマンド・フィールドに以下を入力してプロセスを停止して下さい：

```
stop /protected
```

- Process メニューから Detach オプションを選択して下さい。
- NightView を終了するには File メニューから Exit NightView オプションを選択して下さい。

NOTE

通常は、NightView 内から開始されたプロセスは NightView 終了時に例えデタッチしたとしても kill されます。これはそれらを起動するために NightView で使用されるシェルが SIGHUP シグナルを送信するためです。本アプリケーションは SIGHUP を無視するので実行し続けることが出来ます。

終了 – NightView

これで NightStar RT チュートリアルの NightView の部は終了です。

4

NightTrace の利用

NightTrace はシングルおよびマルチプロセッサー・アプリケーションの動的な挙動を解析するためのグラフィカル・ツールです。NightTrace は複数の CPU または複数のシステム上で同時に実行しているプロセスからユーザー定義のアプリケーション・データ・イベントを記録することが可能です。NightTrace は個々のシステムコール、コンテキスト・スイッチ、マシーン例外、ページ・フォルト、割り込みのようなカーネル・イベントもまた記録することが可能です。アプリケーション・イベントとカーネル・イベントを併用することで、NightTrace はシステム全体の表示を同期して描画します。更には、NightTrace は多種多様な方法でそれらのイベントをユーザーがズーム、検索、フィルタリング、要約、解析することを可能にします。

NightTrace を使用するとユーザーは複数のユーザーおよびカーネル NightTrace デーモンを便利な場所から同時に管理することが可能となります。NightTrace は管理下にある任意のデーモンの実行を開始、停止、一時停止、再開する機能をユーザーに提供します。

NightTrace ユーザーは 1 つ以上のデーモンの定義で構成される「session」を定義および保存することが可能です。これらの定義はデーモンの収集モードと設定、デーモンの優先度とバインドする CPU、データ出力形式、特定のデーモンに記録されるトレース・イベントの種類を含みます。

NightTrace の起動

NightTrace は「NightView の利用」セクションの最後の手順で起動されました。

「NightView の利用」セクションをスキップした場合、チュートリアルの本セクションを開始(およびプロセスの実行を再開)する前にプログラムをビルド(1-4 ページの「プログラムのビルド」を参照)、NightView 内でプログラムを起動(**nview ./app**)、そして 3-23 ページの「トレースポイントの利用」の手順を実行して下さい。

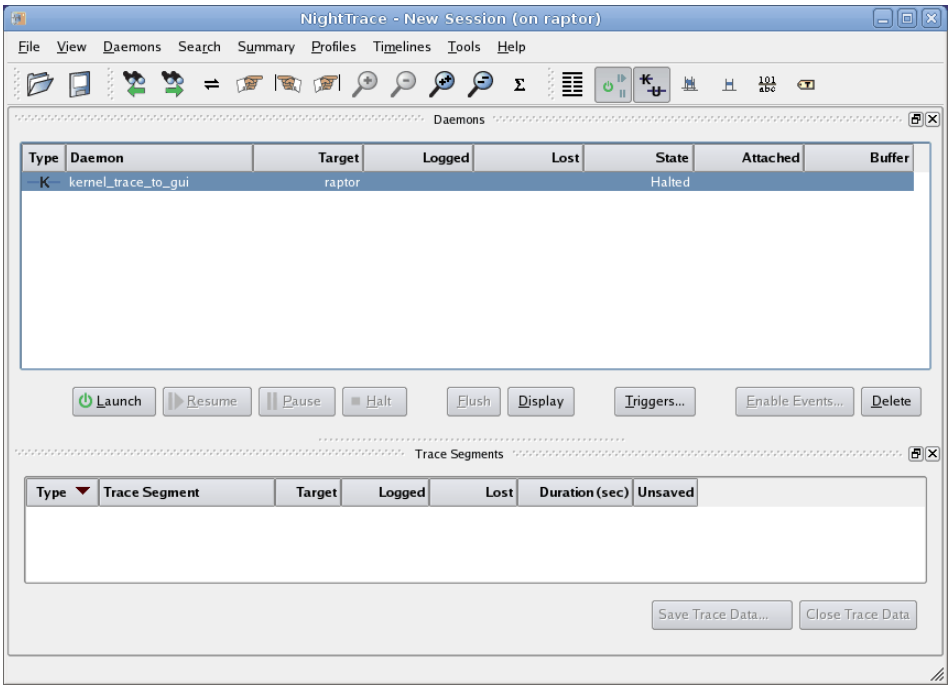


図 4-1. NightTrace メイン・ウィンドウ

メニュー・バーおよびツールバーの下の NightTrace メイン・ウィンドウの初期ページは以下の 2 つのパネルを含みます：

表 4-1. NightTrace Panels

Daemons	構成されたデーモンを表示します。
Trace Segments	各トレース・セグメント(連続したトレース・データの収集)を表示します。

Daemons パネルの統計値はデーモンとユーザー・アプリケーションとの間で使用される共有メモリ・バッファ内の生イベントの数およびデーモンによって NightTrace に書き込まれた生イベントの数を(Buffer と Logged の列の下にそれぞれ)示します。

Trace Segments パネルは現在 Events パネルとタイムラインを使って直ぐに解析可能な処理されたイベントの数を示します。

NOTE

Trace Segments パネル内に表示されるイベントの数は、通常は Daemons パネル内に表示されるイベントの数とは異なります。前者は処理されたイベントであるのに対して後者は生イベントとなります－処理されたイベントは大抵は複数の生イベントから構成されます。

ユーザー・デーモンの構成

NightTrace はユーザー・トレース・イベントを受け取るユーザー・デーモンをユーザーが構成することが可能です。

ユーザー・トレース・イベントは NightTrace API を使用するユーザー・アプリケーション、または NightView でそれらを挿入したプログラムにより生成されます。

これより **app** プログラムが出力するイベントを受け取るユーザー・デーモンを構成します。

実行中のアプリケーションを基にユーザー・デーモンを構成するには

- Daemons メニューから Import...メニュー・オプション、続いて Running Application オプションを選択して下さい。

  Import Daemon Definitions ダイアログが現れます：

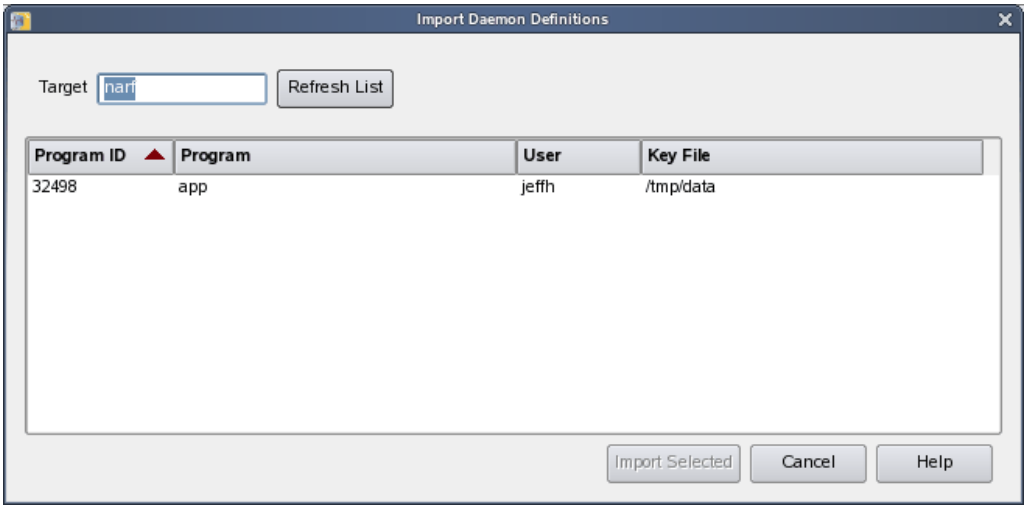


図 4-2. Import Daemon Definitions ダイアログ

Import Daemon Definitions ダイアログは NightTrace API コールを含んでいる実行中のユーザー・アプリケーションに基づきユーザーがデーモンの属性を定義することを可能にします。

- **app** アプリケーションに対応するエントリーを選択して下さい。
- Import Selected ボタンを押下して下さい。

Import Daemon Definitions ダイアログは閉じて、新しいユーザー・デーモンが生成され NightTrace メイン・ウィンドウの Daemon Control Area に追加されます。

ライブ・データを NightTrace GUI にストリーミング

NightTrace はその後の解析用にトレース・イベントを取り込んでそれらをファイルに格納、またはライブ解析用のグラフィカル・インターフェースに直接イベントを流すためのデーモンを使用することを可能にします。

デーモンはライブ・ストリーミング用に構成されています。

- NightTrace メイン・ウィンドウの Daemons パネルから app_data デーモンを選択して下さい。
- Launch ボタンを押下して下さい。
- Resume ボタンを押下して下さい。

デーモンは 3-23 ページの「トレースポイントの利用」で NightView を介して挿入したトレースポイントから **app** プログラムによって生成されているイベントを現在収集しています。

Daemons パネルでは、Buffer 列に表示されているイベントの数が変化し始めます。

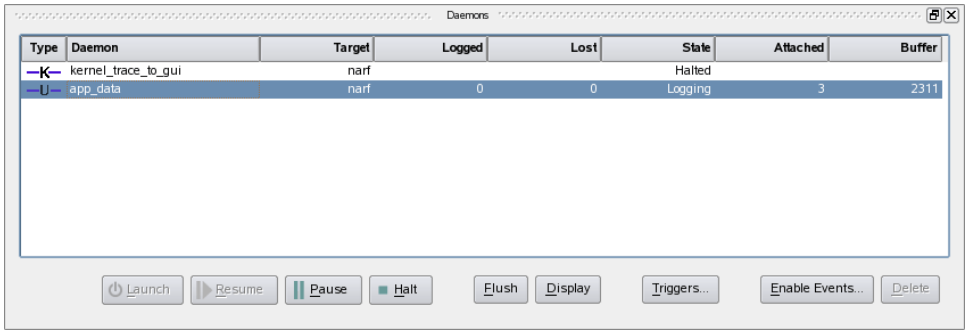


図 4-3. データのロギング

デーモンの停止

チュートリアルの NightTrace 部分は幾分長く、多くのユーザーにとっては新しい経験となる可能性があるので、メモリ消費を縮小するためにデーモンを停止します。

Buffer セルが少なくとも 20,000 となるまで待って、その後 Halt ボタンを押下してデーモンを停止して下さい。

NOTE

デーモンを停止する直前に例え Trace Segments パネルに表示されるイベント数が Daemon Control 領域に表示されるイベント数よりも小さかったとしても心配しないで下さい。後者は生イベント数を示しているのに対し Trace Segments パネルは処理されたイベント数を示しています－処理されたイベントは大抵は複数の生イベントから構成されます。

イベントの表示

ユーザー・トレース・データを検出した時に NightTrace メイン・ウィンドウ内にタブ・ページが生成されます。そのページは記録されたイベントのリストとそれらのイベントをグラフ表示するタイムラインを含む自動的にカスタマイズされたページとなります。

- Events パネルとそれらのイベントに関連するタイムラインを含んでいる新たに生成された app_data タブをクリックして下さい。

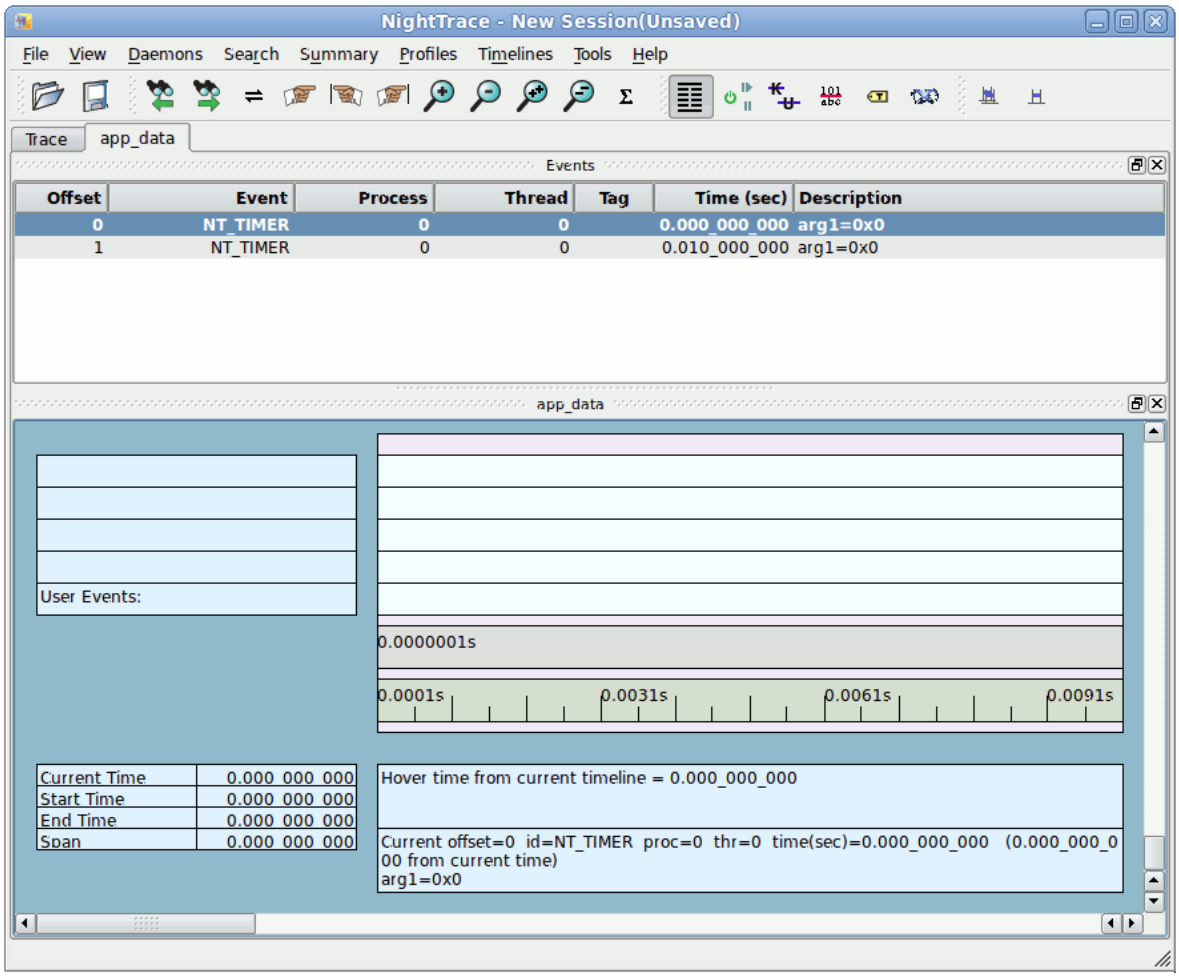


図 4-4. app_data ページ

NOTE

NightTrace を以前に使用したことがある場合、上図で示すようなデフォルトの色を無効にする Preferences を保存しているかもしれません。File メニューから Preferences を選択し、Timelines タブをクリック、Restore Defaults ボタンをクリック、Apply to existing timelines チェックボックスをチェックした後 Save をクリックすることで上図のような色を使用する Preferences に変更することが可能です。

最初は、パネルは大抵ブランクとなります。

即座に調べるためにタイムラインをズーム・アウトすることで強制的にデーモン・バッファからイベントをフラッシュさせ、セグメント領域内に出力ストリームを移動させることが可能です。

- タイムラインを含む表示領域のどこでも良いのでクリックして下さい。
- ズーム・アウトするには Up を押下して下さい。
- 完全にズーム・アウトするには Alt-Up を押下して下さい。

Events リストにはこれまでに記録されたイベントが追加され、タイムラインにはそれらのイベントをグラフィカルに表示します。どのイベントも見えない場合、再度 Alt-Up を押下して下さい。

- パネル下部の中央をクリックして下さい。

NightTrace タイムラインの利用

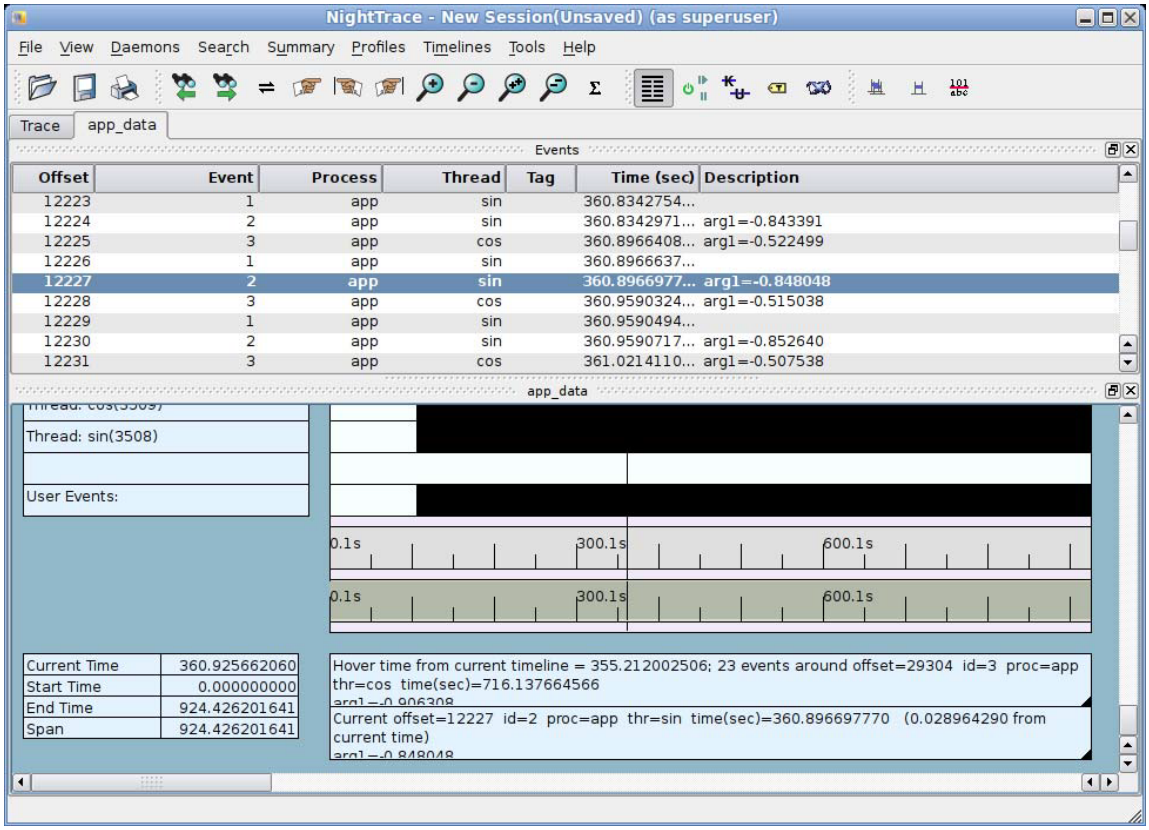


図 4-5. NightTrace タイムライン

タイムラインは静的および動的ラベルとイベントおよびステート・グラフを含んでいます。

デフォルトで、NightTrace は NightTrace API を通して自身を登録したスレッドを検出し、スレッド毎に個々のラベルやグラフを生成します。

本アプリケーションは 5 つのスレッドを含んでおり、その中の 4 つは特定のスレッド名称(heap, sin, cos, main)で自身を登録しています。個々のスレッドの行はそのスレッドに記録されたイベントのみを表示します。加えて、下のほうに全てのスレッドに関するイベントを示すユーザー・イベント・グラフがあります。

NOTE

お手元のタイムラインにはブランクのラベルとグラフが表示されるでしょう。これらはどのイベントも記録していない main および heap スレッドに関するラベルとグラフとなります。main スレッドにより少なくとも 1 つのイベントが記録されるまでラベルの中身は表示されません。

全てがブランクのラベルである場合、恐らく前述の手順で指示されたようにタイムラインの中ほどをクリックしていません。

「NightView の利用」セクション内の 3-23 ページの「トレースポイントの利用」において、トレースポイントを sine と cosine スレッドに挿入しており、それらは自身を「sin」および「cos」として登録しています。

ズーム

グラフ内の各縦線は少なくとも 1 つのイベントを表しています。詳細レベルを調整するのにズーム・インおよびズーム・アウトすることが可能です。

- タイムライン内のどこでも良いので左クリックして下さい。
- グラフ内の個々の線が見えるまで繰り返し Down キーを押下して下さい。
- ズーム・アウトするには Up キーを押下して下さい。
- マウス・ホイールがある場合、ズーム・イン/アウトするにはホイールを前後に動かして下さい。

垂直破線は現在のタイムラインで Events パネル内の強調表示されたイベントと関係があります。

表示領域でマウスを左クリックすると現在のタイムラインが移動します。タイムラインの下の Event Detail 領域内の情報は、現在のタイムラインの左側に最も近いイベントを反映して変化します。

間隔の移動

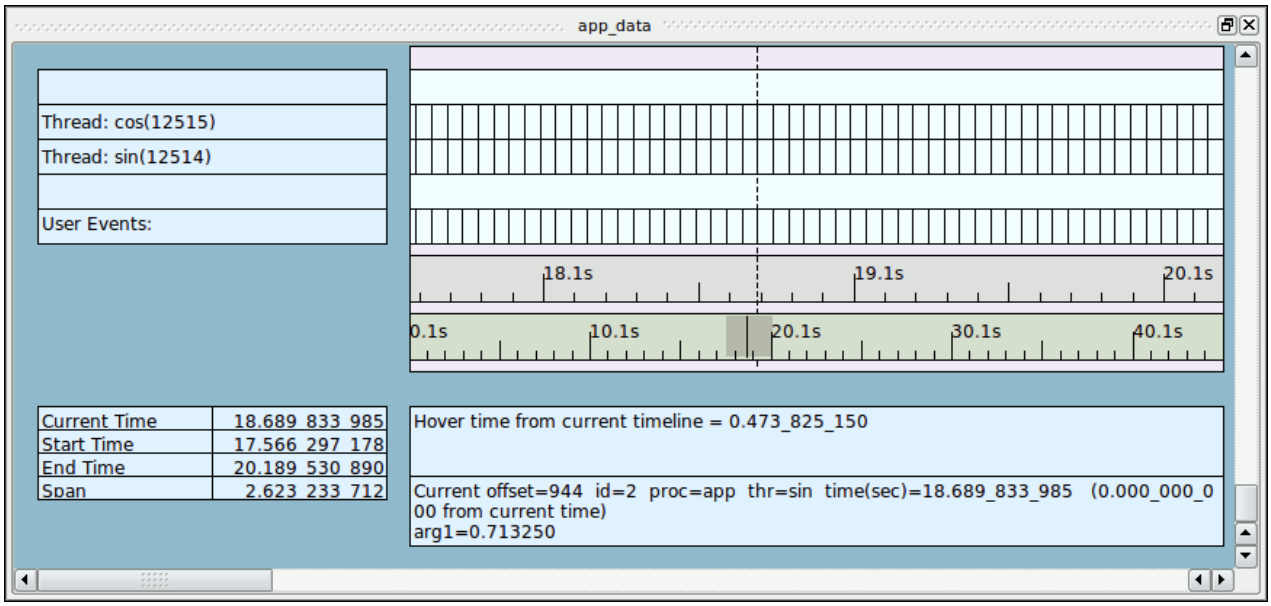


図 4-6. タイムライン間隔パネル

デフォルトで、各タイムラインは 2 つのルーラー行を持っておりイベント・グラフの下とパネル下部の説明ボックスの上に配置されています。これらは時間の間隔を表す数字と記号を含んでいます。

上のルーラーは現在表示されている時間軸を示します。

下のルーラーは現在表示することが可能な全てのデータに関する時間軸を示します。このルーラーをコントロール・ルーラーと呼び、その中に灰色の領域があります。灰色の領域は現在パネルに表示されている時間軸全体の量を表しています。従ってズーム・インすると灰色の領域の幅が縮小し、ズーム・アウトすると逆の効果が得られます。

NOTE

灰色の領域が見えない場合は、見えるまでズーム・アウトして下さい。

タイムライン全体の中を移動する方法がいくつかあります。

- Right キーを押下

  これは現在のタイムラインを次のイベントへ進めます。極端にズーム・アウトするとタイムラインが移動したことに気付かない可能性があります。この場合、タイムラインの移動が見えるようになるまでズーム・インまたは Right キーを押し続けて下さい。

  あるいは、Left キーを押下して現在のタイムラインを前のイベントへ移動します。

- Ctrl+Right を押下

  これは表示された間隔をデフォルトでセクションの右側へ 25%移動します。セクションとはその間隔内に現在見えている時間となります。どのようにコントロール・ルーラの灰色領域が動くのかを注視して下さい。

  あるいは、Ctrl+Left の押下で左へ 1 セクション移動します。

- 灰色の領域とコントロール・ルーラーの右手部分との中間をクリックして下さい。

  コントロール・ルーラー内のどこでもクリックすると現在のズーム設定で選択された時間の中央に移動します。

  従って、データ・セットの先頭または末端に移動するには、コントロール・ルーラーの先頭または終端をクリックします。

テキスト分析用イベント・パネルの利用

Events

Offset	Event	Process	Thread	Tag	Time (sec)	Description
941	2	app	sin		18.639_737_052	arg1=0.707107
942	3	app	cos		18.689_830_607	arg1=0.713250
943	1	app	sin		18.689_830_614	
944	2	app	sin		18.689_833_985	arg1=0.713250
945	1	app	sin		18.739_897_536	
946	3	app	cos		18.739_897_556	arg1=0.707107
947	2	app	sin		18.739_900_519	arg1=0.719340

図 4-7. Events パネル

Events パネルに表示されるイベントはタイムラインに表示されるイベントと同期します。強調表示されたイベントは現在のタイムラインを示します。

- Events パネル内の行をクリックして下さい。
- 次のイベントに進むには Down キーを押下して下さい。
- 前のイベントに進むには Up キーを押下して下さい。

イベントが選択されるまたは現在のイベント行が移動するとすぐに右側のタイムライン下にある Event Detail 領域は(利用可能であれば)イベントに関する追加情報を表示します。

- 次のイベント・セットに進むには PageDown を押下して下さい。
- 前のセットに移動するには PageUp を押下して下さい。

これらのアクションは Events パネル内に表示することが可能なイベントの数によって現在のタイムラインだけが移動します。

イベント説明のカスタマイズ

NightView 内の **tracepoint** コマンドを使って記録したイベント値はイベント ID1～3 でした。これらのイベントの説明をカスタマイズすることが可能です。

- Events パネル内のイベントの Code が 1 と表示している行をクリックして下さい。
- その行を右クリックしてコンテキストメニューから Edit Current Event Description...を選択して下さい。

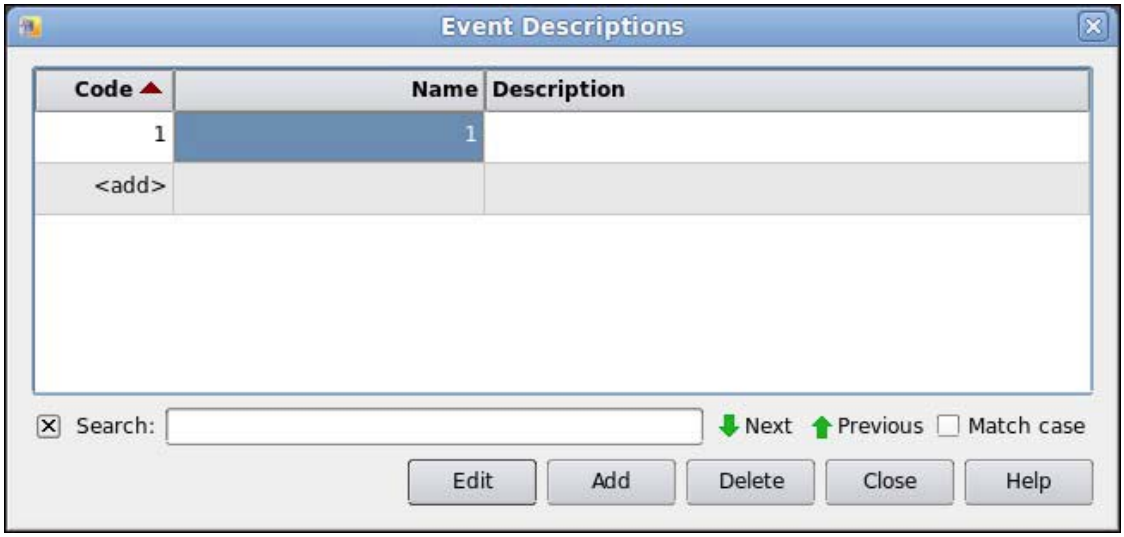


図 4-8. イベント説明追加ダイアログ

- 選択された Name フィールドをダブルクリックして

  cycle_start

  を Name フィールドに入力して下さい。

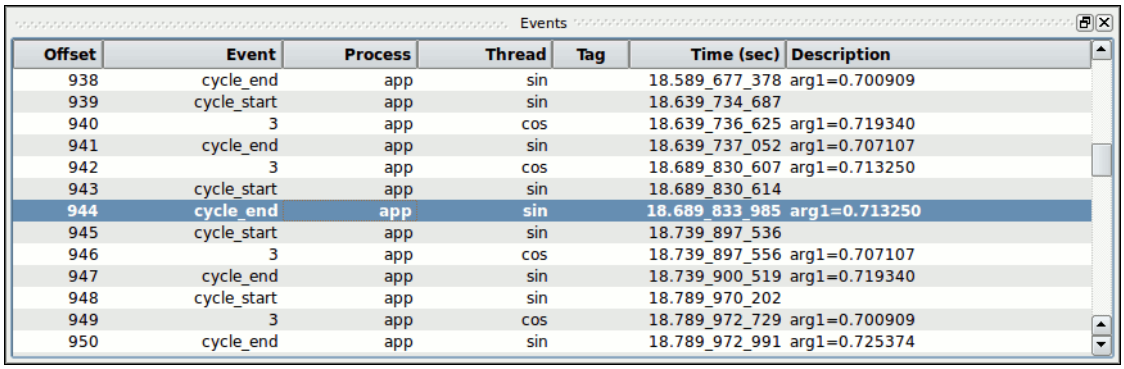
- <add>が含まれるテーブル・セルをダブルクリックして下さい。
- Code セルに 2 を入力して下さい。
- その行の Name セル内をクリックして下さい。
- 以下を

  cycle_end

  Name フィールドに入力して下さい。

- Close ボタンを押して下さい。

Events パネル内のイベントの説明は、現在それらに割り当てたテキスト識別子に対応しています。



イベント・リストの検索

特定のイベントの発生またはイベントに関連する状況やその引数を検索するために NightTrace の検索機能を利用することが可能です。

- Search メニューから Power Search...メニュー項目を選択して下さい。

  定義されたプロファイルのリスト(現在は空)および新しいプロファイルを定義または既存のものを編集することが可能ないくつかのフィールドを含んだダイアログが表示されます：

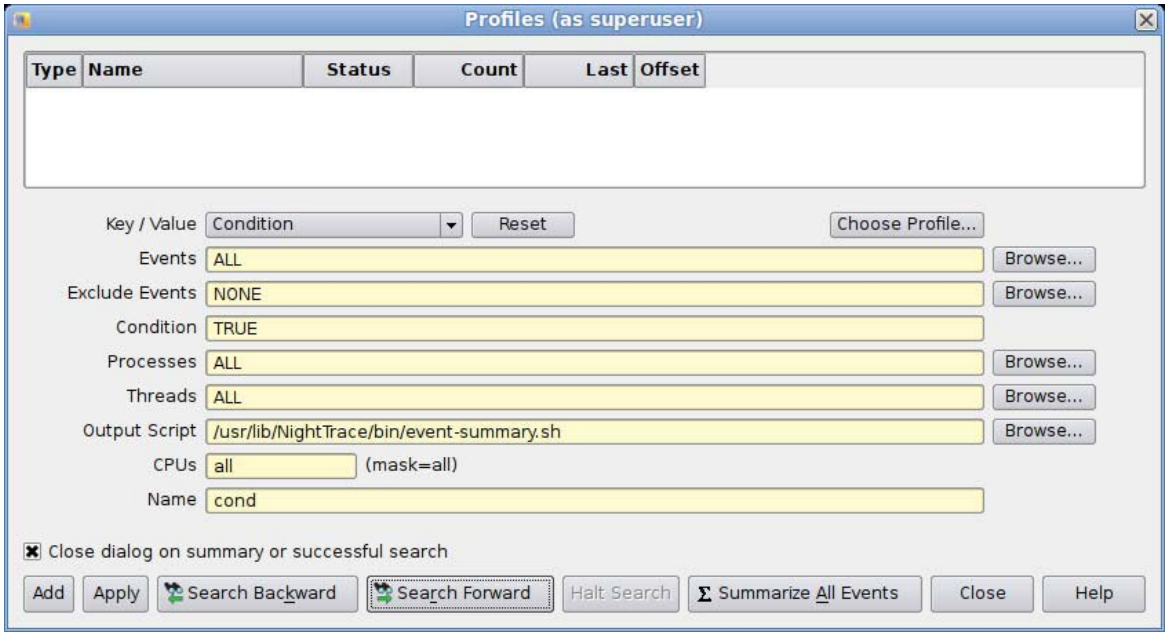


図 4-9. Profiles ダイアログを使った検索

ダイアログの上部はプロファイル・リストで、以前定義した全てのプロファイルを表示します。

ダイアログの他の部分はプロファイルを定義または編集するためのメカニズム、およびそれらに従う一般的な動作を提供します。

- Events フィールドの右側の Browse...ボタンを押して下さい

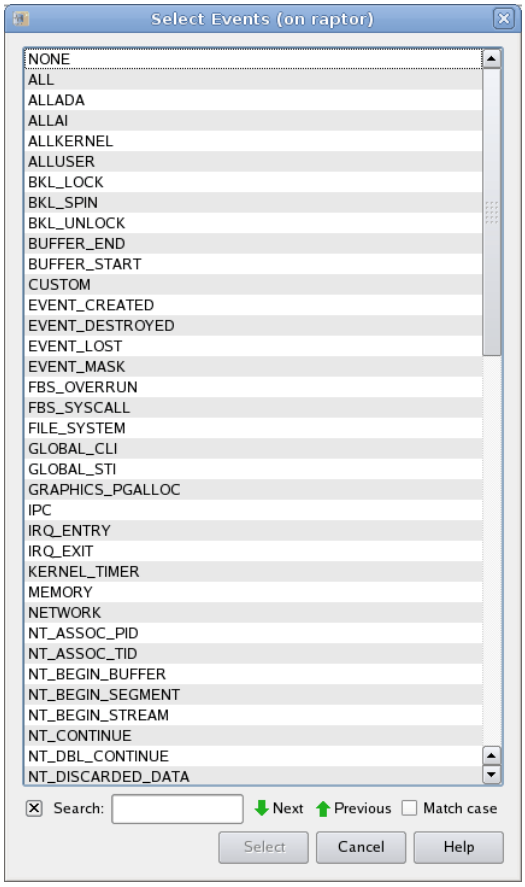


図 4-10. イベント閲覧ダイアログ

- Search テキスト・フィールドをクリックして **cycle** と入力して下さい。その単語を含む最初のイベント名称が表示されます。イベント・リスト内で **cycle_end** が選択されていることを確認、または選択されるまで Next アイコンを押して下さい。その後 Select ボタンを押して下さい。
- Profile パネルの Condition テキスト・フィールドに以下の文字列を入力して下さい：

```
arg_dbl > 0.8
```

- Name テキスト・フィールドに以下の文字列を入力して下さい：

```
obtuse
```

- Profiles パネルの Add ボタンを押して下さい。

  obtuse を呼んだプロファイルが現在定義されダイアログの Profile Status List 領域内に現れます。

- Profiles ダイアログの下部にある Search Forward ボタンを押して下さい。

  現在のタイムラインが検索基準に一致した最初のイベントへ移動されますが、これはサイン値が 0.8 を超えた時のサイクルの最後となります。

NOTE

NightTrace が利用可能なデータセットの最後に達して先頭で検索を再開すべきかどうかを尋ねるポップアップ・ダイアログが表示されたら、OK を押して下さい。

- NightTrace は Profiles ダイアログを閉じて以下に示すようにタイムラインに戻ります：

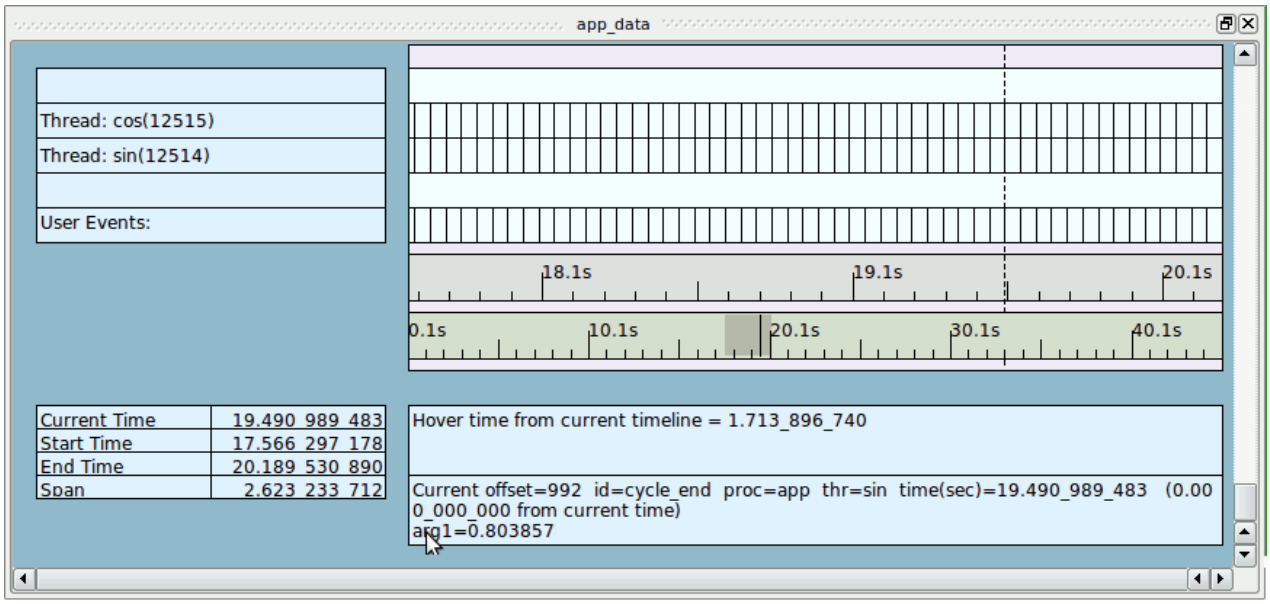
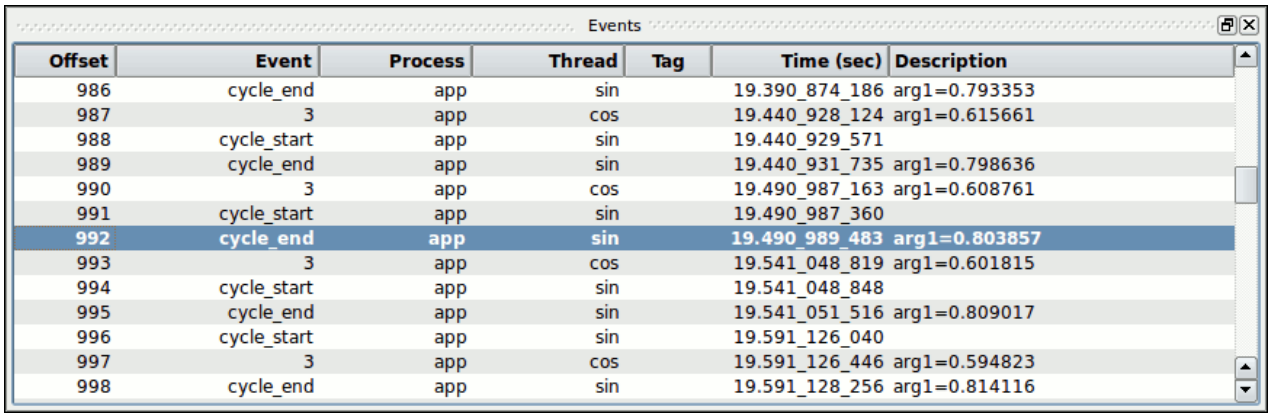


図 4-11. 検索後のタイムライン・パネル

- Events パネルにリストアップされた現在のイベントが 0.8 を超える値を含む arg1 を表示していることを確認して下さい。



Offset	Event	Process	Thread	Tag	Time (sec)	Description
986	cycle_end	app	sin		19.390_874_186	arg1=0.793353
987	3	app	cos		19.440_928_124	arg1=0.615661
988	cycle_start	app	sin		19.440_929_571	
989	cycle_end	app	sin		19.440_931_735	arg1=0.798636
990	3	app	cos		19.490_987_163	arg1=0.608761
991	cycle_start	app	sin		19.490_987_360	
992	cycle_end	app	sin		19.490_989_483	arg1=0.803857
993	3	app	cos		19.541_048_819	arg1=0.601815
994	cycle_start	app	sin		19.541_048_848	
995	cycle_end	app	sin		19.541_051_516	arg1=0.809017
996	cycle_start	app	sin		19.591_126_040	
997	3	app	cos		19.591_126_446	arg1=0.594823
998	cycle_end	app	sin		19.591_128_256	arg1=0.814116

図 4-12. 検索後の Events パネル

同様にタイムラインはパネルの下部にある Event Detail 領域内に現在のイベントを表示します。

- マウス・カーソルをパネル下部のイベント説明ボックスに移動して動かさずにそこでそのままにしてください。

ツールチップがイベントの完全な説明と共に現れます。これは表示された説明がタイムライン・ページ上の説明ボックスのサイズが原因で不完全な時に便利です。

ステートの利用

個々のイベントの表示に加えて、NightTrace はステートを表示することも可能です。

- ツールバー上のどちらかの Profiles アイコンをクリックして下さい 

Profiles ダイアログは既に定義したプロファイルが選択された状態で表示されます。

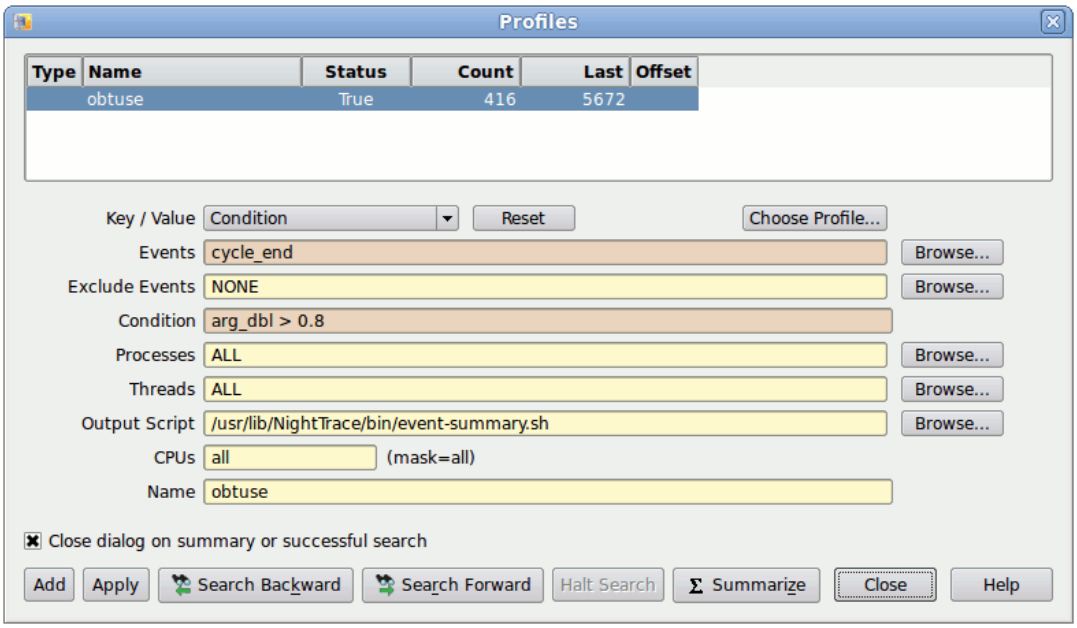


図 4-13. obtuse プロファイルが選択された Profiles ダイアログ

- Reset ボタンを押して下さい。
- Key / Value オプション・リストから State を選択して下さい。
- 以下を

  ```
  cycle_start
  ```

  Start Events テキスト・フィールドに入力して下さい。

- 以下を

  ```
  cycle_end
  ```

  End Events テキスト・フィールドに入力して下さい。

- 以下を

  ```
  sin
  ```

  Threads テキスト・フィールドに入力して下さい。

- 以下を

  ```
  sine
  ```

  Name テキスト・フィールドに入力して下さい。

- Add ボタンを押して下さい。

- ダイアログを閉じて下さい。

sine という名前のステートが定義されてディスプレイ・ページのグラフに発生頻度を表示することが可能となります。

- 表示領域のどこでも右クリックしてコンテキストメニューから Edit Mode を選択または Ctrl-E を押下して*編集モード*に入ります。

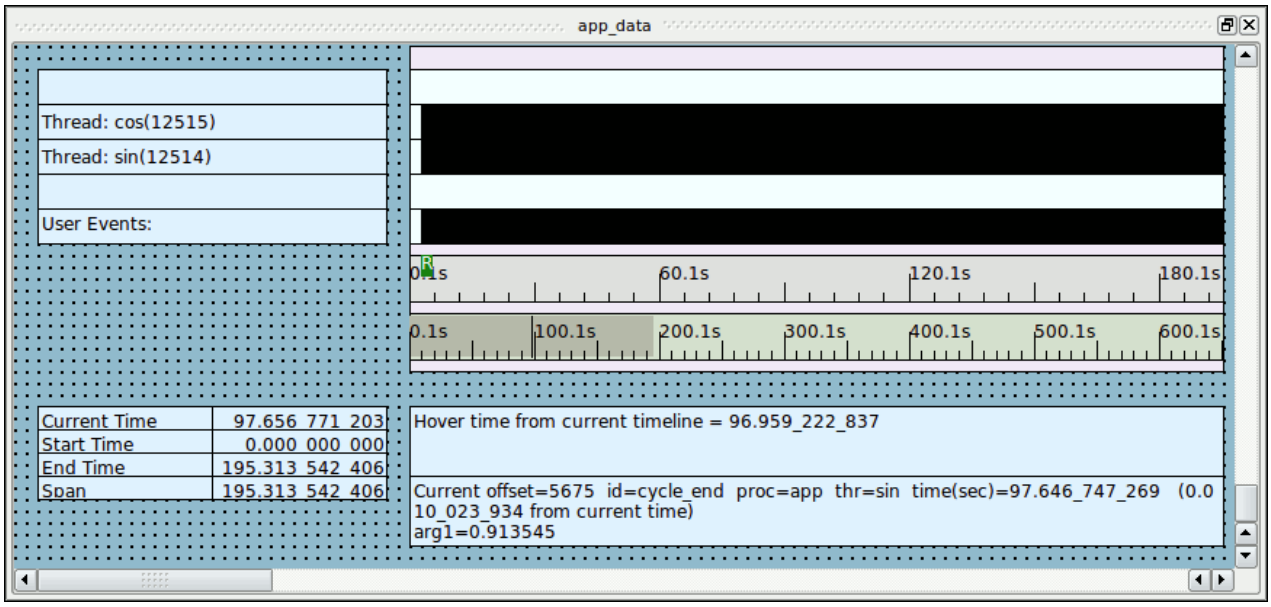


図 4-14. タイムラインの編集

- 「Thread: sin」のラベルの付いた行に紐づいたグラフをダブルクリックして下さい。グラフとは「Thread: sin」のラベルと並んでいる大きなグラフ領域の内側にイベントを表す縦線を伴う行のことです。

Edit State Graph Profile ダイアログが以下に示すように表示されます：

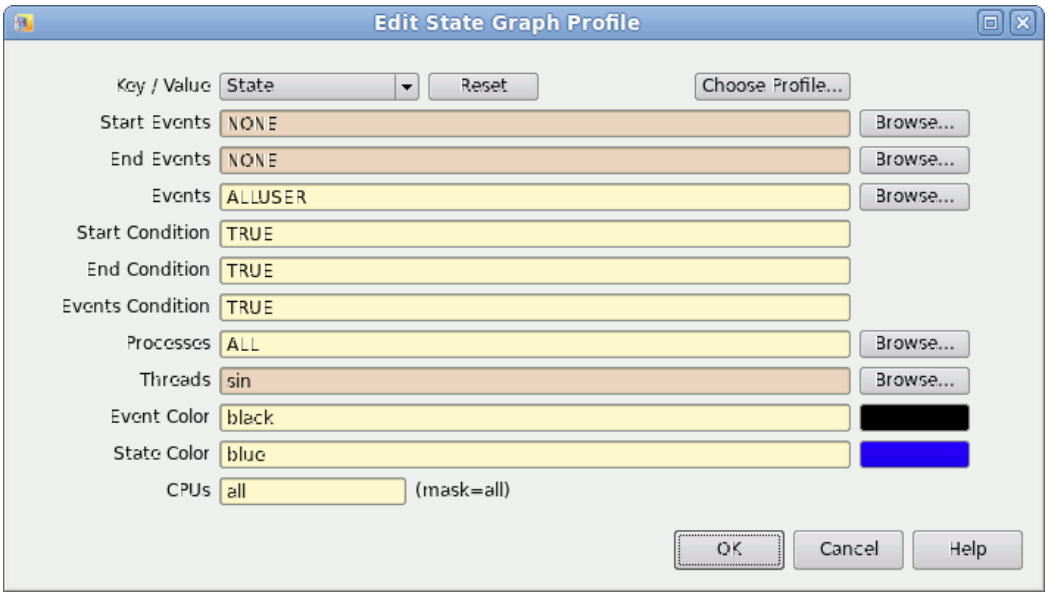


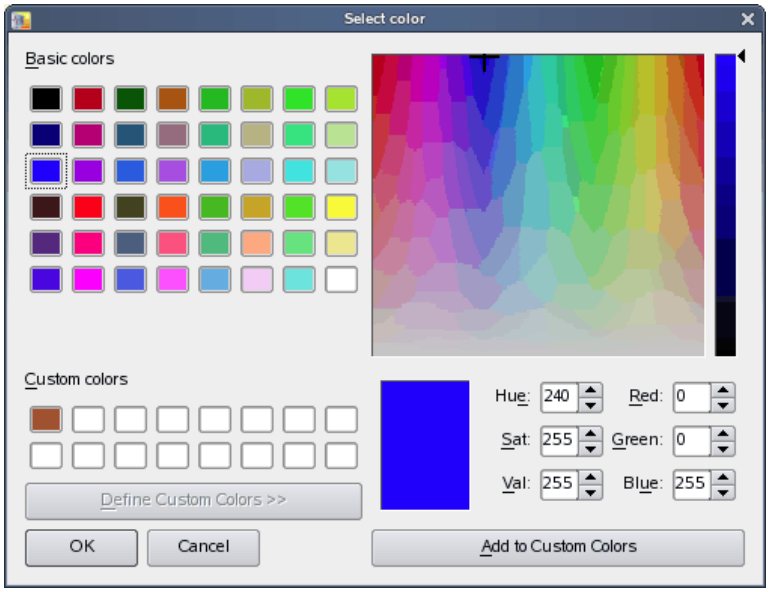
図 4-15. Edit State Graph Profile ダイアログ

- Key / Value オプション・リストから State を選択して下さい。
- Choose Profile...ボタンを押して下さい。

  Choose Profile ダイアログが表示されます。

- リストから sine ステートを選択して下さい。
- Import by reference チェックボックスにチェックがついていることを確認して下さい。
- Select を押して下さい。

- State Color のラベルが付いた行の右側にある色のついたボタンをクリックして下さい。Select color ダイアログが表示されます。



- Select color ダイアログ内で好みの色を選択して OK を押して下さい。
- Edit State Graph Profile ダイアログで OK を押して下さい。
- *表示モード*へ戻るには、表示領域のどこでも右クリックしてポップアップ・メニューから Edit Mode を選択または Ctrl-E を押下して下さい。

今、グラフはステート・グラフの下部の実線の棒として sine ステートを表示するよう構成されました。イベントはグラフの縦方向の高さ全体に及ぶ縦の黒い線として表示されたままになります。

ディスプレイ・ページは著しく変わっていない可能性があります。これは cycle_start と cycle_end のイベントが現在のズーム設定ではそれらを見分けられない時間でとても近接して発生しているためです。

- ステート・グラフの中央をクリックして下さい。
- 2 つのイベントが見分けられるまでマウス・ホイールを使って、またはツールバー上の Zoom In アイコンもしくは Down キーを使ってズーム・インするとステート・バーが表示されます。

ズーム・インした時に現在のタイムラインを再調整する必要があるかもしれません。

NOTE

Down キーが効かない場合、Num Lock キーを押下して再試行して下さい。

NOTE

ステートはズーム・レベルのスケールと比較してとても小さい一部のズーム・レベルにおいて消える可能性があります。その場合は、ただズーム・インし続ければ再び現れます。

下図は sine ステートのインスタンスを示しています。

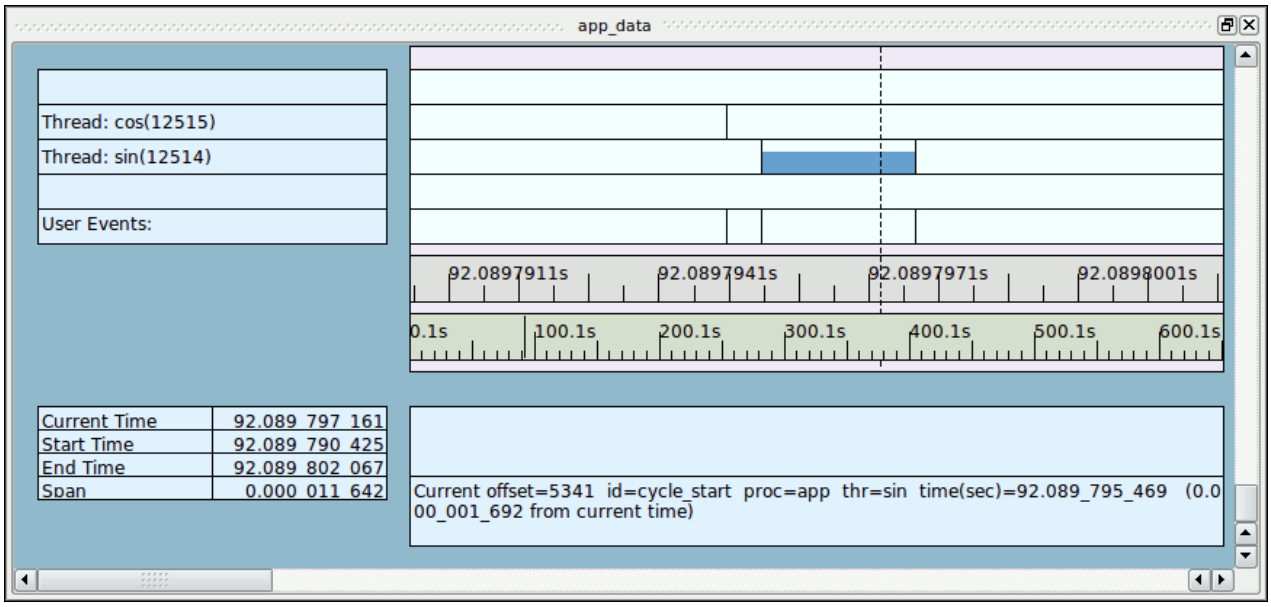


図 4-16. タイムライン内の sine ステート

NOTE

ステートが見えない場合、4-16 ページの「ステートの利用」で説明したように Profiles パネルで sine のプロファイルの定義を再確認して下さい。

cos 行で見られる動作状況は上で示されたものとは異なるかもしれません。

ステートの間隔の表示

最後に完了したステートの間隔はデータ・ボックスを介して表示させることが可能です。

- app_data ラベルが付いたページの表示領域のどこでも右クリックしてポップアップ・メニューから Edit Mode を選択または Ctrl-E を押下して*編集モード*に入ります。
- グリッド(背景に黒い点のある領域)のどこでも右クリックしてポップアップ・メニューから Add Data Box オプションを選択して下さい。

カーソルは+の記号に変わります。

- 新しいデータ・ボックスの概要を生成するには、左マウス・ボタンを使ってグリッド上(現在表示されているグラフまたはデータ・ボックスの外側、つまりドット付きグリッドを表示する背景の空き領域)のディスプレイ・ページ左側にある空白の領域をクリックしてマウスをドラッグし、マウス・ボタンを離して下さい。
- データ・ボックスをダブルクリックして下さい。Edit Data Box Profile ダイアログが現れます。
- Output フィールドに以下を入力して下さい：

```
format ("cycle = %f ms", state_dur(sine)*1000.0)
```

- OK ボタンを押して下さい
- *表示モード*へ戻るには、表示領域のどこでも右クリックしてポップアップ・メニューから Edit Mode を選択または Ctrl-E を押下して下さい。

データ・ボックスは現在ミリ秒で sine ステートの最後に完了したインスタンスの長さを(現在時刻に関する視覚インジケーターと共に)表示しています。

サマリー情報の生成

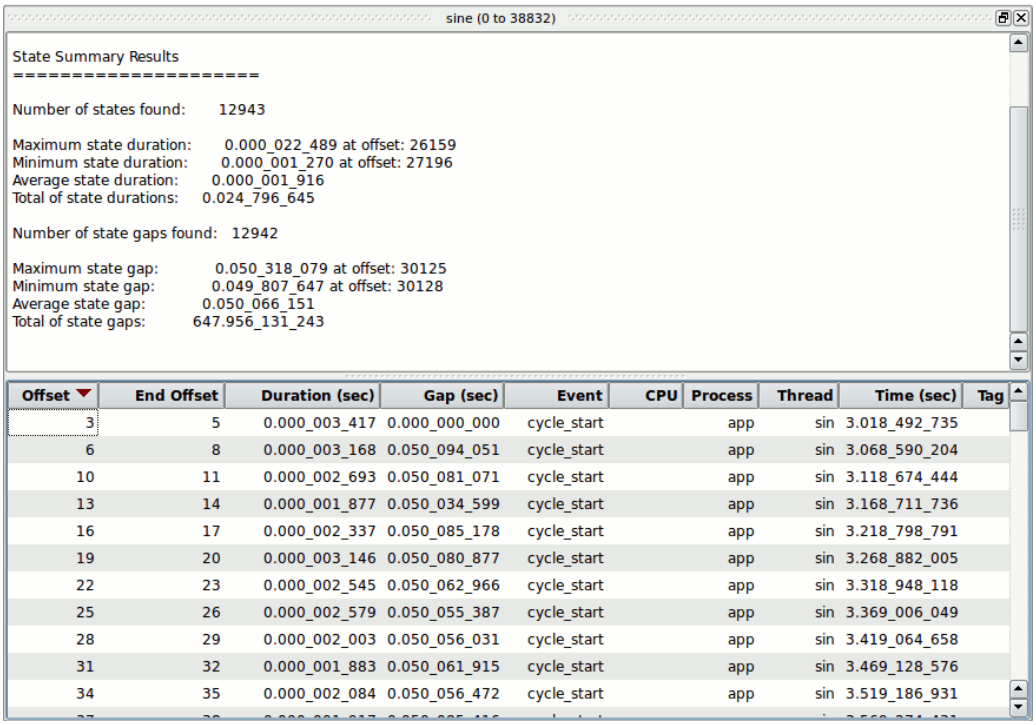
特定のイベントやステートに関する詳細な情報の取得に加えて、サマリー情報は簡単に生成されます。

- Summary メニューから Change Summary Profile...メニュー項目を選択して下さい。
- Profile Status List テーブルに表示されたプロファイルの一覧から sine ステートに一致するプロファイルを選択して下さい。

sine プロファイルは既に選択されているかもしれません。ダイアログの下部近くにある Name テキスト領域に表示されているプロファイル名称を見ることでそれを確かめることが可能です。

- Summarize All Events ボタンを押して下さい。

新しいページが生成されてサマリーの結果を表示します。



Offset	End Offset	Duration (sec)	Gap (sec)	Event	CPU	Process	Thread	Time (sec)	Tag
3	5	0.000_003_417	0.000_000_000	cycle_start		app	sin	3.018_492_735	
6	8	0.000_003_168	0.050_094_051	cycle_start		app	sin	3.068_590_204	
10	11	0.000_002_693	0.050_081_071	cycle_start		app	sin	3.118_674_444	
13	14	0.000_001_877	0.050_034_599	cycle_start		app	sin	3.168_711_736	
16	17	0.000_002_337	0.050_085_178	cycle_start		app	sin	3.218_798_791	
19	20	0.000_003_146	0.050_080_877	cycle_start		app	sin	3.268_882_005	
22	23	0.000_002_545	0.050_062_966	cycle_start		app	sin	3.318_948_118	
25	26	0.000_002_579	0.050_055_387	cycle_start		app	sin	3.369_006_049	
28	29	0.000_002_003	0.050_056_031	cycle_start		app	sin	3.419_064_658	
31	32	0.000_001_883	0.050_061_915	cycle_start		app	sin	3.469_128_576	
34	35	0.000_002_084	0.050_056_472	cycle_start		app	sin	3.519_186_931	

図 4-17. サマリー結果ページ

サマリー結果ページはステートの開始と終了のオフセット、ステートの持続時間、ステートとその直前の実現値との間のギャップを含む多くの情報の列を提供します。どのようにリストをソートさせるかを制御するためカラム・ヘッダをクリックすることが可能です。

リスト内の行をダブルクリックするとステートのそのインスタンスの先頭に現在のタイムラインの位置を合わせてその場所にタグを生成します。

最長のステート持続時間のインスタンスへ移動するには、次のようにして下さい：

- ソート・キーとして持続期間を選択するには Duration ヘッダーをクリックして下さい。
- ソート順が最大から最小となるまで Duration ヘッダーをクリックして下さい。
- 最長の持続期間を持つステートのインスタンスが行の一番上に表示されます。
- その行をクリックして下さい。

Events や Timeline のパネルで見られるとおり、現在のタイムラインはそのステートのインスタンスへ移動します。

最小と最大のステートの実現値は大抵は興味深いものです。また一方、ステートの持続時間のグラフィック表示はより啓発的になる可能性もあります。

- Summary メニューから Graph State Durations...オプションを選択して下さい。
- ダイアログ内の標準偏差値を **0** に変更して下さい。
- OK ボタンを押して下さい。

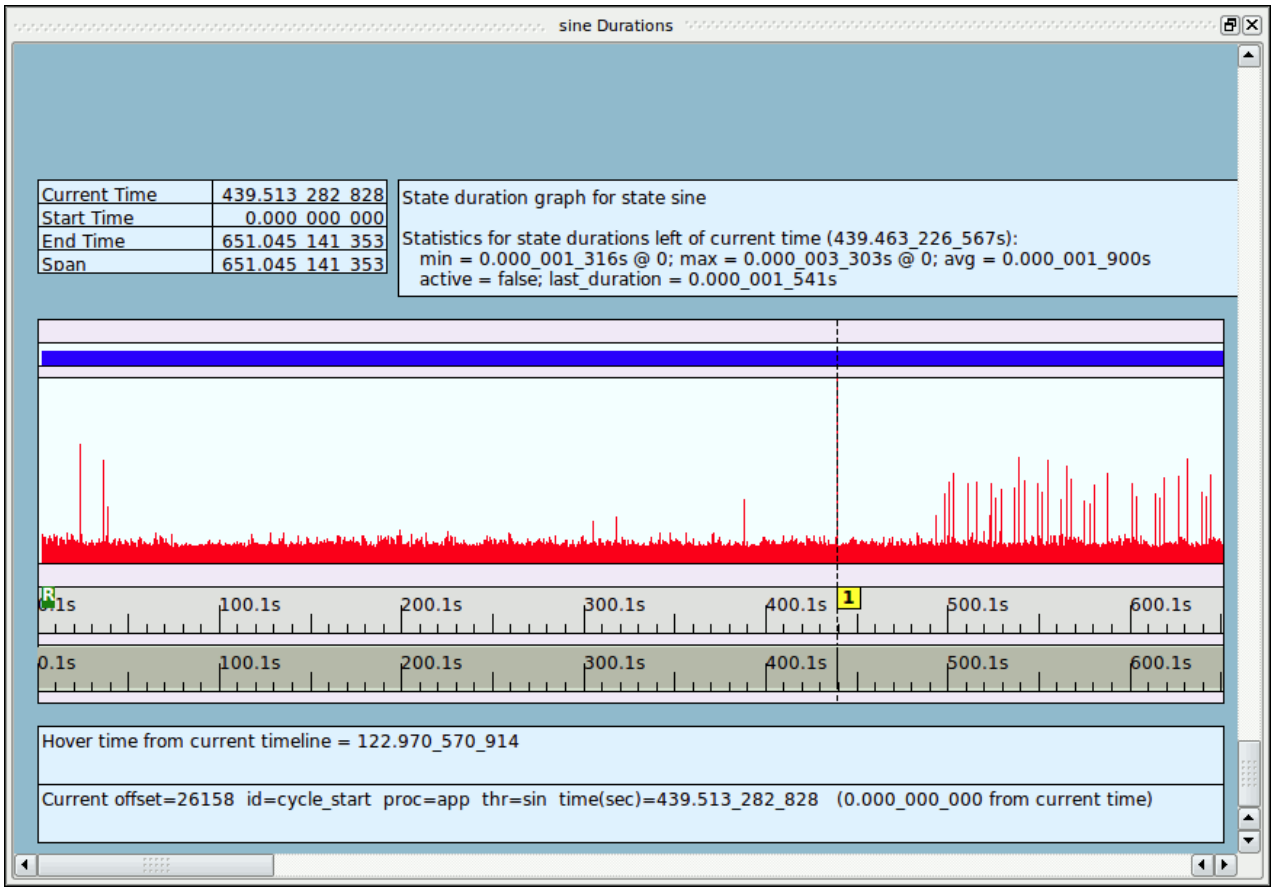


図 4-18. サマリー・グラフ

新しいページがサマリー・グラフおよびステートのインスタンスに関するテキスト記述と共に生成されます

青色の行はステートの個々のインスタンスを示しています。青の棒線が単独の線のように見える場合、個々のインスタンスが表示されるまでズーム・インして下さい。

- Alt+Up を押下して完全にズーム・アウトして下さい。

  データ・グラフは青いステート・インジケータの行の真下にある背の高い行に表示されます。

  各々の赤い線はステートのインスタンスの持続期間を示しています。

  時にはステートの単一の実現値が他の殆どの実現値よりも非常に長い場合があります。そのような場合、その詳細は不明瞭です。

  異なる標準偏差インデックスを使ってページを再構築することが可能です。

- サマリーを含むタブを右クリックして Delete Current Page をクリックして下さい。
- Summary メニューから Graphs State Durations を選択し標準偏差を要求するダイアログに値 1 を入力して下さい

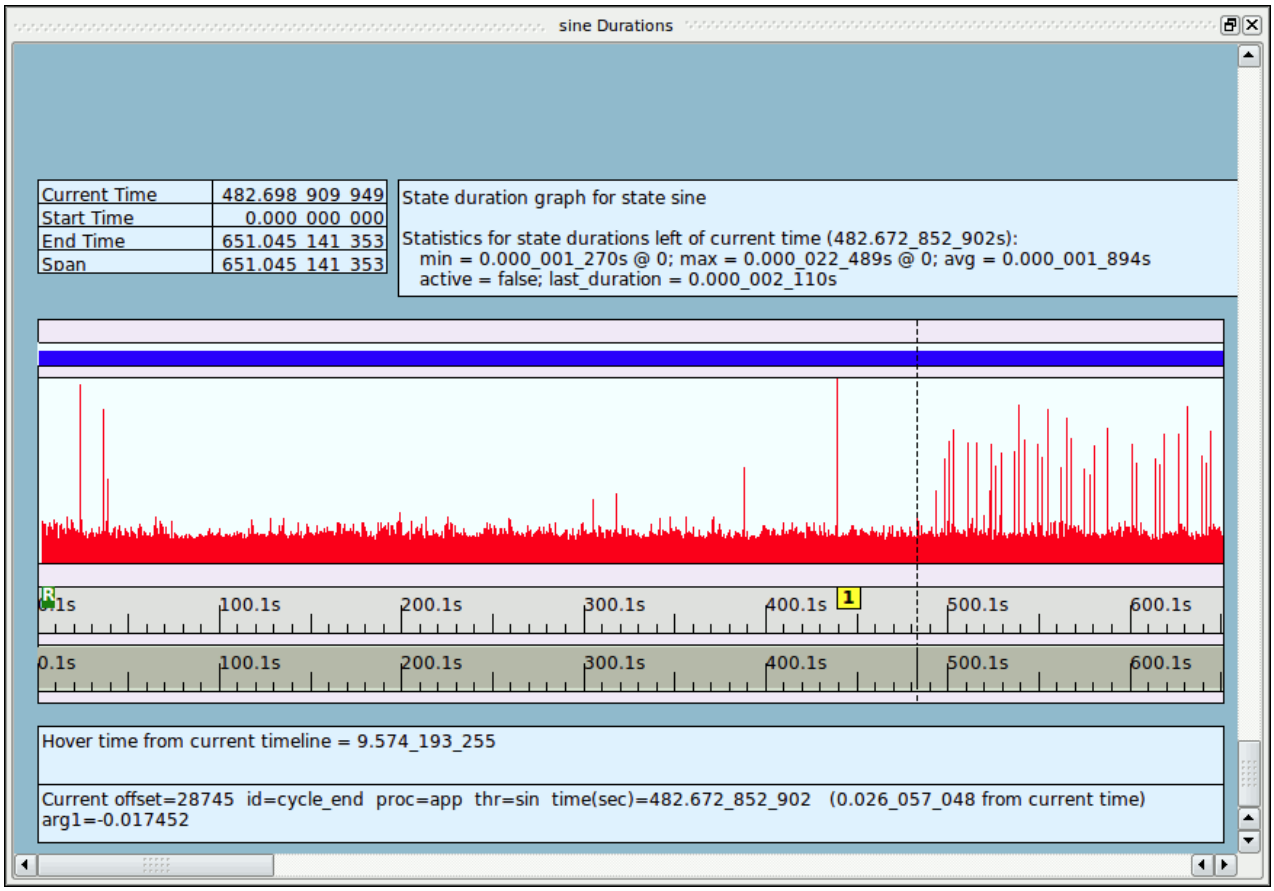


図 4-19. 変更されたステート間隔グラフ

グラフはより詳細をすぐに表示します。データ・グラフ内の現在のタイムラインは全タイムラインのタイムラインおよび Events パネルとリンクしています。グラフのどこでもクリックするとそれら全てのパネル内のタイムラインが移動します。

NOTE

様々な要素によりますが、標準偏差に 1 を設定すると実際には逆効果で更に詳細が不明瞭となる可能性があります。最も便利なものが見つかるまで上記手順を使って標準偏差係数(実際には係数 0 となる可能性があります)を試して下さい。

データ・グラフの定義

- app_data タイムライン・ページのタブをクリックして最上位にしてください。
- Events パネルのタイトルバーの右上部分にある閉じるボタンをクリックしてパネルを削除して下さい。

イベントを含む個々の行を含んでいる領域をグラフ・コンテナーと呼びます。これは最上部と最下部で見ることが出来るピンクの背景を持っています。

- app_data のラベルが付いた表示パネルのどこでも右クリックしてポップアップ・メニューから Edit Mode を選択または Ctrl-E を押下して*編集モード*に入ります。
- **graph container** 最上部の水平線中央をクリックして下さい。
- マウス・カーソルを動かして列最上部の水平線中央に移動して下さい。
- カーソルが上下を指す 2 つの矢印に変わったら、データ・グラフ用の空間を作るためにグラフ・コンテナーの上部境界をクリックして上のほうへドラッグして下さい。

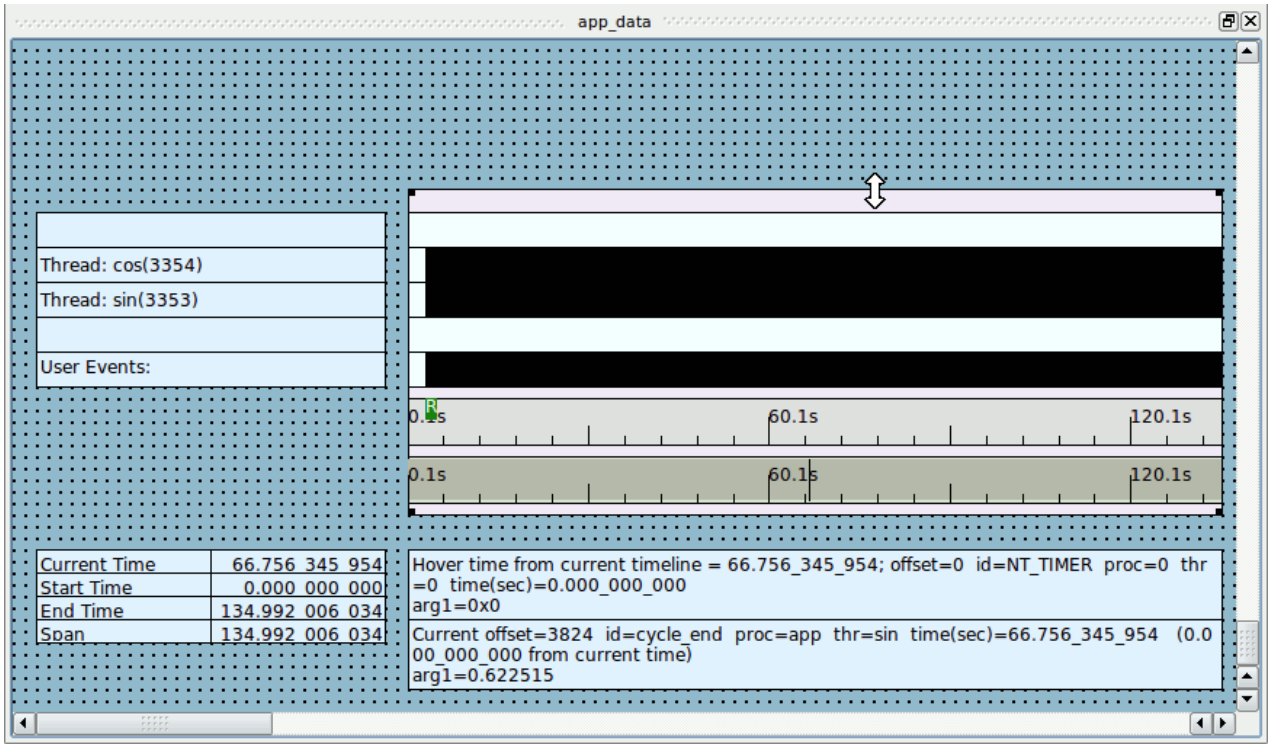


図 4-20. 編集モードのタイムライン

- 十分な空間(縦方向に約 1 インチ以上)が出来たらマウス・ボタンを離して下さい。
- グラフ・コンテナー最上部の水平線をクリックして下さい。

- グラフ・コンテナーの内側を右クリックしてポップアップ・メニューから Add to Selected Graph Container を選択、続いてサブメニューから Data Graph を選択して下さい。

  カーソルは十字記号に変わります。

- 前述の手順で生成された空間をクリックして下さい。

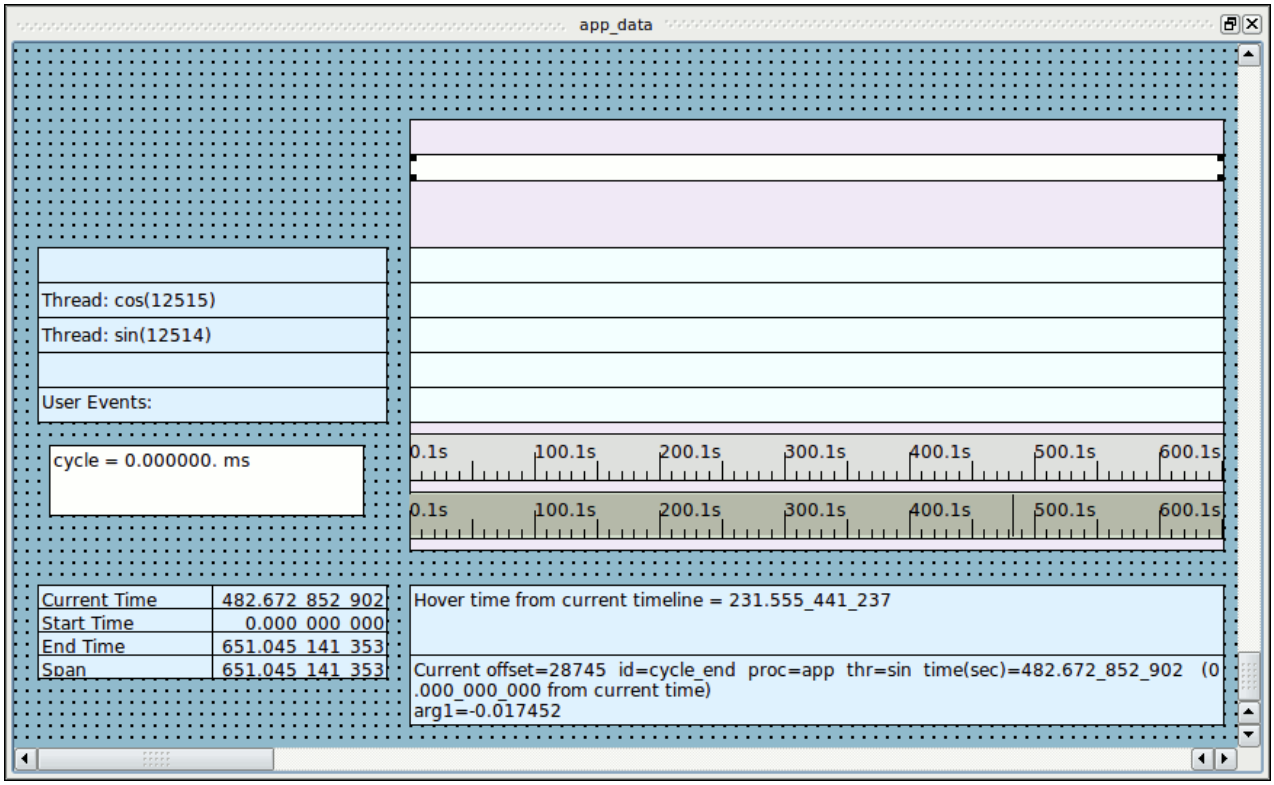


図 4-21. データ・グラフの追加

- 挿入したばかりのデータ・グラフの内側をクリックして下さい。
- グラフ・コンテナー上部の空いている空間が埋まるように上部境界線をデータ・グラフ上部へ、下部境界線をデータ・グラフ下部へドラッグして下さい。
- 空いている空間を埋めるには新しく挿入されたデータ・グラフの上下の線をクリックしてドラッグして下さい。
- データ・グラフの中ほどをダブルクリックして下さい。

Edit Data Graph Profile ダイアログが現れます。

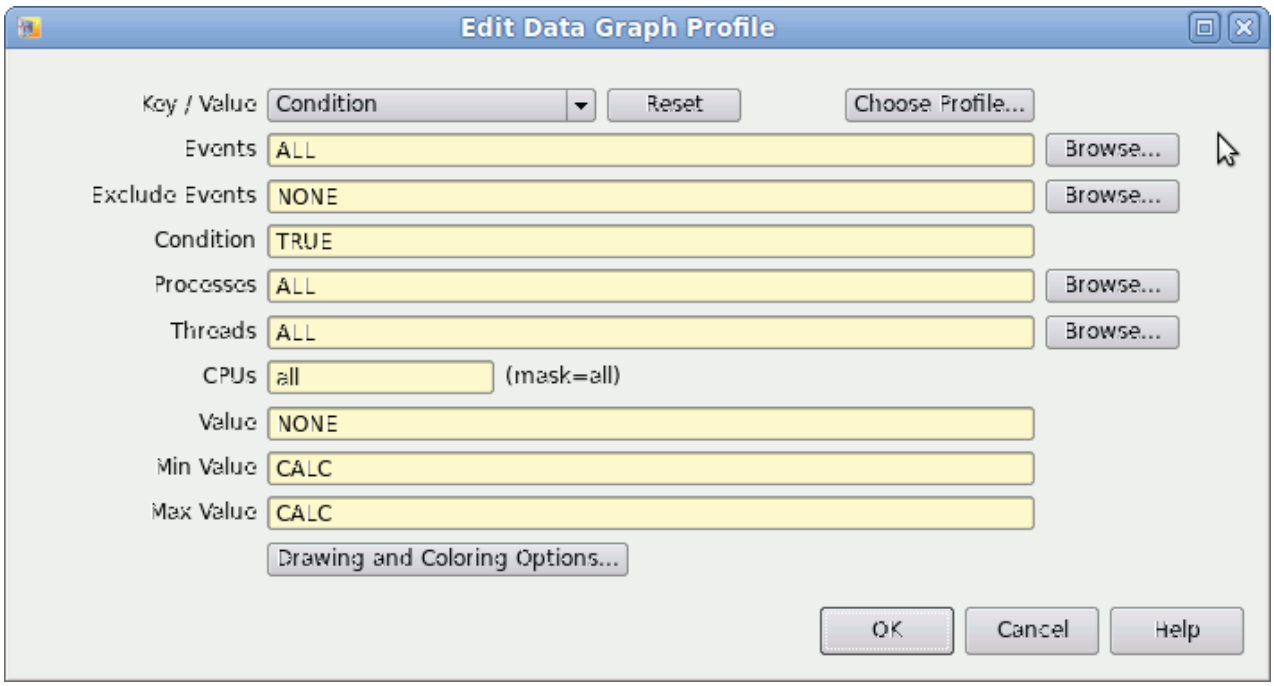


図 4-22. Edit Data Graph Profile ダイアログ

- 以下を

  ```
  cycle_end
  ```

  Events テキスト・フィールドに入力して下さい。

- 以下を

  ```
  arg1_dbl
  ```

  Value テキスト・フィールドに入力して下さい。

- Edit Data Graph Profile ダイアログを閉じるには OK を押して下さい。
- データ・グラフの内側を右クリックしてポップアップ・メニューから Adjust Colors in Selected を選択、サブメニューから Data Graph Value Color...を選択して下さい。
- データ・グラフ用に Select color ダイアログからお好みの色を選択して下さい。Select color ダイアログを閉じるには OK をクリックして下さい。
- *表示モード*へ戻るには、app_data のラベルが付いた表示パネルのどこでも右クリックしてポップアップ・メニューから Edit Mode を選択または Ctrl-E を押下して下さい。

- プログラムが作った正弦波を見るには表示をズームして下さい。

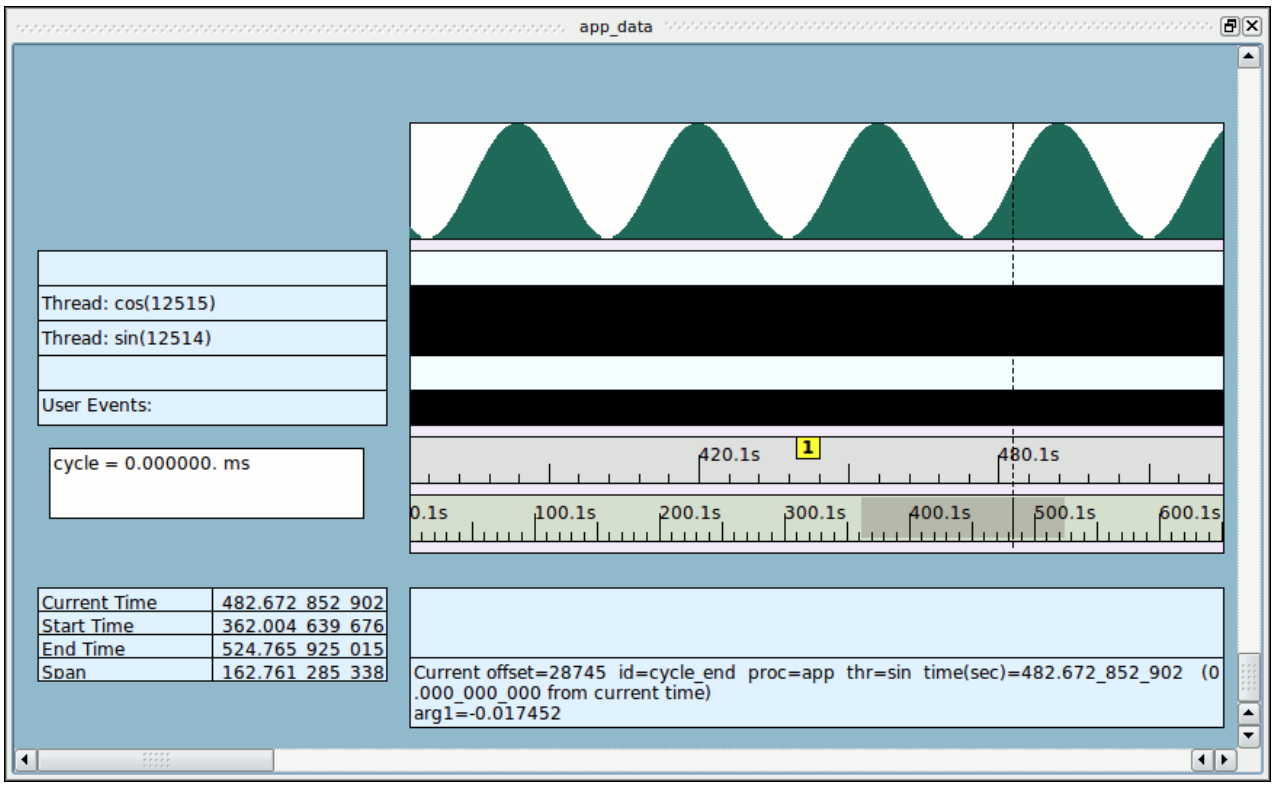


図 4-23. データ・グラフを含むタイムライン

カーネル・トレース

カーネル・トレースはシステムの動作内容およびアプリケーションが相互におよびカーネルとどのように関わっているかを驚くほど知る手掛かりとなります。

カーネル・トレースを利用するにはトレース対応のカーネルを実行している必要があります。

-trace および**-debug** で終わるカーネル名称はカーネル・トレースが有効となります。以下のコマンドを使用することでどのカーネルが実行中かを確認することが可能です。

```
uname –r
```

トレースが有効であるカーネルを実行していない場合、直ぐに再起動してブート時の GRUB メニューからそれを選択して下さい。現時点でシステムを再起動することが出来ない場合、チュートリアルに従い事前に記録されたカーネル・データを指示通りにロードして下さい。

- NightTrace メイン・ウィンドウの 1 番目のタブをクリックして下さい。
- Daemons パネルの(デーモンがまだ実行中の場合は有効となっている) Halt ボタンをクリックしていない場合はそれを押して、ユーザー・デーモンが停止していることを確認して下さい。
- Trace Segments パネルの app_data セグメントを選択して下さい。
- Trace Segments パネルの Close Trace Data ボタンを押して下さい。

NightTrace はトレース・データが保存されておらず破棄されることを警告するダイアログをポップアップ表示しますが、本チュートリアルではこのデータを保存する必要はありません。

カーネル・トレース・データの取得

もしトレースが有効であるカーネルを実行していない場合、本項はスキップして 4-32 ページの「Using Prerecorded Kernel Data」項を参照して下さい。

- NightTrace メイン・ウィンドウの最初のページにある Daemons パネルの kernel_trace_to_gui エントリーをダブルクリックして下さい。

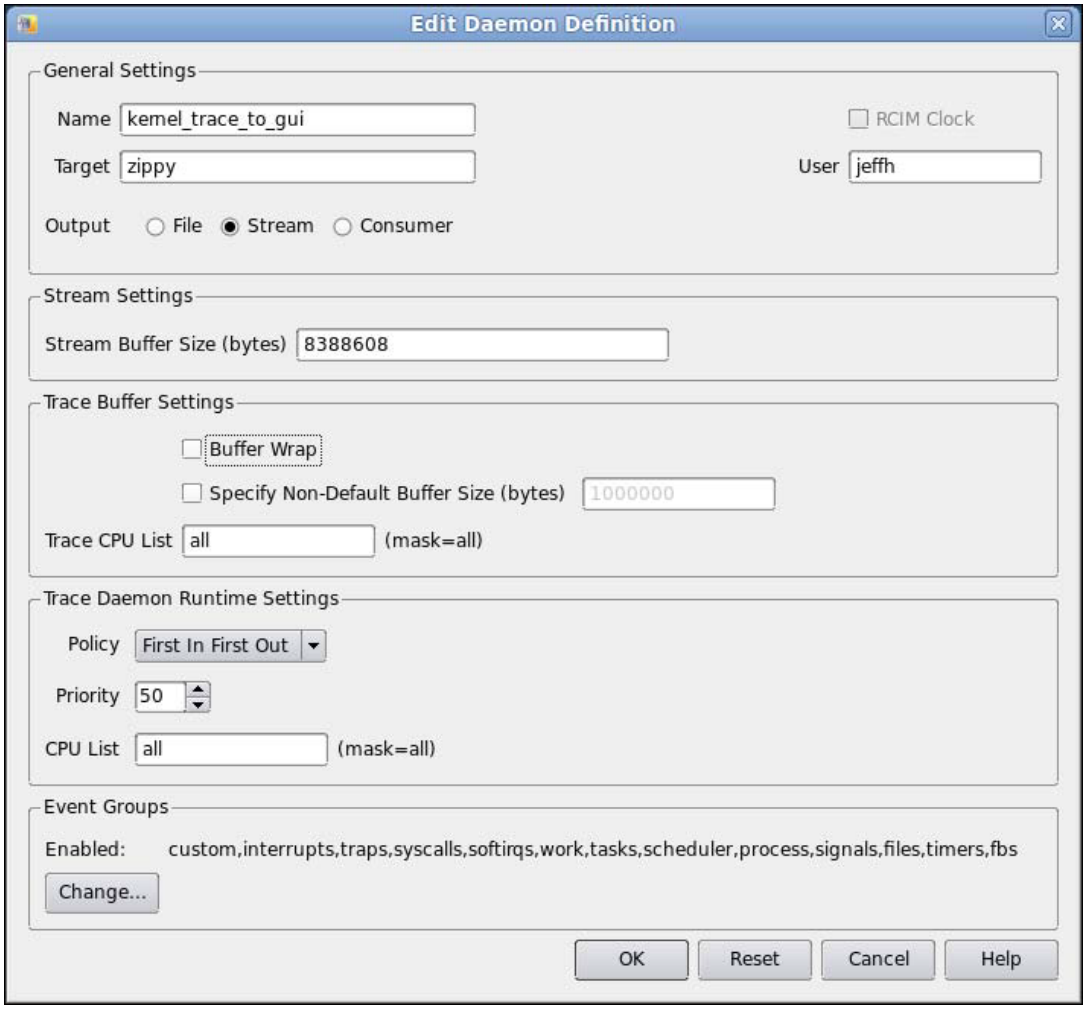


図 4-24. Edit Daemon Definition ダイアログ

NOTE

Trace Buffer Settings と Event Groups は実行している RedHawk カーネルのバージョンに応じて異なる可能性がありますが、全てのバージョンで Buffer Wrap 設定があります。

現時点では何も変更する必要はありません。この手順は、制御可能なカーネル・トレースの属性を示すことを目的とする参考情報です。

- OK を押して下さい。

システムの動作状況にもよりますが、巨大なカーネル・トレース・データが比較的短時間に生成される可能性があります。NightTrace の操作は多くのユーザーにとって新たな経験となる可能性がありますので、新しいユーザーのためにデータ・フローを管理可能なサイズに制限します。

- kernel_trace_to_gui が Daemon Control Area で選択されていることを確認して下さい。
- Launch ボタンを押して下さい。
- Resume ボタンを押して下さい。
- Daemon Control Area のデーモンの総計を観察して下さい。少なくとも 200,000 イベントが Logged 列に示されたら、Halt ボタンを押して下さい。

次項はスキップして 4-33 ページの「カーネル・データの解析」へ直接ジャンプして下さい。

事前に記録したカーネル・データの利用

本項はトレースが有効であるカーネルを起動しないでこのチュートリアルを使用している人に対してのみ提供します。

前項で実際のカーネル・トレース・データを収集した場合、4-33 ページの「カーネル・データの解析」へスキップして下さい。

NightStar の **tutorial** ディレクトリには、4-33 ページの「カーネル・データの解析」項で使用することが可能な予め記録されたカーネル・データが含まれています。

- NightTrace メイン・ウィンドウの File メニューから Open Files...メニュー項目を選択して下さい。
- ファイル・ダイアログの File name テキスト領域に以下を入力して下さい：

```
/usr/lib/NightStar/tutorial/.kernel-data
```

- Open Trace FILE(s)ボタンを押して下さい。

次項へ進んで下さい。

カーネル・データの解析

NightTrace はカーネル・データがキャプチャされたシステムに合わせて既定のカーネル表示ページを自動的に生成します。

- 新たに生成されたカーネル表示ページ表示するには NightTrace メイン・ウィンドウに生成されたタブをクリックして下さい。タブは<machine_name> Timeline のような名称が付けられます。

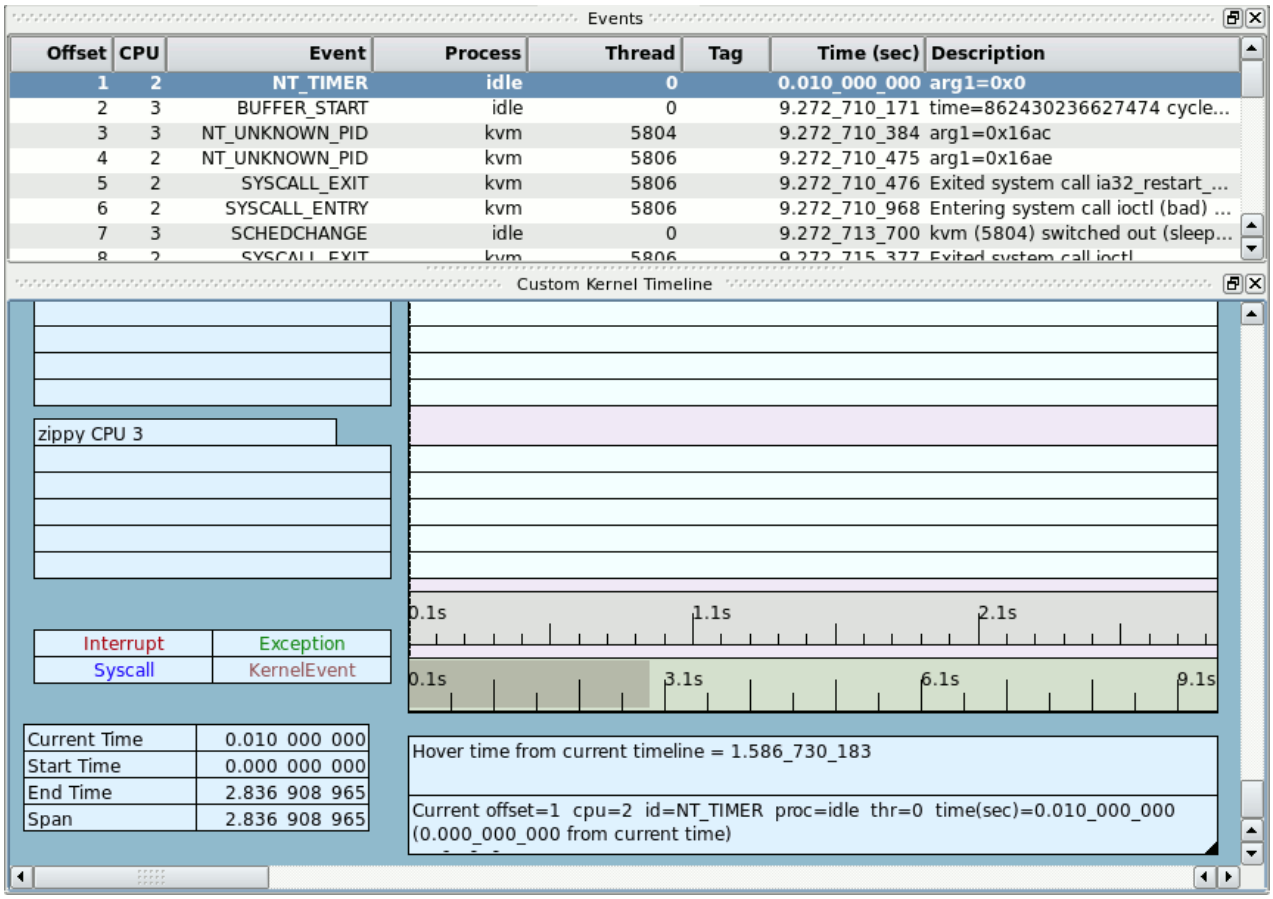


図 4-25. カーネル表示ページ

NOTE

CPU の数が異なる場合はタイムラインは大幅に異なって見える可能性があります。特別なシステム動作もまた表示が変わる可能性があります。この段階ではそのような違いについて気にしないで下さい。

- データ・セットの終端へ移動するには Alt+Right を押下して下さい。
- アクティブ領域をクリックして詳細が見れるようになるまでズーム・インして下さい。

各 CPU については、次の情報が表示されます：

- 割り込みアクティビティ(赤)
- マシーン例外アクティビティ(緑)
- システムコール・アクティビティ(青)
- プロセス毎 CPU 使用状況(様々な色で表示)
- 詳細なカーネル・イベント(暗赤)

表示ページの左側にあるデータ・ボックスはそれらが説明する情報に一致するよう色分けされています。それらの中身は現在のタイムラインの位置に基づき動的に変化します。

- Ctrl+F を押下して Profiles ダイアログを表示して下さい。
- Key/Value 選択領域の右にある Reset ボタンをクリックして下さい。
- Processes テキスト・フィールドの右にある Browse...ボタンを押して下さい。

  Select Processes ダイアログが現れます。

- 既知プロセスのリストから **app** プロセスを選んで下さい。
- Select ボタンを押して Select Processes ダイアログを閉じて下さい。
- Key / Value オプション・リストから System Call Enter Events オプションを選んで下さい。

  Select System Calls ダイアログが現れます。

- 表示されたシステムコールのリストから nanosleep を選んで下さい。
- Select ボタンを押して Select System Calls ダイアログを閉じて下さい。
- SYSCALL_RESUME のみが含まれるように Events テキスト・フィールドのイベントのリストを変更して下さい。
- Search Forward ボタンを押して下さい。

入力した情報に基づく新しいプロファイルが Profile Status List に追加され、現在のタイムラインがプロセス **app** の停止した nanosleep システムコールの再開の次の発生場所へ変更されます。

NOTE

NightTrace が入力したソート基準に一致する事象を見つけられない場合、検索基準を見直して下さい。上述の手順で Reset ボタンを押すのを飛ばした可能性があります。Threads テキスト・フィールドは ALL を示しており sin ではないことを確認して下さい。

- カーネル表示ページに対応するタブをクリックして下さい。
- フォーカスを戻すには背景内のページの(現在時刻のインジケーターが変わるのでグリッド内ではない)どこでもクリックして下さい。

- 以下に示すような詳細な情報が見れるまでズーム・インして下さい：

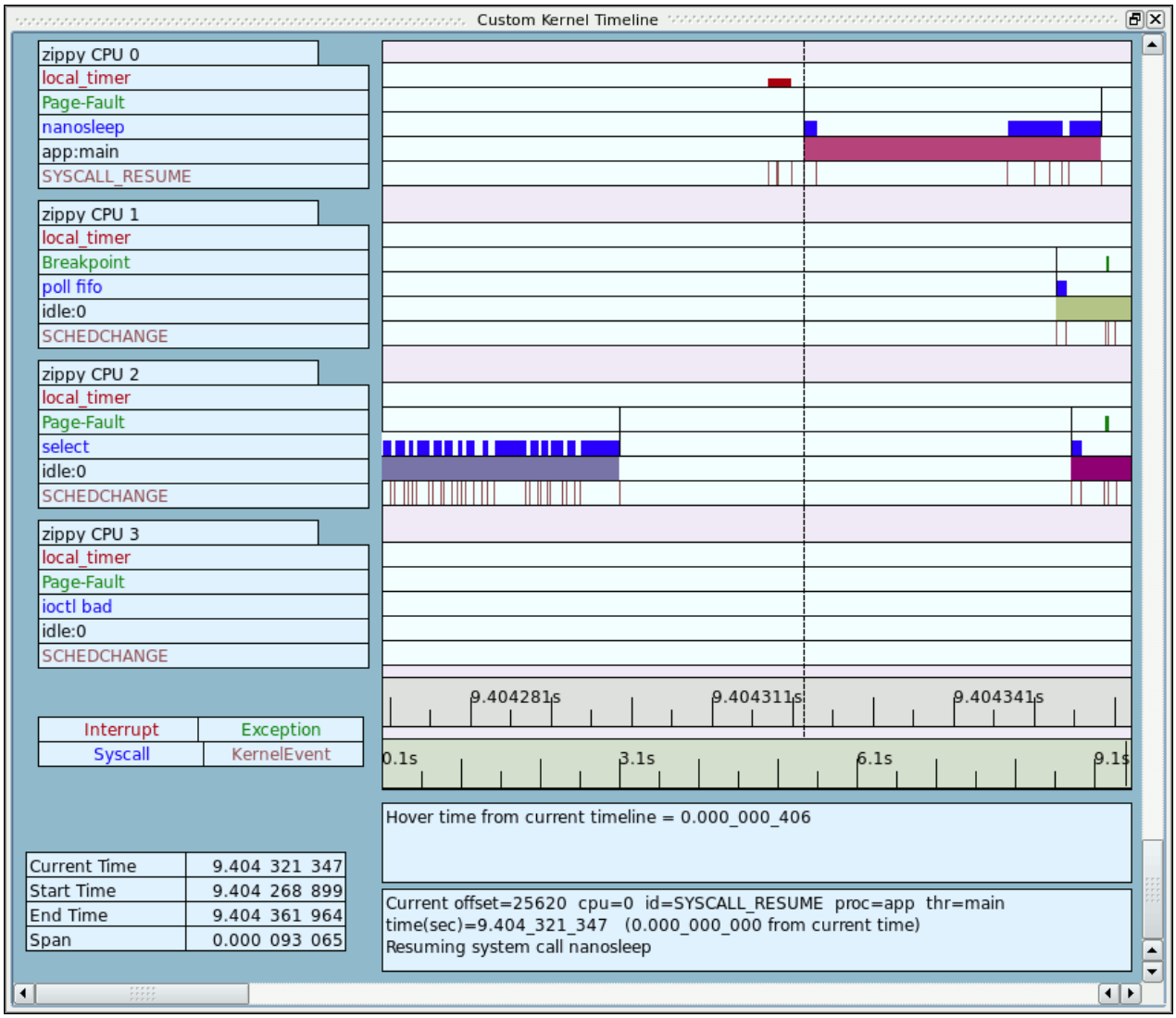


図 4-26. nanosleep におけるシステムコールの再開

NOTE

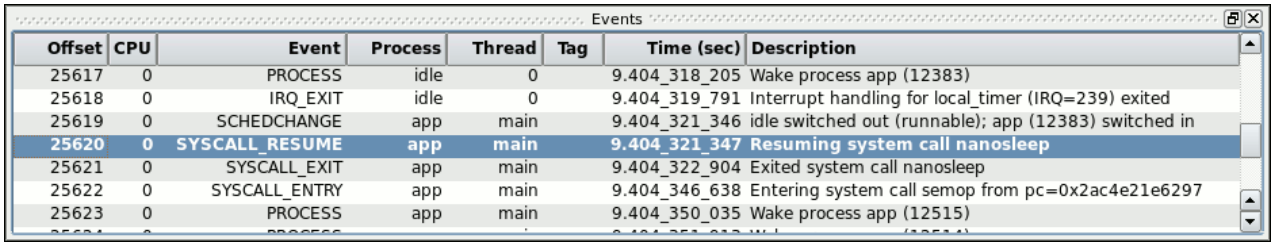
CPU の数が異なる場合は表示されるタイムラインは大きく異なる可能性があります。更なるシステム・アクティビティも同様に表示を変えさせる可能性があります。**app** プロセスを示す行に似ているものが見える事象を探し出すには検索を数回繰り返して下さい。ツールバーの順方向検索アイコンを押すまたは Ctrl-G を押下することで最新の検索を繰り返すことが可能です。

現在のタイムラインの左側にある赤い棒は割り込みが発生したことを示しています。この場合、これは `local_timer` 割り込みです。

システムコールと例外の行を繋ぐ長い縦の黒い線はコンテキスト・スイッチを表しています。現在のタイムライン(長方形の表示領域全体に及ぶ破線)はこのズーム設定でコンテキスト・スイッチの線と重なっている可能性があります。

- Events パネル内の強調されたイベントを選択して下さい。これは現在のタイムラインのイベントで、SYSCALL_RESUME のはずです。

現在強調されているイベントに関する Events パネルの Description 列はイベントについてさらに詳細を説明しています：



Offset	CPU	Event	Process	Thread	Tag	Time (sec)	Description
25617	0	PROCESS	idle	0		9.404_318_205	Wake process app (12383)
25618	0	IRQ_EXIT	idle	0		9.404_319_791	Interrupt handling for local_timer (IRQ=239) exited
25619	0	SCHEDCHANGE	app	main		9.404_321_346	idle switched out (runnable); app (12383) switched in
25620	0	SYSCALL_RESUME	app	main		9.404_321_347	Resuming system call nanosleep
25621	0	SYSCALL_EXIT	app	main		9.404_322_904	Exited system call nanosleep
25622	0	SYSCALL_ENTRY	app	main		9.404_346_638	Entering system call semop from pc=0x2ac4e21e6297
25623	0	PROCESS	app	main		9.404_350_035	Wake process app (12515)

図 4-27. 検索後の Events パネル

- 現在のタイムラインが SYSCALL_RESUME イベントである間に Up キーを押下して下さい。

現在のタイムラインは前方のイベントに変更され、テキストの記述は次のような文章を含むコンテキスト・スイッチを示します：

```
idle switched out (runnable); app (12383) switched in
```

青い棒はシステムコールの動作状況を表します。左側のデータ・ボックスは現在のタイムライン上または左側のシステムコールに関するシステムコール名称を表示します。

- SYSCALL_RESUME イベントへ戻るには Ctrl-G キーを押下して下さい。

上のスクリーン・ショットで表示された事例では、nanosleep から戻った sine スレッドの直ぐ後に、メイン・スレッドが **app.c** の 97 行目の nanosleep 呼び出しを抜けます。次に 99 行目の semop ライブラリ呼び出しを実行するため semop システムコールに入ります。

NOTE

一部のシステムでは、システムコールが semop の代わりに ipc と説明される可能性があります。

カーネル・データとユーザー・データの混合

トレースが有効であるカーネルを実行していない場合、本項はスキップして 4-40 ページの「NightTrace Analysis API の利用」へ進んで下さい。

- NightTrace メイン・ウィンドウの最初のタブをクリックして下さい。

- (前述の手順で停止されているはずですが)Halt ボタンが有効である場合、それを押してカーネル・デーモンが停止されていることを確認して下さい。
- Trace Segments パネルの kernel_trace_to_gui セグメントを選択し、コンテキスト・メニューの Close Trace Data メニュー・オプションを選択して下さい。
- Click と Shift+Click のマウスとキーボードの操作で Daemon Control Area の両方のデーモンを選択して下さい。
- 継続する前に両方のデーモンが完全に選択されていることを確認しましょう。そうではない場合、正しい状態にして下さい。
- Launch ボタンを押して下さい。

継続する前に次の 4 つの手順を読んで、その後、順に実行して下さい。

- Resume ボタンを押して下さい。
- app_data 行の Buffer セルに 5000 以上のイベントが表示されるまで待って下さい。
- Flush ボタンを押して下さい。
- Halt ボタンを押して下さい。

ユーザー・アプリケーションとカーネルの両方からのデータがキャプチャされ NightTrace に取り込まれました。

- Summary メニューから Change Summary Profile を選択して下さい。
- ページ上部にある Profile Status List から sine プロファイルを選択して下さい。
- ツールバー上の Summarize アイコンを押して下さい。

最後の操作で、4-22 ページの「サマリー情報の生成」で定義した sine ステートのサマリーを含む新しいページを生成させました。

- 降順でソートされるまで Duration ヘッダーをクリックして下さい(1 回以上クリックする必要があるかもしれません)。
- 先頭行の Duration の値を含んでいるセルをクリックして下さい。
- カーネル表示ページに対応するタブをクリックして下さい。
- タイムラインにフォーカスを戻すにはページ内のどこでも、但し背景のみをクリックして下さい。(グリッドの内側をクリックすると現在のタイムラインが変わります)
- sine スレッドのサイクルに関する詳細がはっきりと見えるまで必要に応じてズーム・インまたはズーム・アウトして下さい。

下に示す図では、sine スレッドは rcim 割り込みのカーネル処理によってプリエンプトされています。

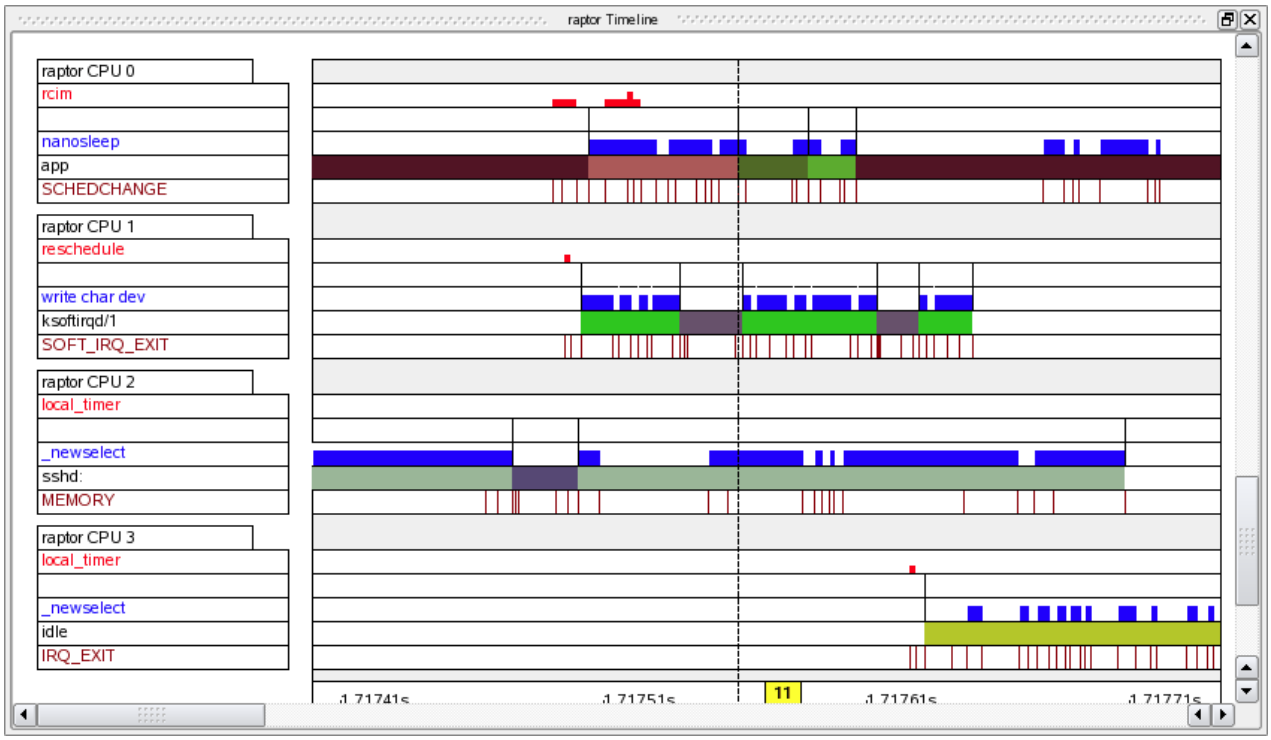


図 4-28. 長時間ステートの事例

トレース・データ内のサイクルが間延びした理由は他の事情が原因である可能性があります。

- sine_thread() は他のプロセスにプリエンプトされた？
- 割り込みはサイクルの中で発生した？
- sine スレッドが実行中の CPU からサイクルが奪われたハイパースレッド化したシブリング CPU 上で著しい活動があった？
- アプリケーションはページ・フォルトもしくは他のマシン例外を受けた？
- ハイパースレッド化したシブリング CPU 上の活動が app が実行中の CPU に干渉した？

これらの状況の一部は 7-9 ページの「オーバーラン検知とシステム・チューニング」でさらに詳細に説明しています。

マシン例外は、例外の種類、(適用可能な場合)障害のあるアドレス、例外が発生した PC を詳述する情報を含みます。

- カーネル表示ページが選択されている間に Ctrl+F を押下して下さい。
- Key / Value オプション・リストから Exception Enter Events を選択して下さい。
- 例外のリストから Page-Fault を選択して下さい。
- Select ボタンを押して下さい。

- Search/Forward ボタンを押して下さい。

ページ・フォルトが見つかった場合、現在のタイムラインは次のページ・フォルトの発生場所へ移動します。カーネル表示ページ上部のテキスト領域は、障害が発生した PC や障害のアドレスを含む例外に関する詳細な情報を含んでいます。

NightTrace Analysis API の利用

NightTrace はユーザー・アプリケーションが事前に記録したトレース・データを解析または生トレース・データを監視および解析することが可能な強力な API を提供します。

ユーザーは状態や条件を定義しイベントが発生した時にそれらを処理するプログラムを書くことが可能です。

本チュートリアルでは、NightTrace に API プログラムを機械的に構築するよう指示します。

- 2 つの Profiles ツールバー・アイコン  のどちらかをクリックして下さい。

- Profile Status List から sine プロファイルを選択して下さい。
- Profiles メニューから Export to API Source...メニュー項目を選択して下さい。

下のダイアログが表示されます：

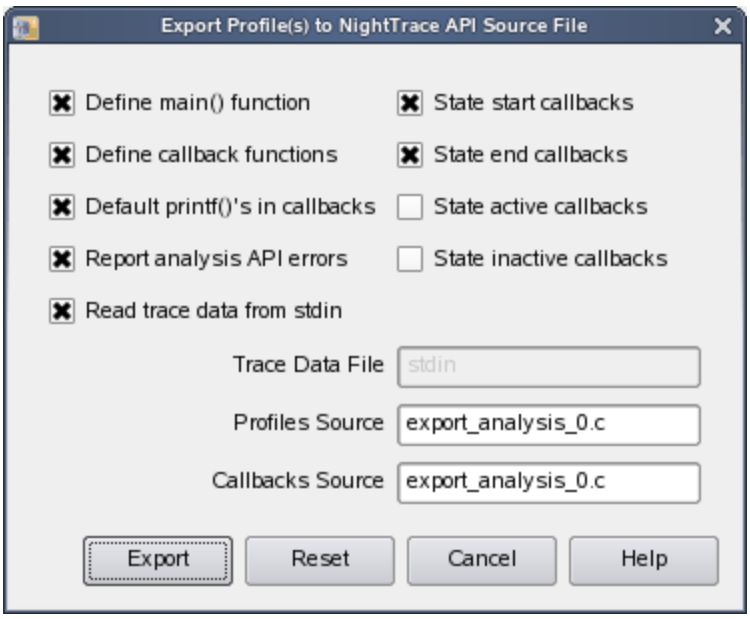


図 4-29. Export Profiles to NightTrace API Source File ダイアログ

- State start callbacks チェックボックスを外して下さい。
- Export ボタンを押して下さい。
- NightTrace メニューから Exit Immediately メニュー項目を選択して NightTrace を終了して下さい。

NightTrace は **sine** プロファイルで定義された状態の発生を待って各インスタンスに関する一部の情報を出力する API プログラムを生成しました。

- 次のコマンドを使って API プログラムをビルドして下さい：

```
cc -g export_analysis_0.c -lntrace_analysis
```

本プログラムは生トレース・データを取り込むことを想定しています。

NightTrace GUI を使ってユーザー・デーモンを構成し、NightTrace に解析プログラムを自動的に起動させることが可能です。

あるいは、同じ効果を得るためにコマンド・ライン・ユーザー・プログラム **ntraceud** を使用することも可能です。

- 次のコマンドを入力して下さい：

```
ntraceud --stream --join /tmp/data | ./a.out
```

このコマンドはハンドルとして**/tmp/data** ファイルを使用している実行中のアプリケーションからのトレース・データのキャプチャーを開始するよう **ntraceud** に指示します。**--stream** オプションは指定したファイルにデータを記録する代わりに **stdout** に送信されることを意味します。

データ・レートが極めて低くバッファリングが関わるため、アプリケーション・プログラムは直ぐには出力の生成が始まらない可能性があります。

- アプリケーションが即座に消費するために現在のバッファをフラッシュするには、次のコマンドを別のターミナル・セッションで発行して下さい：

```
ntraceud --flush /tmp/data
```

NOTE

データがシステム・バッファを通過することを許可するために数秒の間に
そのコマンドを数回繰り返す必要があるかもしれません

以下と同じようなデータが解析プログラムが起動されたターミナル・セッションの **stdout** に現れます：

```
sine (end)offset 665 occur 333 code 2 pid 3399 time 16.628649 duration 0.000003
sine (end)offset 667 occur 334 code 2 pid 3399 time 16.678631 duration 0.000003
sine (end)offset 669 occur 335 code 2 pid 3399 time 16.728655 duration 0.000003
sine (end)offset 671 occur 336 code 2 pid 3399 time 16.778676 duration 0.000003
sine (end)offset 673 occur 337 code 2 pid 3399 time 16.828693 duration 0.000003
sine (end)offset 675 occur 338 code 2 pid 3399 time 16.878716 duration 0.000004
sine (end)offset 677 occur 339 code 2 pid 3399 time 16.928745 duration 0.000003
sine (end)offset 679 occur 340 code 2 pid 3399 time 16.978760 duration 0.000003
sine (end)offset 681 occur 341 code 2 pid 3399 time 17.028779 duration 0.000003
```

- 次のコマンドを発行してデーモンを終了して下さい：

```
ntraceud --quit-now /tmp/data
```

トレースが有効なカーネル・デーモンが実行中ではない場合、本項の残りはスキップして 4-64 ページの「終了- NightTrace」へ進んで下さい。

いくつかの API プログラムのサンプルは NightTrace と一緒に提供されます。

- 次のコマンドを入力して watchdog サンプル・プログラムをビルドして下さい：

```
cc -g -o watchdog ¥
/usr/lib/NightTrace/examples/c/analysis_api/watchdog.c ¥
-lntrace_analysis
```

この簡素なサンプル・プログラムは特定の CPU 上のコンテキスト・スイッチを待って、切り替わったプロセスの名称を出力します。

今回、**ntracekd** カーネル・デーモンは 5 秒のカーネル・データを取得するために使用され、その出力を **watchdog** プログラムに流します。

- 次のコマンドを発行して下さい：

```
ntracekd --stream --wait=5 /tmp/x | ./watchdog 1
```

プログラムは最終的に以下と同じような出力を生成します：

```
context switch: 4.979350027     4 ksoftirqd/0
context switch: 4.979358275  2846 X
context switch: 4.983906074     0 idle
context switch: 4.983960385  2846 X
context switch: 4.994892976  3167 firefox-bin
context switch: 4.994989171  4492 ntfilterl
context switch: 4.995070736  4489 watchdog
context switch: 4.995092415  4492 ntfilterl
context switch: 4.995173214  4489 watchdog
context switch: 4.995188096  4492 ntfilterl
context switch: 4.995256175  4489 watchdog
context switch: 4.995270824  4492 ntfilterl
context switch: 4.995332743  4489 watchdog
context switch: 4.995355783  2846 X
context switch: 5.000351519     4 ksoftirqd/0
context switch: 5.000360675  2846 X
```

アプリケーションの自動トレース

本項では NightTrace 解析ツールの新たな呼び出しを利用することになります。

- NightTrace セッションがアクティブのままである場合、File メニューから Exit NightTrace Immediately を選択して NightTrace を終了して下さい。

NightTrace は Application Illumination と呼ぶコンポーネントを提供し、それは自動的にユーザー・アプリケーションにサブプログラムの入り口と出口を記録するトレース・ポイントを備えます。

特にそれらサブプログラムの呼び出しに対する引数と戻り値はトレース・データの一部として含めることが可能であるため、データを解析する時にそれらを見ることが可能です。

全てのサブプログラムに自動的にトレース機能を備えるという事ではありません。Application Illumination はグローバルに利用できる外部シンボル名称を持たない関数(例えば、C 言語での `static void func();`)を検出することが出来ません。同様にリンクされた共用ライブラリにとって完全に内部にある関数(つまり外部エントリ・ポイントを持たない関数)も検出することは出来ません。更にデフォルトで Application Illumination はコンパイラーが生成したデバッグ情報を持つ関数についてのみ動作しますが、この動作は変更が可能です。

ユーティリティ **/usr/bin/nlight** はユーザー・アプリケーションにトレース機能を備えるために使用される主要なインターフェースです。

nlight はサブプログラムの選択と除外、同様に詳細レベルのカスタマイズを提供します。

本チュートリアルでは、これまで使用してきた **app** プログラムに早く簡単にトレース機能を備えるために **nlight** のウィザードを使用します。

nlight ウィザード - プログラムの選択

- 本チュートリアルの初期段階で生成した tutorial テスト・ディレクトリの中にいる間に **nlight** ツールを起動します：

```
nlight &
```

以下のウィンドウが表示されます。

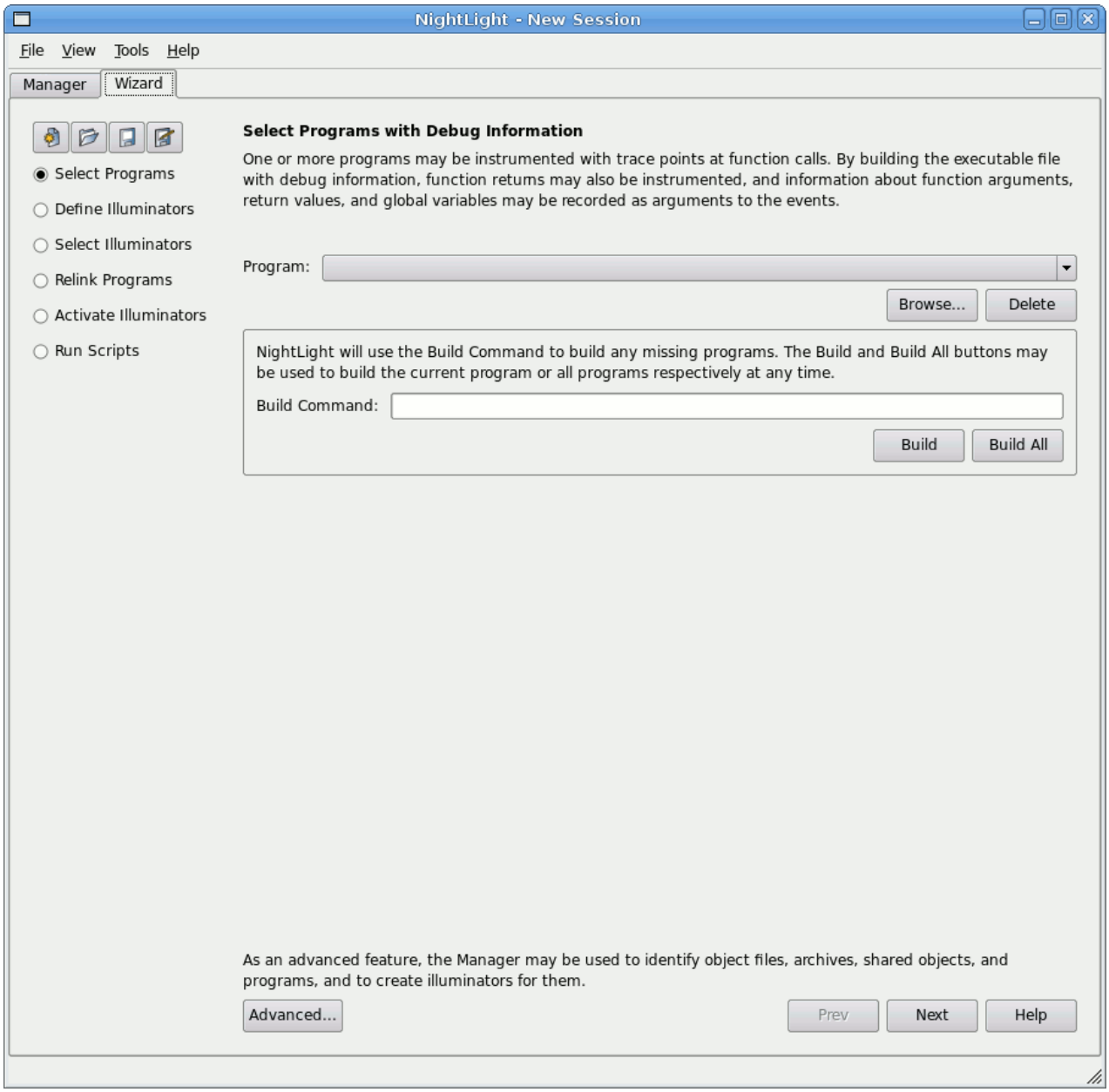


図 4-30. nlight Wizard - Select Programs ステップ

NOTE

ウィンドウ表示が上図と著しく異なる場合、$HOME ディレクトリ(または root として実行中の場合は root の$HOME ディレクトリ)から.nlightrc ファイルを削除して下さい。nlight を終了後に上述どおりに再度起動して下さい。

デフォルトで Wizard タブが呼び出されユーザー・アプリケーションにトレース機能を備えるための段階的な指示を提供します。

ページ左側の箇条書きリストはウィザード内のどのステップを現在取り組んでいるのかを示しており、一方下部の Prev および Next ボタンはステップを移動します。

最初のステップは Select Program で、どのプログラムを装飾するのかを **nlight** に指示することです。

- Browse...ボタンを押してファイル選択ダイアログから **app** プログラム・ファイルを選択、続いて Save を押してファイル選択ダイアログを閉じて下さい。

現在プログラム選択の下の Build Command テキスト領域はデフォルトの **make** コマンドを含んでいることに注意して下さい。厳密にいえば要求されませんが、**nlight** 内部からリビルドすることを選択するといけないので、元のプログラムをリビルドできるコマンドを **nlight** に提供すると便利です。更に指定されたプログラム・ファイルが存在しないことに気付いた場合、**nlight** は自動的にそのコマンドを起動します。

- 次のステップへ進めるには Next ボタンを押して下さい。

nlight ウィザード - Illuminator の定義

Define Illuminators のステップが表示され、これは装飾したいアプリケーション内のコードの一部を選択することが可能です。

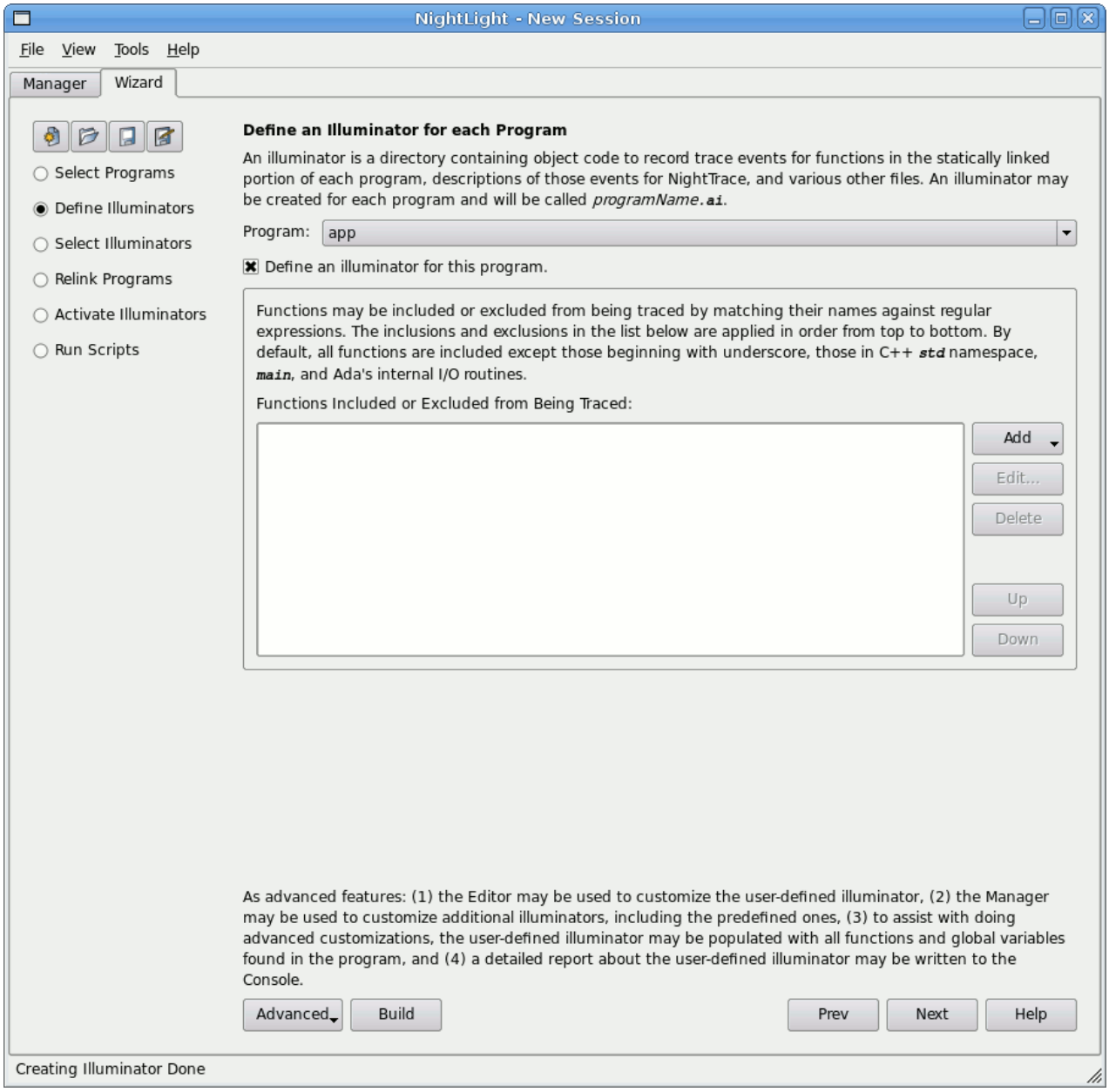


図 4-31. nlight Wizard - Define Illuminators ステップ

用語 *illuminator* はトレース機能を備えるコードで必要となる **nlight** が生成したファイルを含むディレクトリを指します。通常、そのディレクトリの中身は直接関わることはなく、**nlight** が全ての作業を行います。Define an illuminator for this program チェックボックスは **nlight** に **app** プログラムの静的にリンクした部分にトレース機能を備える事を指示します。

本ページはトレース機能の装備を構成したいもしくは除外したい特定のサブプログラムを指定することが可能な選択および除外領域も含みます。また、複数の関数を簡単に構成または除外するために正規表現経由でパターンを指定することも可能です。

このステップでは **app** プログラムの静的にリンクされた部分全てを **nlight** に装飾させます。

- Define an illuminator for this program のラベルが付いたチェックボックスがチェックされていることを確認して下さい。
- 次のステップへ進めるには Next ボタンを押して下さい。

nlight ウィザード - Illuminator の選択

Select Illuminators のステップが現在表示されています。

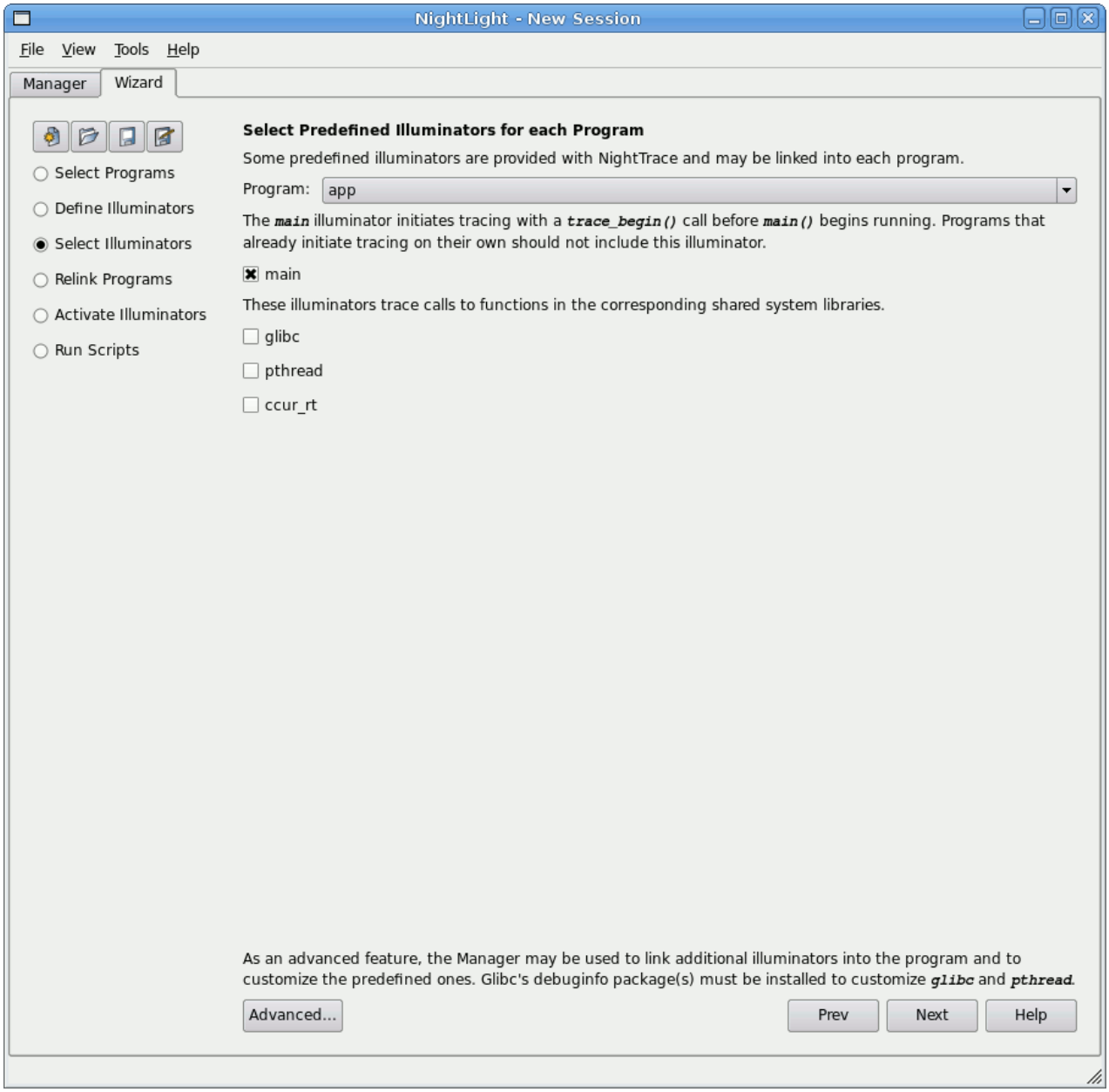


図 4-32. nlight Wizard - Select Illuminators ステップ

本ステップは更にプログラムに対し事前に定義された illuminator を選択することが可能です。

NOTE

事前定義された illuminator のリストはお手元のシステムとは異なる可能性があります。しかしながら、全てのシステムで `main`, `glibc`, `pthread` があるはずです。

main illuminator は特別でユーザー・アプリケーションが NightTrace API をまだ使用していない場合にのみ必要となります。**app** プログラムは既にそうしているので、このチェックボックスは外す必要があります。

- main チェックボックスを外して下さい。

その他の illuminator は既にビルドされて NightTrace に標準装備されています。このページの中間部分でプログラムで使用するシステム・ライブラリ用の illuminator を含めることが可能です。

- glibc チェックボックスをチェックして **glibc** illuminator を含めて下さい。
- pthread チェックボックスをチェックして **pthread** illuminator を含めて下さい。
- 次のステップへ進めるには Next ボタンを押して下さい。

nlight ウィザード - プログラムの再リンク

Relink Programs のステップが現在表示されています。

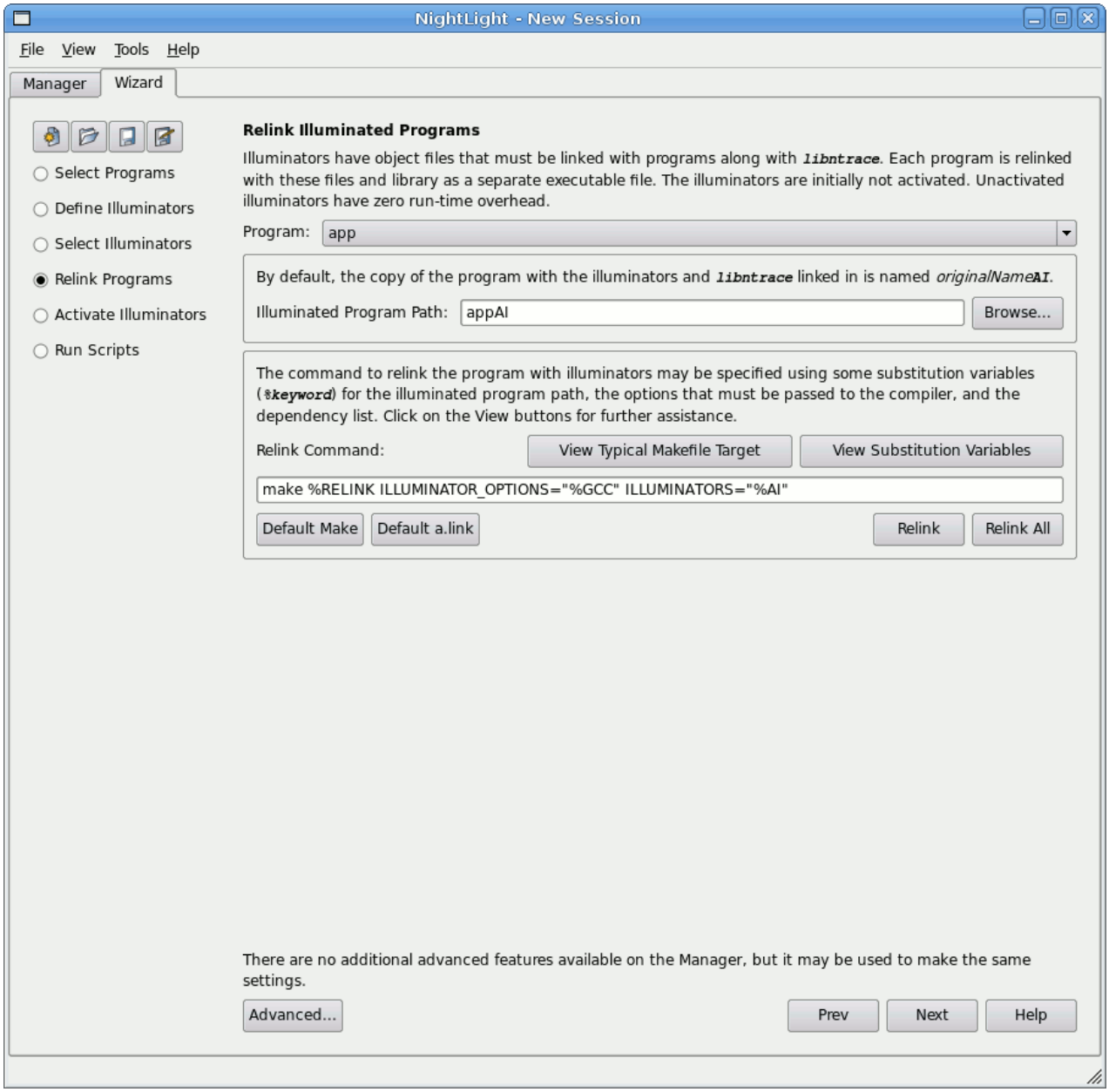


図 4-33. nlight Wizard - Relink Programs ステップ

illuminator を利用するには、元プログラムと全く同じオブジェクトとライブラリだけでなく **nlight** が生成したファイルもまたリンクする新しいバージョンの実行可能プログラムを生成する必要があります。

得られた実行ファイルは元プログラムから修正されていないオブジェクト・ファイルとライブラリが含まれますが、実行時に実際のトレース・イベント呼び出しを差し込む「ラッパー」関数もまた含まれます。

基本的に元プログラムを再生成していくつかの新しいリンク・オプションを追加する必要があるので、ウィザードはそれを行うコマンドをユーザーに入力してもらう必要があります。デフォルトの「relink」コマンドは既に入力されており、プログラムをビルドする **make** ユーティリティを使用することを前提としています。**Makefile** の命令を構成することをとても簡単にするいくつかの **make** パラメータを渡して新しいプログラムをビルドします。

殆どの場合、単に元のアプリケーションを生成するために必要な最終命令をコピーして改名し、リンク行にウィザードから渡されるオプションを追加することが可能です。

tutorial テスト・ディレクトリ内の **Makefile** は既に機能を組み込んだプログラム名に対して定義した命令を持っており、それは(慣例により)プログラムの元の名称に文字「AI」が加えられたものです。以下は **app** と **appAI** をビルドする命令を示す **Makefile** からの抜粋です。

```
app: app.c
    cc -g -o app app.c ¥
        -lntrace_thr -lpthread -lm –lrt

appAI: app.c
    cc -g -o appAI app.c ¥
        $(ILLUMINATOR_OPTIONS) -lntrace_thr -lpthread ¥
        -lm –lrt
```

appAI (トレース機能を備えたバージョンのプログラム)をビルドするルールは、ウィザードで渡される「relink」コマンドのオプションを含んでいることを除いては元の **app** プログラムをビルドするルールと全く同じであることに注目して下さい。

- Next ボタンを押して下さい。

これでプログラム appAI は自動的にリンクされます。

nlight ウィザード - Illuminator の有効化

Activate Illuminators のステップが表示されます。

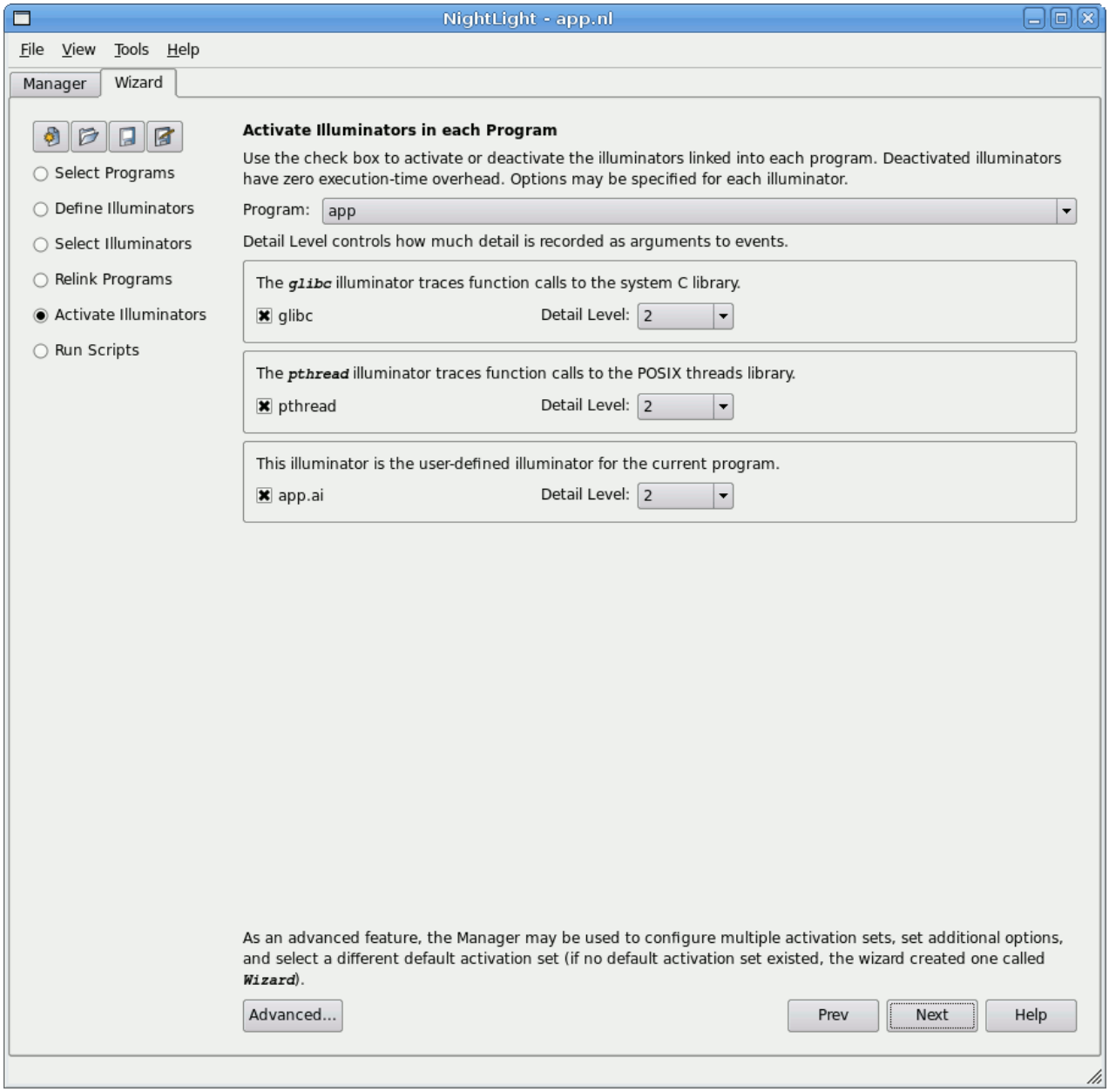


図 4-34. nlight Wizard - Activate Illuminators ステップ

Application Illumination の重要な特徴は、プログラムを再リンクして illuminator を含めると illuminator は不活性になる事です。illuminator が不活性であると同時にアプリケーションはオーバーヘッドなしでアプリケーションを実行することが可能です。

このステップでは、プログラムを実行した時にトレース・データが記録されるようにそれらを有効にします。

デフォルトの有効レベルは 2 で、これは各イベントと一緒に中程度の詳細を提供します。本チュートリアルでは更に詳細を見たいので、各 illuminator の詳細レベルを拡大します。

- glibc illuminator に関する Detail Level を 3 に変更して下さい。
- pthread illuminator に関する Detail Level を 3 に変更して下さい。
- app.ai illuminator に関する Detail Level を 3 に変更して下さい。
- 有効化を確定して次のステップに進むには Next ボタンを押して下さい。

プログラムの実行

Run Scripts のステップがウィザード内に現在表示されています。

ウィザードは便宜上このステップを提供します。

あえて **nlight** を閉じて、**nlight** の外でユーザー自身がアプリケーションを実行します。

- File メニューから Exit Immediately を選択して下さい。
- シェル・セッションで機能が組み込まれたプログラムを開始して下さい：**./appAI &**

IMPORTANT

app ではなくトレース機能を備えたプログラム **appAI** を起動したことを確認して下さい。

アプリケーションの Illumination イベントの解析

トレース機能を備えたプログラムが生成するデータを解析するには NightTrace を起動します。

- (3-42 ページの「終了 - NightView」の状態のままなので)同じようにトレース・データを生成する app プログラムの実行のインスタンスとの混合を避けるため、そのプログラムを終了します：

```
killall -9 app
```

- **appAI** プログラムを含んでいるディレクトリの中で次のコマンドを入力して下さい：**ntrace --import=appAI**

NightTrace 解析インターフェースが現れます。

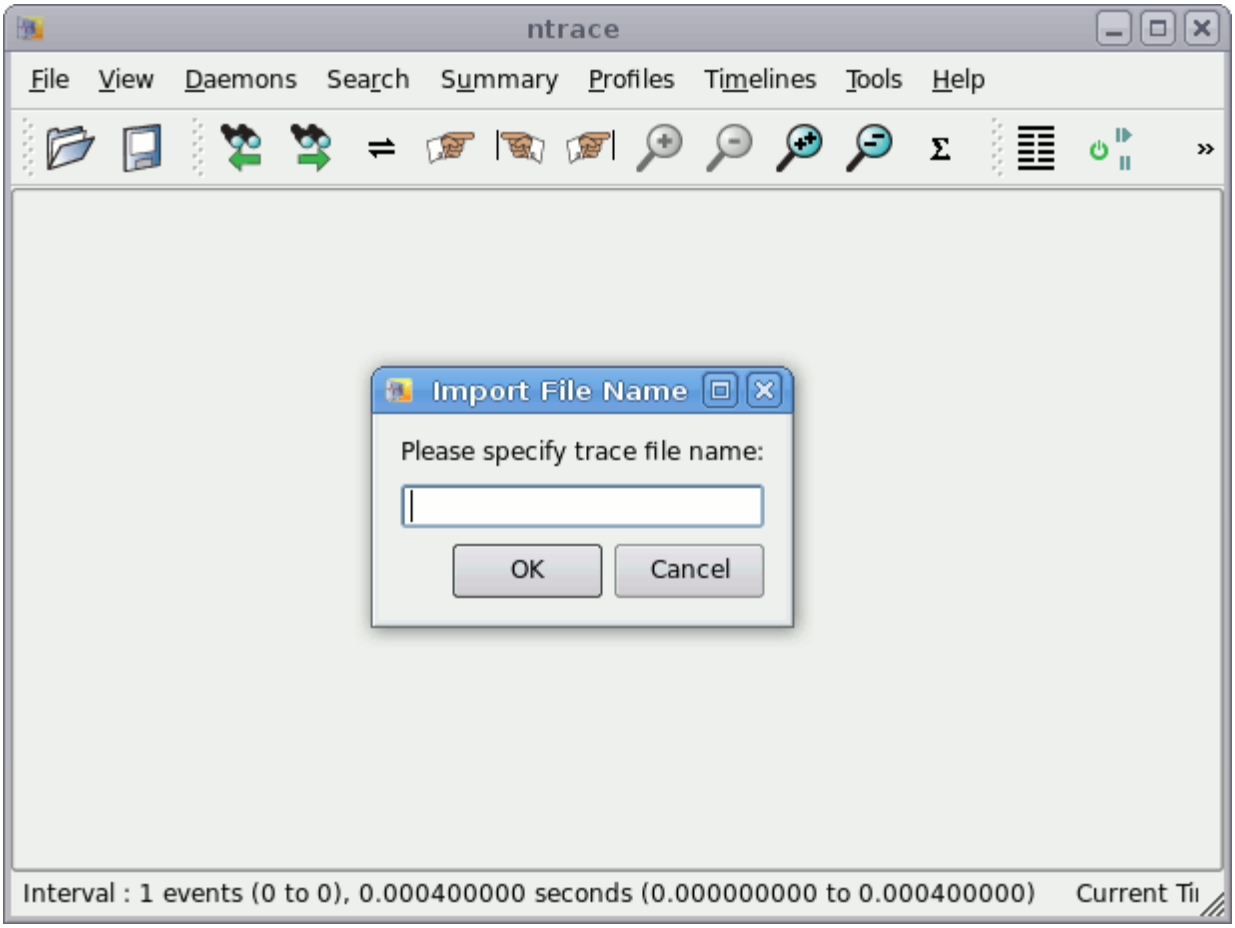


図 4-35. NightTrace - Import File Name

NightTrace は**--import** オプション付きで起動されたので、プログラムが trace_begin コールに渡す最初のパラメータであるトレース・データ・ファイルの名称を求めます。

- プロンプト・ダイアログに**/tmp/data** を入力して OK を押して下さい。

--import オプションの使用は、受け取ったトレース・イベントを完全に記述できるように **nlight** で生成された補助データをロードすることを NightTrace に指示します。その情報の場所はトレース機能を備えたアプリケーションの中に埋め込まれており、今回のケースでは **appAI** となります。

NOTE

main illuminator が **nlight** で選択された場合は、**ntrace** はトレース・ファイルの名称を既に分かっています。今回の例では、プログラムが **nlight** とは無関係にトレースが開始されるので main illuminator を含めませんでした。

現在 Daemons パネルはトレース機能を備えた **appAI** プログラムからトレース・ポイントをいつでも収集できるユーザー・デーモンを含んでいます。

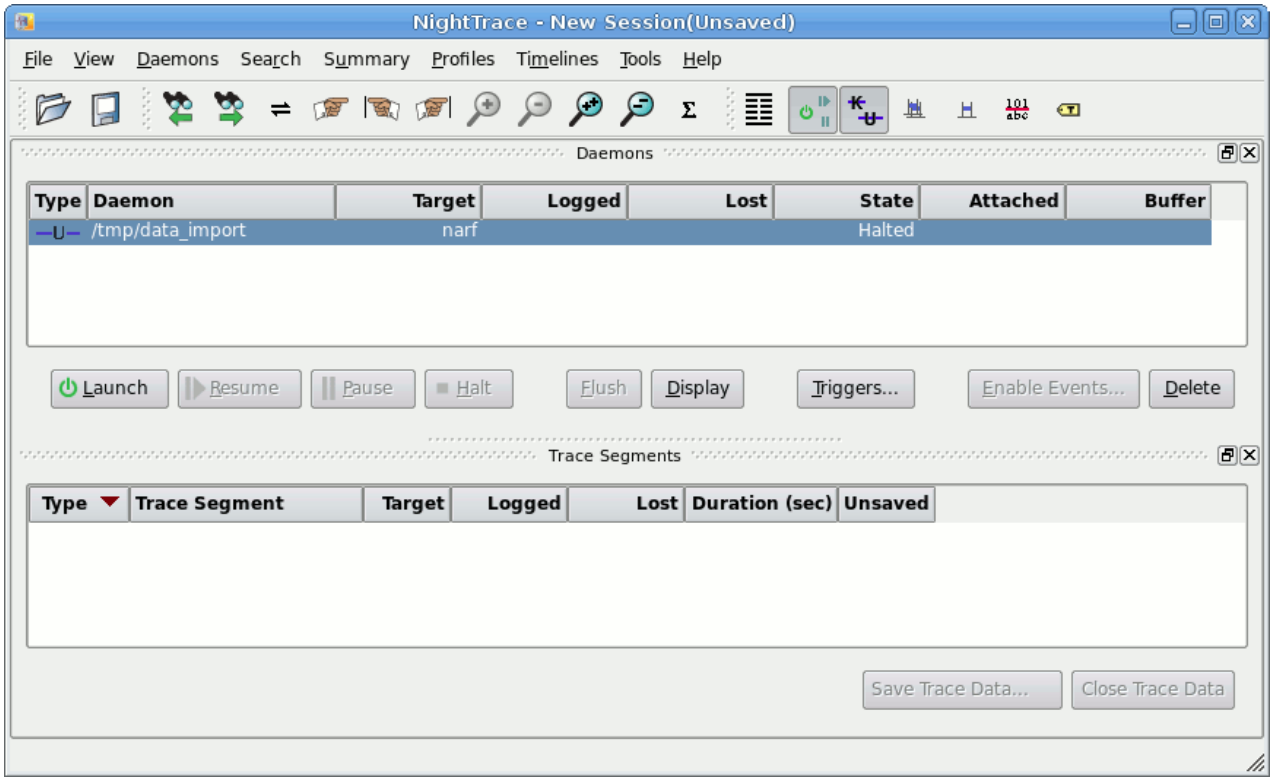


図 4-36. NightTrace - 起動可能なデーモン

Daemon の名称が**/tmp/data_import** であって単なるトレース・ファイルの名前ではないことに注目して下さい。これは単に--import オプションを介して取り込まれたことを示すために NightTrace がトレース・ファイル名に「_import」を加えて構成した名称です。

Daemon 行をダブルクリックすると表示されたダイアログはトレース・ファイルが**/tmp/data** であることを示します。

- Launch ボタンを押してデーモンを起動して下さい。
- Resume ボタンを押してトレース・イベントの収集を開始して下さい。

Daemons パネルに戻ると Buffer 列の数値が着実に増えているのでユーザー・デーモンがイベントを収集していることが確認できます。

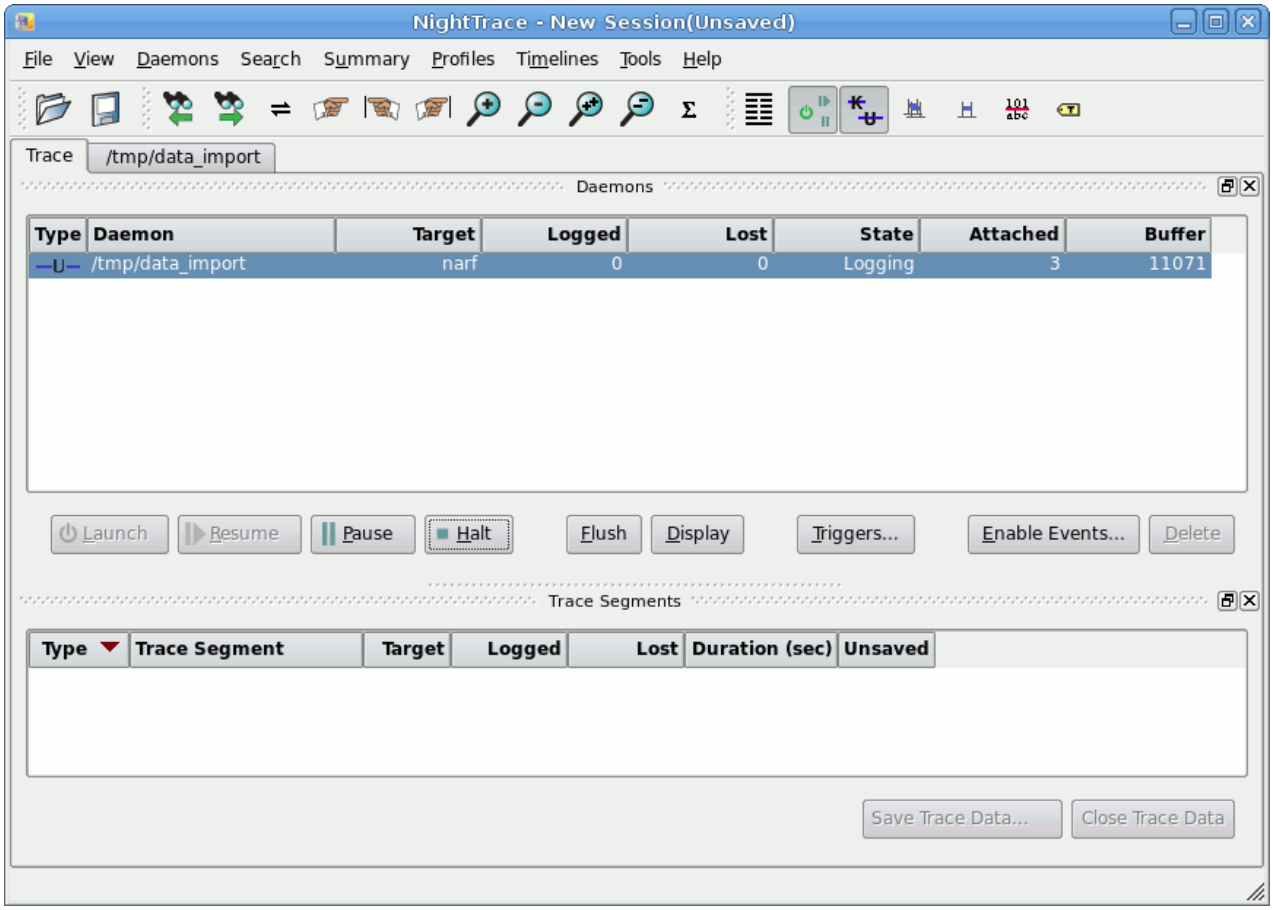


図 4-37. NightTrace - イベント収集中のデーモン

- Buffer 列のイベント総数が 10,000 以上に達するまで待って下さい。
- Daemons パネルの Halt ボタンを押してデーモンを停止して下さい。
- /tmp/data_import タブをクリックして NightTrace ウィンドウの最上位に Events および Timeline パネルを移動して下さい。
- タイムラインのグラフ・コンテナー領域の中央をクリックして Alt+Up を押下して下さい。

- タイムライン内のアクティビティの中央をクリックして個々のラインが見えてくるまで拡大して下さい。

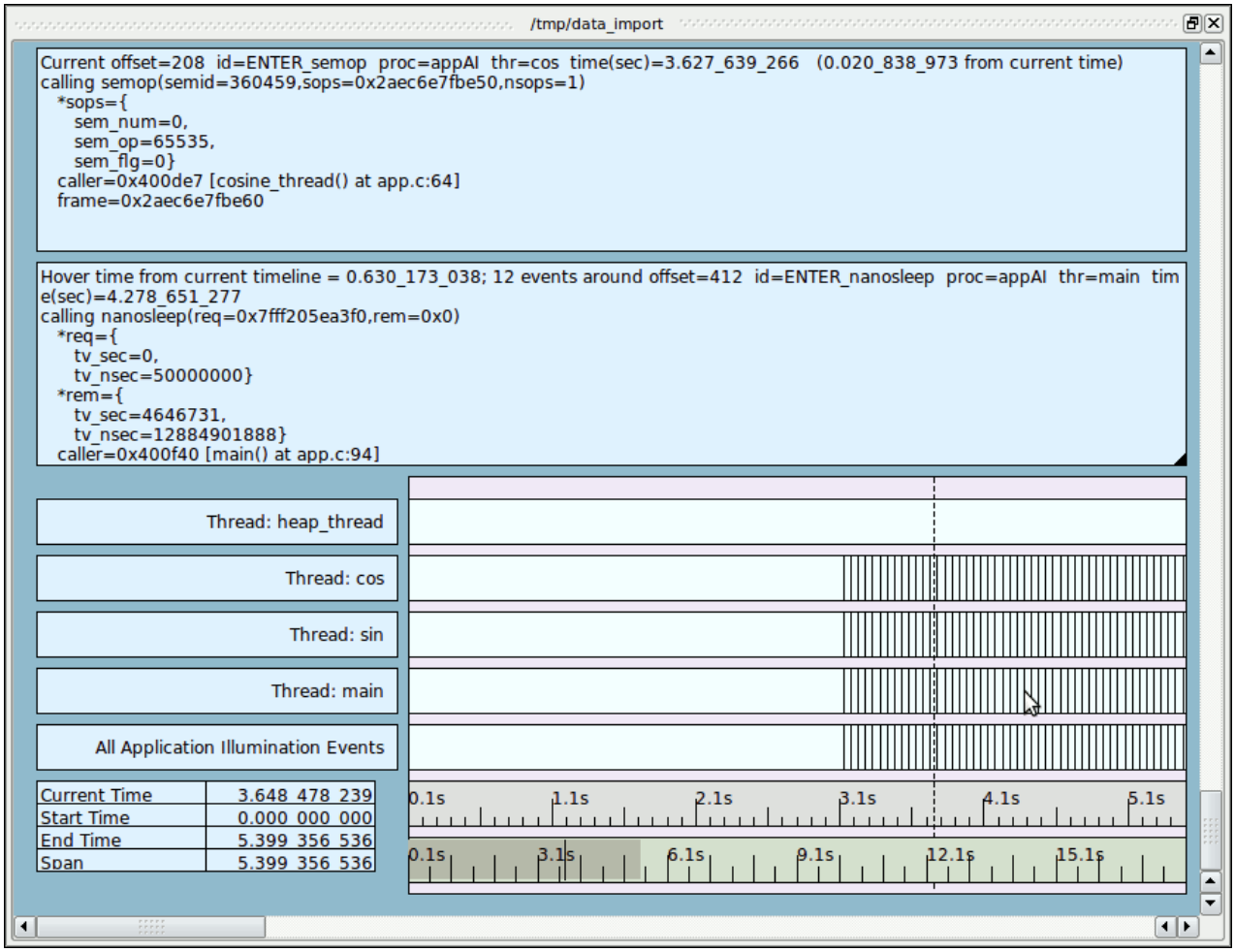


図 4-38. NightTrace - /tmp/data_import タイムライン

NOTE

NightTrace ウィンドウの縦サイズを拡大する必要がある場合は図に大まかに一致するまで/tmp/data_import パネルを引き上げて下さい。

AI タイムラインは標準的なユーザー・トレース・タイムラインとそっくりですが、イベント説明やポップアップ・ヒント説明が通常のトレース・データよりも更に詳細かつ冗長であるため、これらの領域が非常に大きくなります。

上図の中で、semop と nanosleep のライブラリ呼び出しの引数に関する詳細を含んだ説明に注目して下さい。

説明領域の右下隅にある黒の三角に気付いたかもしれません。これはコンテナー内に収めることが出来るよりも更に多くのテキストが利用可能であることを示しています。コンテナーのサイズを変更(右クリックして Edit モードを選択、コンテナーの角をつかんでサイズを変更して下さい)、

もしくは単にマウス・カーソルをコンテナーの上に移動するだけでポップアップが現れて完全なテキストを表示させることが可能です。

Events パネルに目を向けて下さい。

- 選択したイベントの説明領域の上にマウスを移動して下さい。

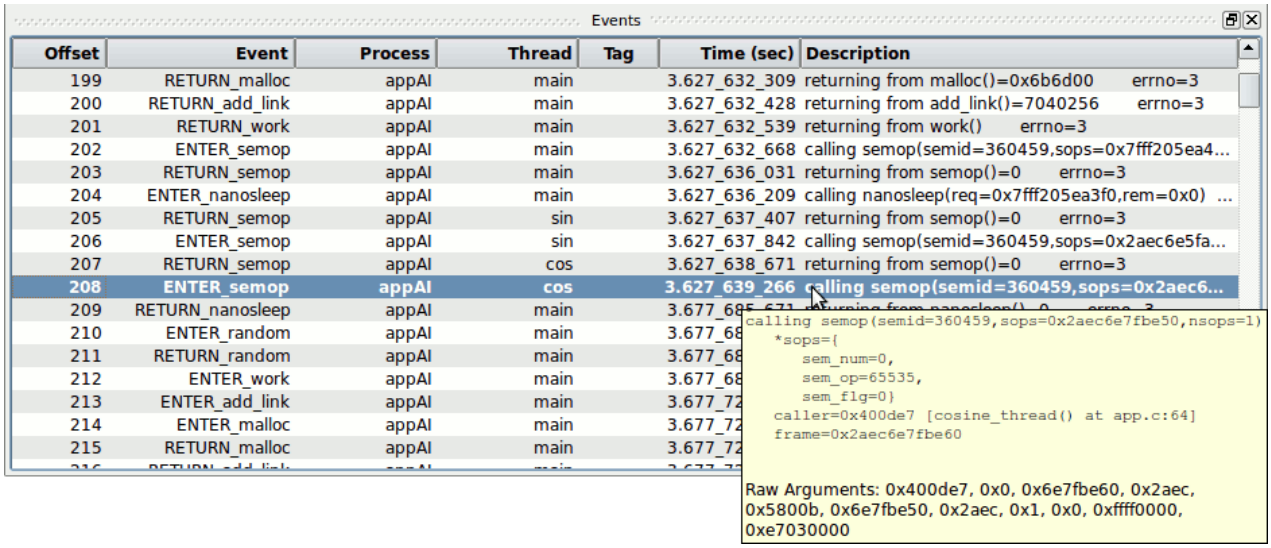


図 4-39. NightTrace - ツール・チップが表示された Events パネル

前述したようにトレース・イベントの説明は非常に長く、Events パネル内の最後の列の説明は切り捨てられているかもしれません。その領域にカーソルを移動することで完全な説明が提供されます。

- フォーカスが Events パネル内にある間に Ctrl+T を押下して Textual Search ダイアログを有効にしてください。

テキスト検索ダイアログが現れます。

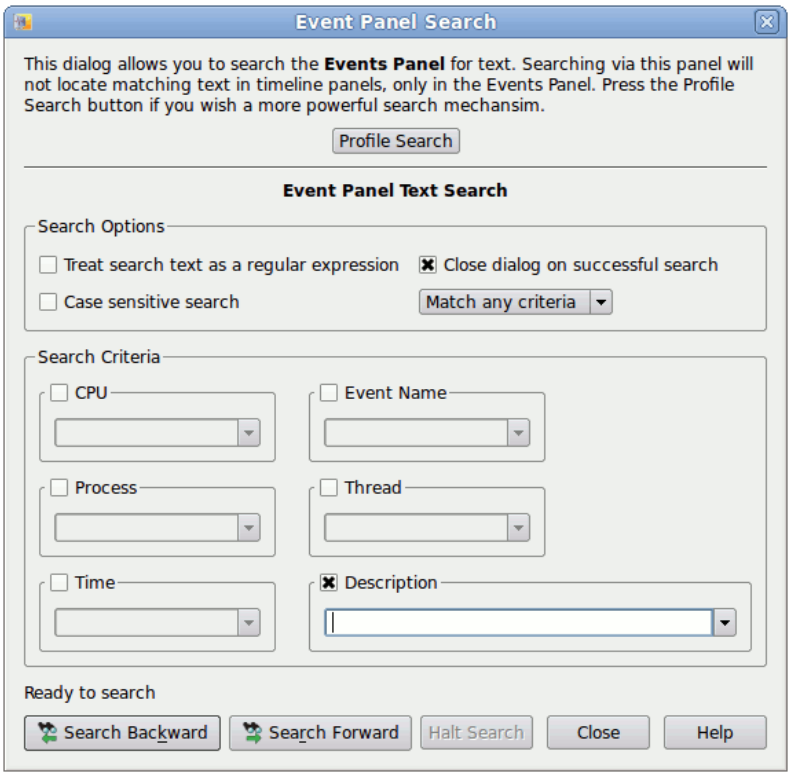
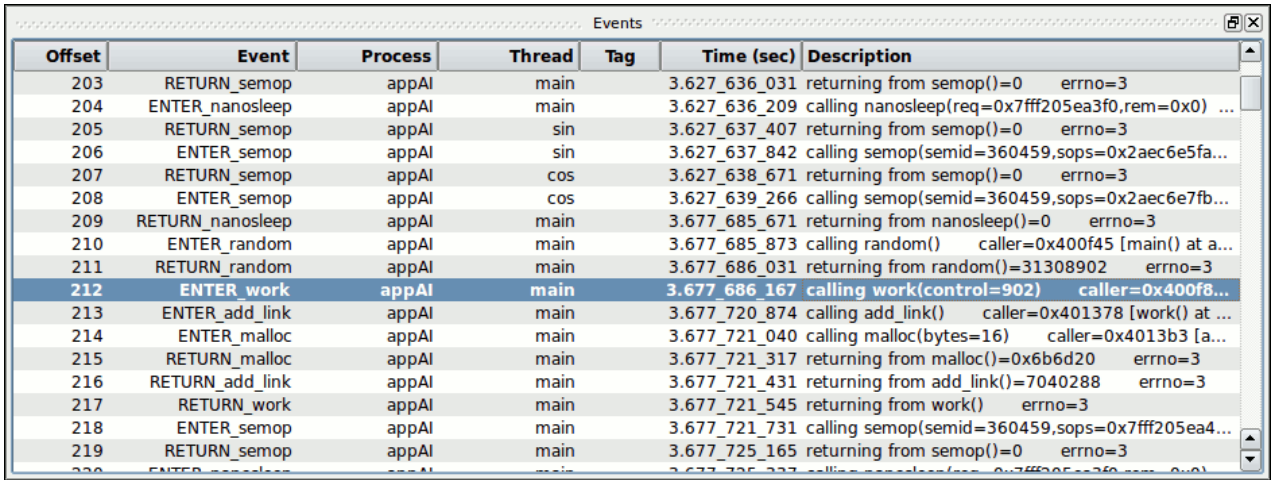


図 4-40. NightTrace - Event Panel Search ダイアログ

- Event Name フィールドのチェックボックスを ON にして有効にしてください。
- Event Name テキスト・フィールドに ENTER_work を入力して Search Forward ボタンを押して下さい。

Events パネルは現在 ENTER_work イベントの次の発生を選択しています。



Offset	Event	Process	Thread	Tag	Time (sec)	Description
203	RETURN_semop	appAI	main		3.627_636_031	returning from semop()=0 errno=3
204	ENTER_nanosleep	appAI	main		3.627_636_209	calling nanosleep(req=0x7fff205ea3f0,rem=0x0) ...
205	RETURN_semop	appAI	sin		3.627_637_407	returning from semop()=0 errno=3
206	ENTER_semop	appAI	sin		3.627_637_842	calling semop(semid=360459,sops=0x2aec6e5fa...
207	RETURN_semop	appAI	cos		3.627_638_671	returning from semop()=0 errno=3
208	ENTER_semop	appAI	cos		3.627_639_266	calling semop(semid=360459,sops=0x2aec6e7fb...
209	RETURN_nanosleep	appAI	main		3.677_685_671	returning from nanosleep()=0 errno=3
210	ENTER_random	appAI	main		3.677_685_873	calling random() caller=0x400f45 [main() at a...
211	RETURN_random	appAI	main		3.677_686_031	returning from random()=31308902 errno=3
212	ENTER_work	appAI	main		3.677_686_167	calling work(control=902) caller=0x400f8...
213	ENTER_add_link	appAI	main		3.677_720_874	calling add_link() caller=0x401378 [work() at ...
214	ENTER_malloc	appAI	main		3.677_721_040	calling malloc(bytes=16) caller=0x4013b3 [a...
215	RETURN_malloc	appAI	main		3.677_721_317	returning from malloc()=0x6b6d20 errno=3
216	RETURN_add_link	appAI	main		3.677_721_431	returning from add_link()=7040288 errno=3
217	RETURN_work	appAI	main		3.677_721_545	returning from work() errno=3
218	ENTER_semop	appAI	main		3.677_721_731	calling semop(semid=360459,sops=0x7fff205ea4...
219	RETURN_semop	appAI	main		3.677_725_165	returning from semop()=0 errno=3

図 4-41. NightTrace - 検索後の Events パネル

(説明の上にマウスを移動して見る)説明領域は 16 進数の PC 位置だけでなくサブプログラムとファイルの名称および行番号情報の両方を使った呼び出し元の位置を含んでいることに注目して下さい：

```
caller=0x400f83 [main() at app.c:98]
```

NOTE

コンパイラーのバージョンや実際のソースの中身にもよりますが、表示される行番号がその呼び出し後の次のコード生成ソース行に実際に関連付けられている可能性があります。それはトレース・イベントが含まれている PC の戻り値が「戻りアドレス」であるためで、その命令は呼ばれた関数の後に実行されます。

NightTrace は常に説明の呼び出し元の一部の PC アドレスをサブプログラムおよびファイル/行番号にマッピングしようとしますが、対応するルーチンがデバッグ情報付きでコンパイルされなかった場合はこの情報を提供することは出来ません。

ファイルや行番号がイベント説明で得られる場合、コンテキスト・メニューを使ってテキスト・エディタにソース行を表示するよう NightTrace に指示することが可能です。

- Enter_work イベントの説明をマウスで右クリックしてコンテキスト・メニューから Show Source File From Description…オプションを選択して下さい。

図 4-42. NightTrace - Events パネルのコンテキスト・メニュー

下図で示すように NightTrace はソース・ファイルをロードして適切な行番号にテキスト・エディタの位置を合わせます。

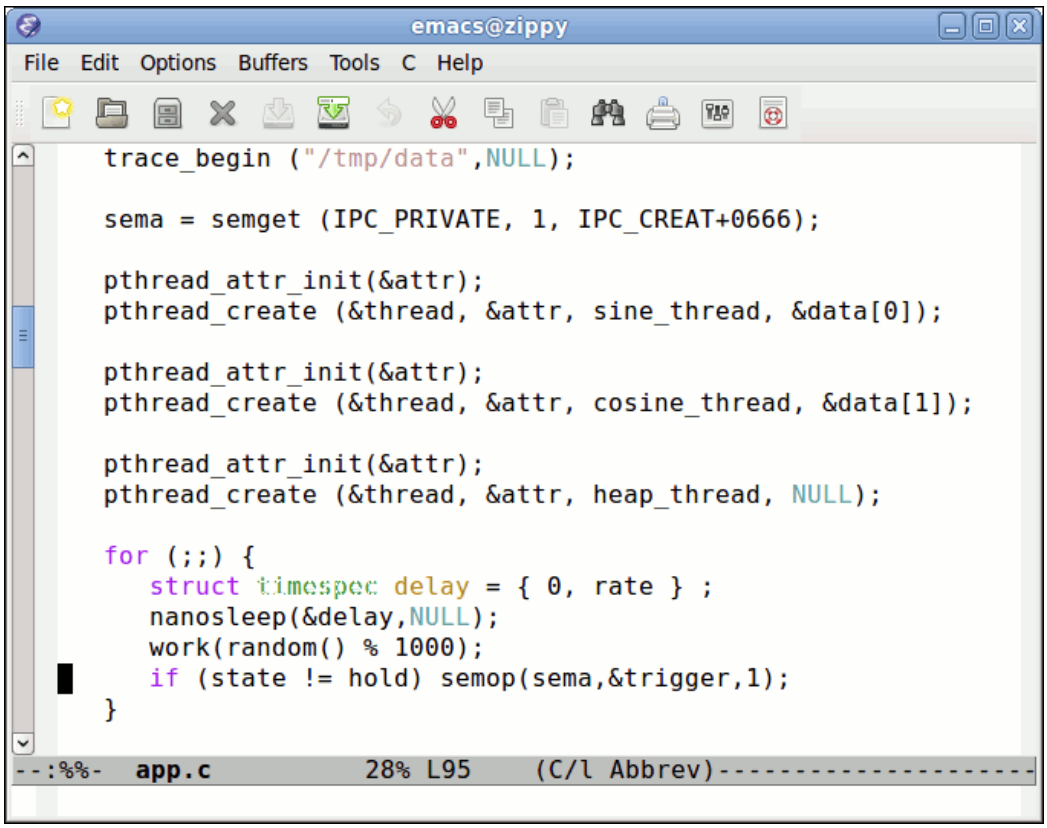


図 4-43. NightTrace - ソース・ファイルの行番号の位置でエディタを起動

NOTE

前述のとおり、PC の戻りは常に呼び出しの後の次の命令で、これは上の例のように次のソース行に関連付けられていることを指している可能性があります。

NOTE

NightTrace は EDITOR 環境変数を介してエディターを選択します。

- 進める前にエディターを閉じて下さい。

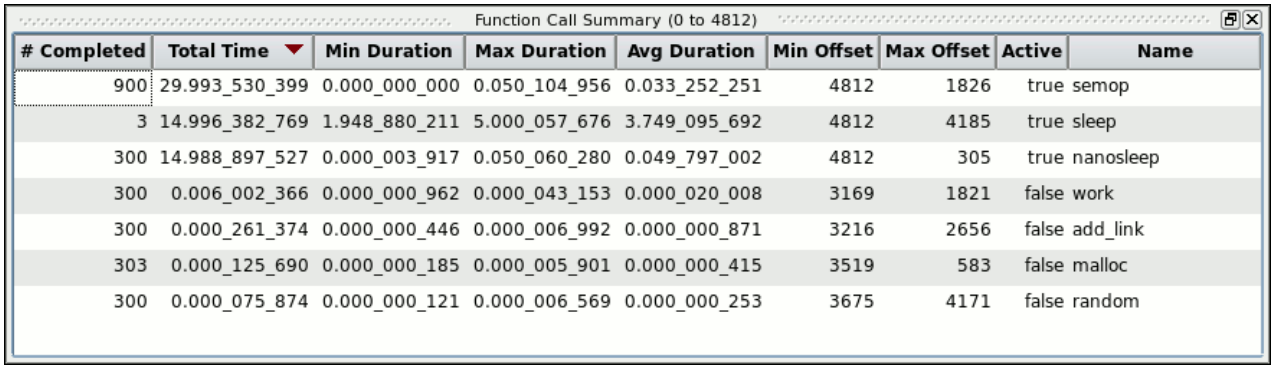
負荷性能のサマリー

本チュートリアルの前項でスレッドの負荷性能を要約したことを思い出して下さい。NightView を介して挿入したトレース・ポイントを使いそれらに対してステートを定義しました。

ここでも基本的に全く同じことをしますが、今回は **nlight** によって自動的に生成されたトレース・イベントだけを使用します。

- Summary メニューから Summarize Functions を選択して Summarize All Events サブメニュー・オプションを選択して下さい。

呼びだされた全てのトレース機能を備えた関数のサマリーを含むパネルが現れます。



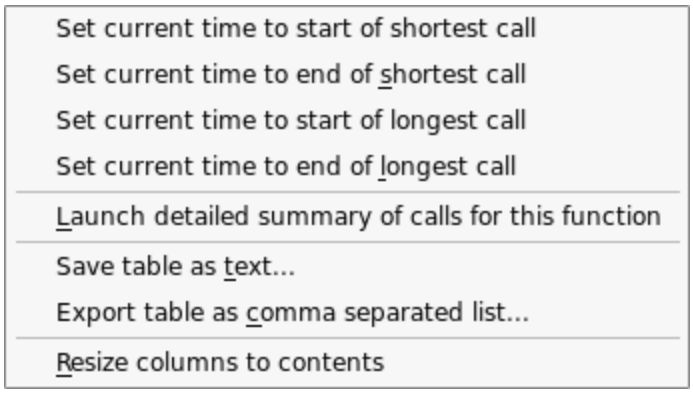
# Completed	Total Time	Min Duration	Max Duration	Avg Duration	Min Offset	Max Offset	Active	Name
900	29.993_530_399	0.000_000_000	0.050_104_956	0.033_252_251	4812	1826	true	semop
3	14.996_382_769	1.948_880_211	5.000_057_676	3.749_095_692	4812	4185	true	sleep
300	14.988_897_527	0.000_003_917	0.050_060_280	0.049_797_002	4812	305	true	nanosleep
300	0.006_002_366	0.000_000_962	0.000_043_153	0.000_020_008	3169	1821	false	work
300	0.000_261_374	0.000_000_446	0.000_006_992	0.000_000_871	3216	2656	false	add_link
303	0.000_125_690	0.000_000_185	0.000_005_901	0.000_000_415	3519	583	false	malloc
300	0.000_075_874	0.000_000_121	0.000_006_569	0.000_000_253	3675	4171	false	random

図 4-44. NightTrace - 関数のサマリー・テーブル

各トレース機能が備えられた関数に対して 1 つの行を提供するテーブルが生成されます。これは起動回数、最小、最大、平均の長さに関する統計値および関数の名称を含みます。

Active のラベルが付いた列はデータ・セットの最後(もしくは集約した間隔の最後)で関数呼び出しが継続していたかどうかを示します。

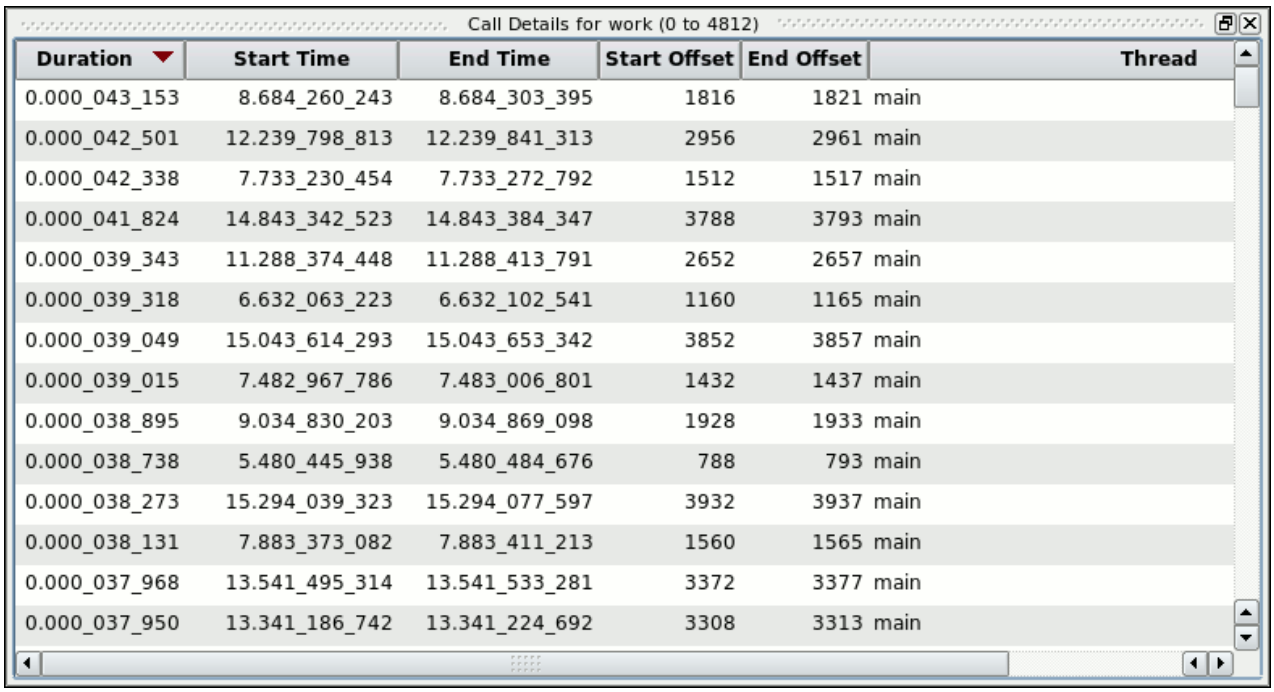
コンテキスト・メニューは以下の操作を提供します：



テーブルの行を右クリックすることで特定の関数の詳細を取得することが可能です。

- work 関数の行を右クリックして Launch detailed summary of calls for this function を選択して下さい。

テーブルはその関数の呼び出しごとに 1 行を使って表示されます。



Call Details for work (0 to 4812)

Duration	Start Time	End Time	Start Offset	End Offset	Thread
0.000_043_153	8.684_260_243	8.684_303_395	1816	1821	main
0.000_042_501	12.239_798_813	12.239_841_313	2956	2961	main
0.000_042_338	7.733_230_454	7.733_272_792	1512	1517	main
0.000_041_824	14.843_342_523	14.843_384_347	3788	3793	main
0.000_039_343	11.288_374_448	11.288_413_791	2652	2657	main
0.000_039_318	6.632_063_223	6.632_102_541	1160	1165	main
0.000_039_049	15.043_614_293	15.043_653_342	3852	3857	main
0.000_039_015	7.482_967_786	7.483_006_801	1432	1437	main
0.000_038_895	9.034_830_203	9.034_869_098	1928	1933	main
0.000_038_738	5.480_445_938	5.480_484_676	788	793	main
0.000_038_273	15.294_039_323	15.294_077_597	3932	3937	main
0.000_038_131	7.883_373_082	7.883_411_213	1560	1565	main
0.000_037_968	13.541_495_314	13.541_533_281	3372	3377	main
0.000_037_950	13.341_186_742	13.341_224_692	3308	3313	main

図 4-45. work 関数における関数詳細テーブル

このテーブルは Function Summary テーブルのコンテキスト・メニューと同じようなコンテキスト・メニューを持っています。

関数のバッチ・サマリー

全ての関数または特定の関数に関するサマリー情報を取得するために非 GUI モードで **ntrace** を使用することも可能です。

```
-killall appAI
```

前の手順で既にトレース・データを収集しましたが、そのデータを使用する代わりにコマンド・ラインだけを使ってゼロから収集する方法を以下に手順を示します：

```
./appAI &
ntraceud --join /tmp/data
sleep 5
ntraceud --quit-now /tmp/data
killall appAI
```

次のコマンドのどちらかを使い **ntrace** を起動することが可能です：

```
ntrace --verbose --summary=fs:* appAI /tmp/data
ntrace --verbose --summary=fs:work appAI /tmp/data
```

グラフィカル・インターフェースを提供することなく、上図で生成されたテーブルの中身と同じような出力を生成するでしょう。

シャット・ダウン

- NightStar の File メニューから Exit Immediately を選択して NightTrace セッションを終了して下さい

終了 - NightTrace

これで NightStar RT チュートリアルの NightTrace の部は終了です。

5

NightProbe の利用

NightProbe は単独で実行中のプログラムのデータを眺めるおよび修正するためだけでなく、その後の解析用にデータを記録するためのグラフィカル・ツールです。

本章は **app** プログラムを既にビルドしていることを想定しています。プログラムをビルドしていない場合、1-4 ページの「プログラムのビルド」の説明を使いビルドして下さい。

- 次のコマンドを発行して以前の app のインスタンスが実行中ではないことを確認して下さい：

```
killall -9 app
```

- 進める前に次のコマンドを介して新たにアプリケーションを開始して下さい：

```
./app &
```

NightProbe の起動

調査対象のプログラムは特別な API 呼び出しが実装されている必要はありません。しかしながら、NightProbe に関してはシンボリック変数名を参照するため、プログラムはデバッグ情報と一緒にコンパイル(通常は**-g** コンパイル・オプション)する必要があります。

NightProbe は RedHawk カーネルの著しい実行能力のアドバンテージを利用して、ダイレクト・メモリ・フェッチおよびストアを使い他のプログラムから変数のサンプリングや変更をすることでプロセスへの侵入を排除します。現在実行中のいずれの NightStar ツールの Tools メニューから NightProbe Monitor を選択して NightProbe を起動して下さい。NightProbe デスクトップ・アイコンを使って、または次のコマンドをシェル・コマンド・プロンプトで入力することで起動することも可能です：

```
nprobe &
```

NightProbe メイン・ウィンドウが表示されます。

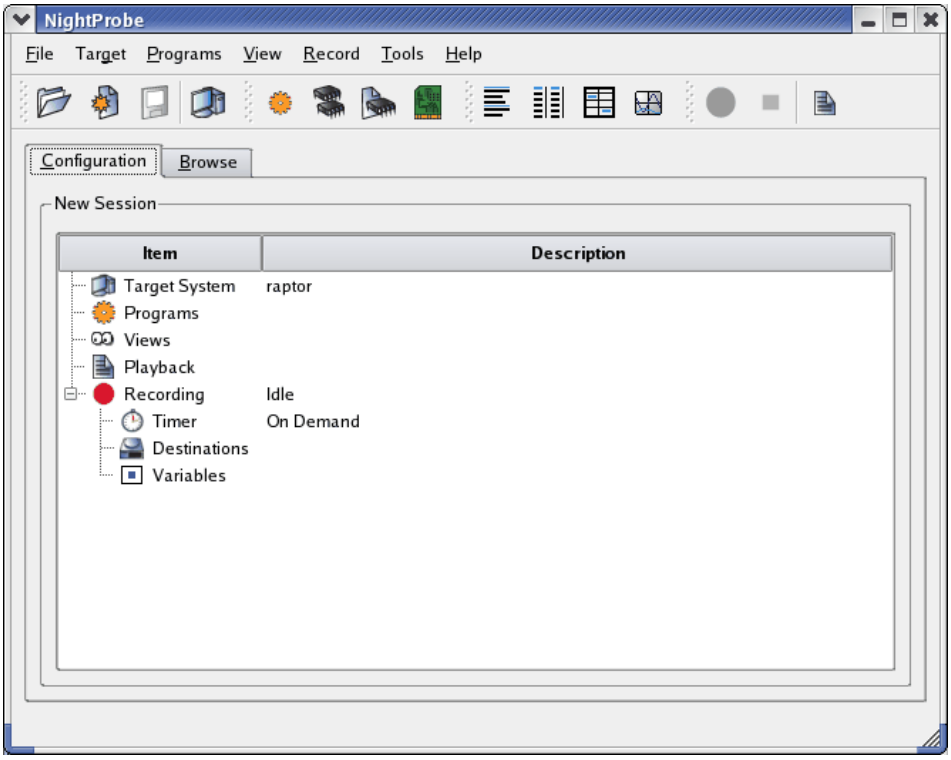


図 5-1. NightProbe メイン・ウィンドウ

プロセスの選択

NightProbe はプログラム、共有メモリ・セグメント、メモリ・マップされたエンティティ、PCI デバイスを含む数種類のリソースを調査する機能を持っています。

- Configuration ページの Programs アイコンを右クリックして Program...メニュー・オプションを選択して下さい。

Program Selection ダイアログが現れます：

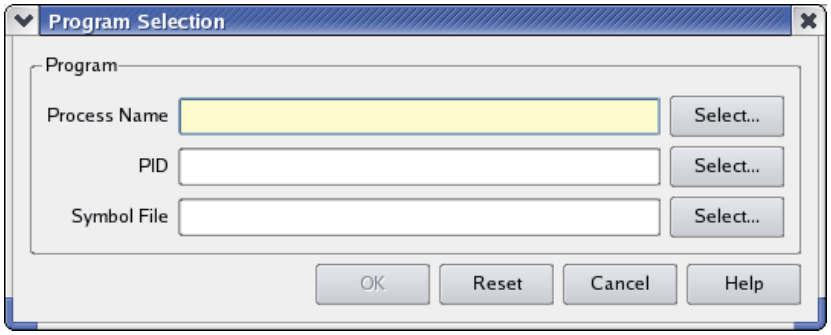


図 5-2. Program Selection ダイアログ

- PID フィールドの右にある Select...ボタンを押して下さい。

Process Selection ダイアログが現れます。

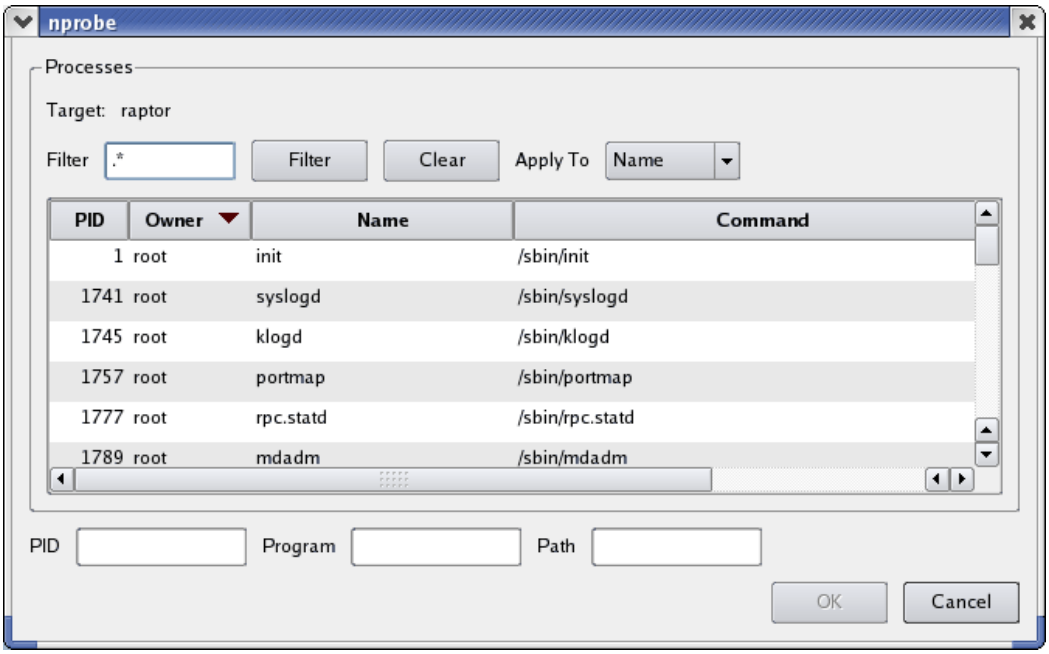


図 5-3. Process Selection ダイアログ

- Filter フィールドに^app を入力して Enter キーを押下して下さい。

リストは **app** で始まる名称のプロセスだけがフィルター処理されます。

- テーブルの中からプログラムを探して選択して下さい(一致するプログラムが 1 つしかない場合は既に選択されています)。
- Enter キーを再び押下してダイアログを閉じて下さい。

app プログラムに関連付けられたプロセス ID は PID テキスト・フィールドに配置され、Process Name および Symbol File テキスト・フィールドはそれに応じて更新されます。

- OK をクリックしてダイアログを閉じて下さい。

app プログラムは Configuration ページ内の Programs 項目の下に表示された状態で調査するリソースのリストおよび Browse タブ内部のツリーに追加され、そのタブは自動的に呼び出されます。

ライブ・データの閲覧

- Browse タブが選ばれていることを確認して下さい(そうではない場合はクリックして下さい)。

Live Browser が Browse タブの内側に表示されます。

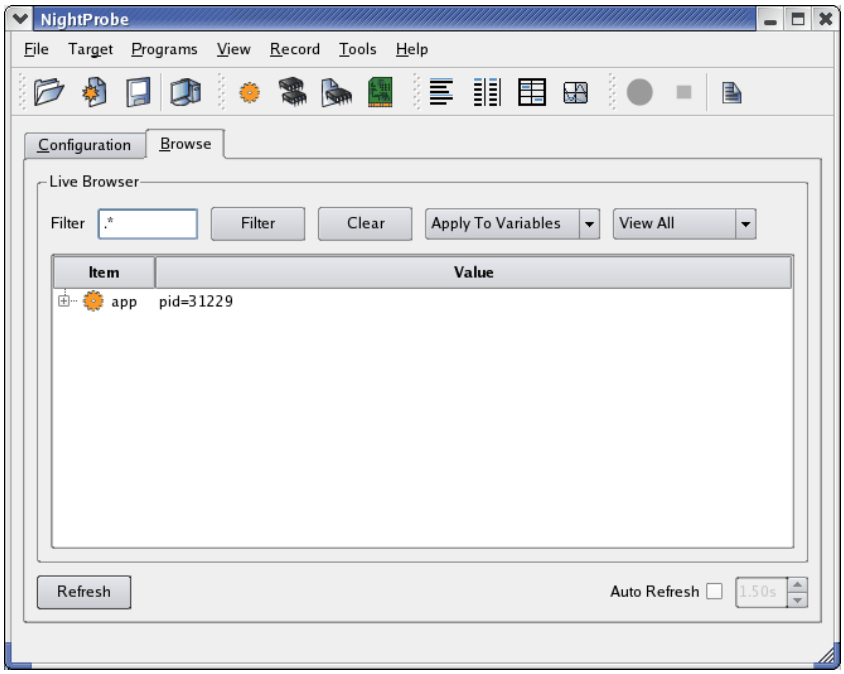


図 5-4. NightProbe の Browse パネル

Browse ページは 2 つの目的を受け持ちます。記録または View パネルに代わって調査する対象の興味のある変数をプログラムを見て選択することを許可します。

また、Browse ページ内に直接表示されているツリーを使って変数の即席表示も提供します。

- ツリー内の app エントリーを展開して下さい。

プログラムのアイコン下にある項目は全グローバル変数だけでなく Ada パッケージなどのネストされたスコープ、または静的データ項目を含む関数を含みます。

各変数項目は変数がスカラー、ポインタ、または配列や構造体などの複合項目であるかどうかを示すアイコンを持っています。

data 変数は複合オブジェクトで展開することが可能です。

- data 変数を展開して下さい。

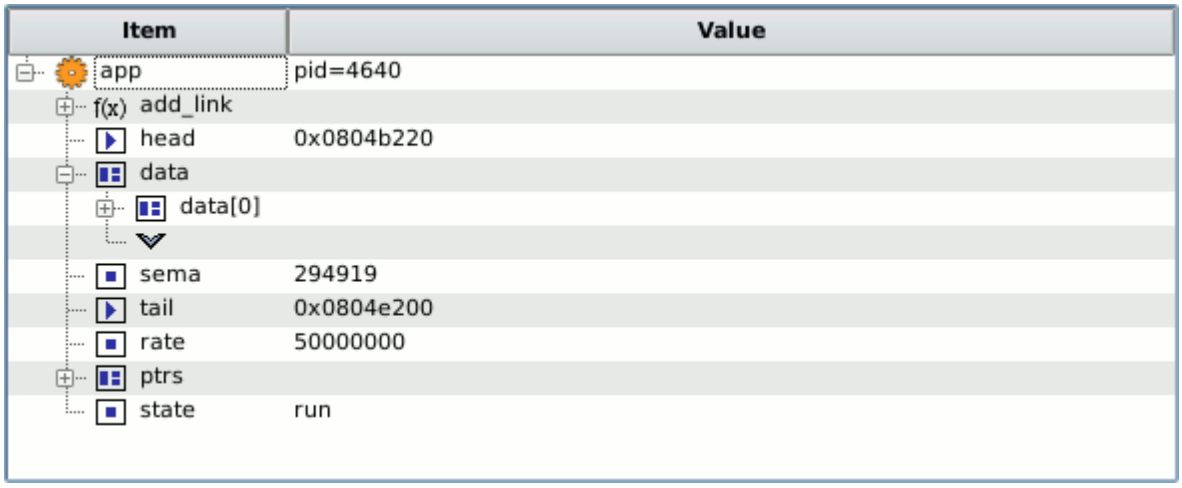


図 5-5. 展開された data 項目

下向き矢印の先端は配列の添え字展開アイコンです。そのアイコンのクリックで配列の追加コンポーネントが表示されます。

- 配列展開アイコンをクリックして data[1]を表示して下さい。
- 表示されている両方の構造体 data[0]と data[1]を展開して下さい。

Browse ページにおいてツリー内に表示されている全ての変数の現在値が、ページ下部にある Refresh ボタンを押すたびに、Auto refresh チェックボックスで制御される自動リフレッシュが発生するたびに表示されます。

- Auto Refresh チェックボックスをクリックして下さい。

これで Auto Refresh チェックボックスの右側にあるスピンボックスで示されるレートでディスプレイを自動的にリフレッシュさせます。

データ配列の各コンポーネントの count, angle, value コンポーネントの値が変化していることに注目して下さい。

変数の変更

app メイン・プログラムは処理を行うために各スレッドを反復的に起こします。state 変数はこれを発生すべきかどうかを制御します。

state 変数の現在値は列挙値 run であることに注意して下さい。

state 変数の値をダブルクリックして下さい。

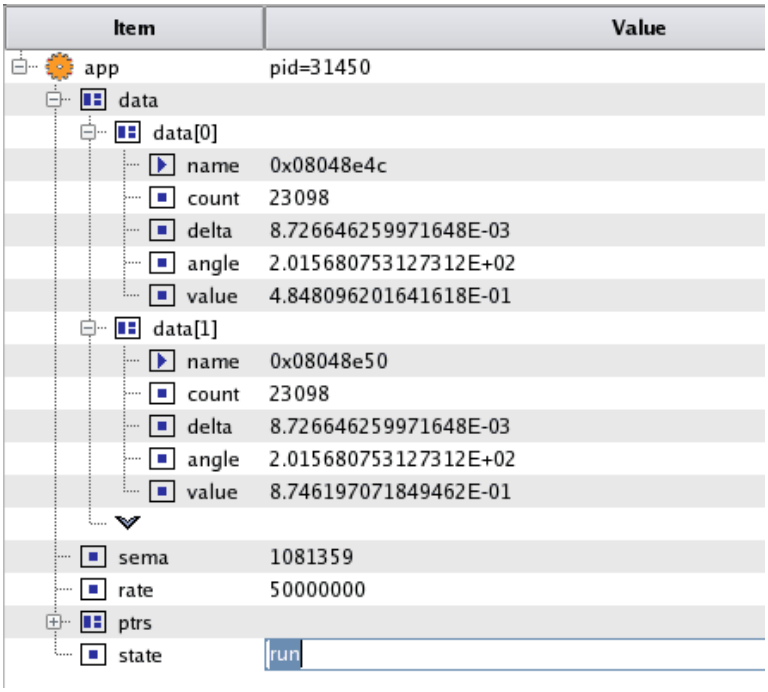


図 5-6. 変数の修正中

値を含んでいるセルは更新が停止され現在の値が選択されます。

変数の値を変更するには、必要なことは新しい値を提供してプログラムにその変更を委ねるだけです。

- セルに以下を入力して下さい：

  hold

- Enter キーを押下してプログラムに値を委ねて下さい。

state 変数の値は現在、**app.c** のソース・コードの抜粋が示すようにプログラムに計算用スレッドを起動させない hold です：

```
95 for (;;) {
96            struct timespec delay = { 0, rate };
97            nanosleep(&delay,NULL);
98            work(random() % 1000);
99            if (state != hold) semop(sema.&trigger,1);
100}
```

- hold と表示されている値をダブルクリックし後にオプション・リスト・アイコン(値の行の右端に表示されている下向きの山形紋)を使って run を選択し Enter を押下して state 変数の値を元の run に変更して下さい。

レコーディングおよびオプション閲覧用に変数を選択

各変数は Mark と Record の属性を持っています。Mark 属性は(設定した場合)特に関心のある変数であることを示し他のパネルでも表示させることが可能になります。Record 属性はその変数が記録セッションに含まれることを明示しています。

項目をダブルクリックするとその色を赤色系に変えて Mark と Record 属性を設定します。あるいは、属性を個別に設定するために項目のコンテキスト・メニューを使用することも可能です。

- `data[0]` と `data[1]` 構造体の両方から名称列の `count`, `angle`, `value` フィールドをダブルクリックして下さい。
- `rate` 名称列の変数をダブルクリックして下さい。

Browse ページのツリーは次のように見えるはずです：

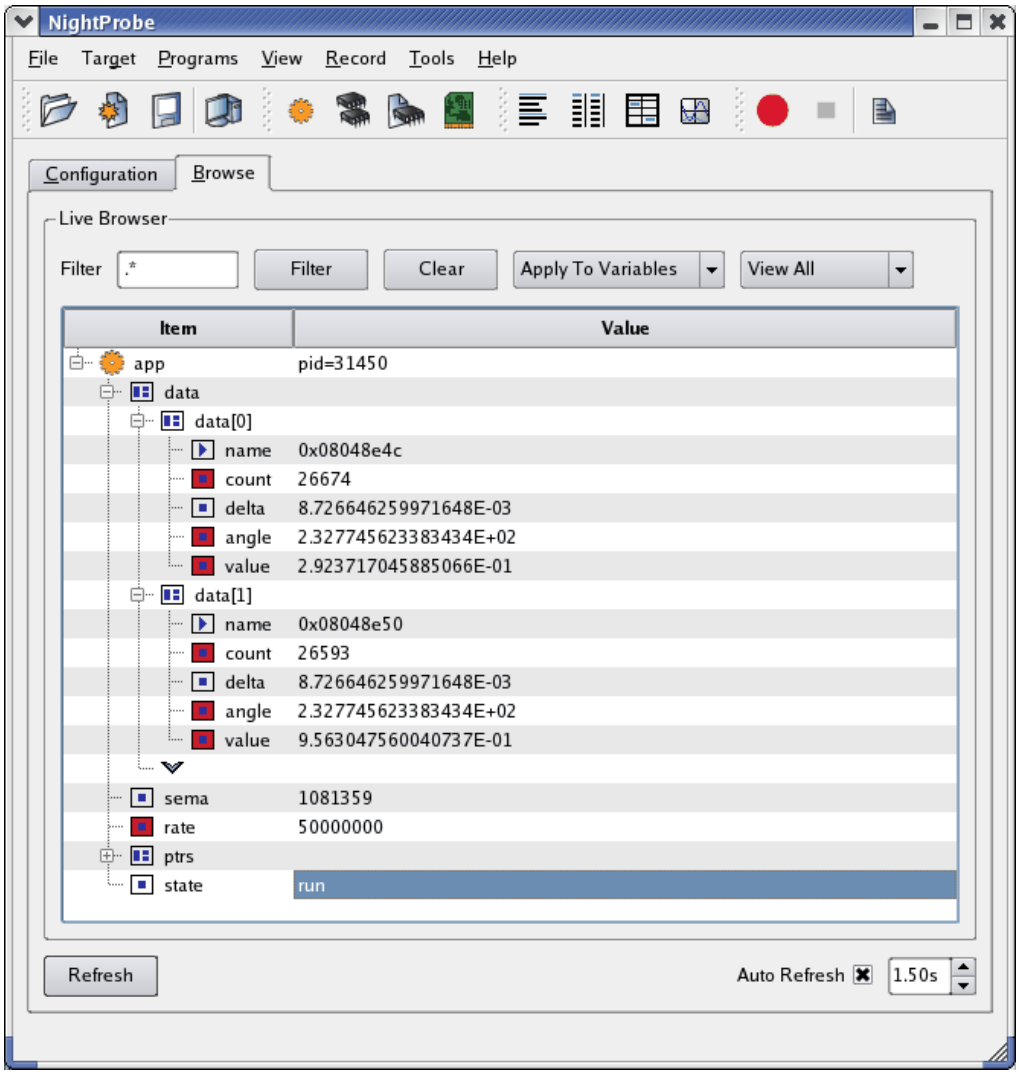


図 5-7. Mark と Record 属性の設定

閲覧方法の選択

NightProbe はデータの閲覧に様々な方法を提供します：

- Browse ページ
- リスト・ビュー
- テーブル・ビュー
- スプレッドシート・ビュー
- グラフ・ビュー

また、記録セッションの出力をライブ解析のために NightTrace またはユーザー・アプリケーションへ、NightProbe でのその後の解析のためにファイルへ流すことも可能です。

テーブル・ビュー

テーブル・ビューは時間と共に変数が縦方向と横方向に変数の値を含んで広がるスクロール可能なテーブルを提供します。

- View メニューから Table オプションを選択して下さい。

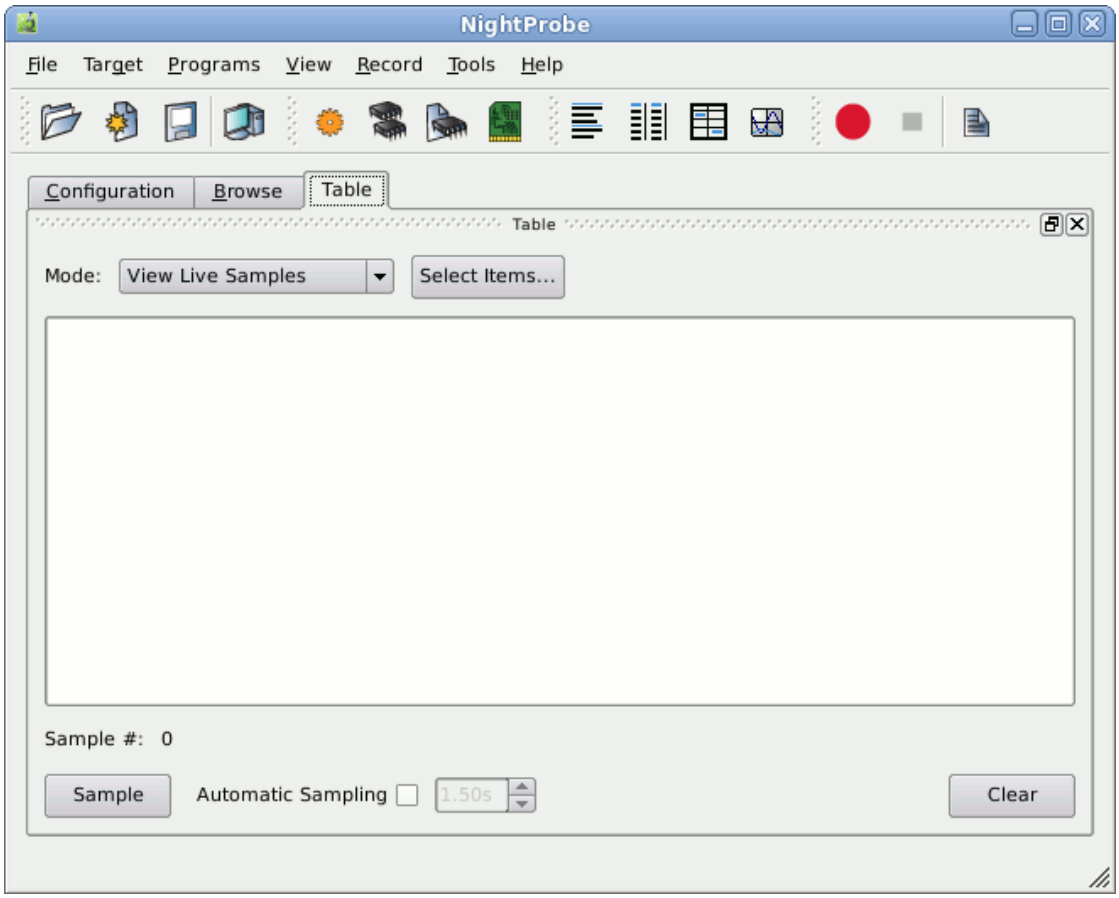


図 5-8. Table ビュー

最初はテーブルは空です。最初のステップはテーブルに表示させたい項目を選択することです。

- Select Items...ボタンを押して下さい。

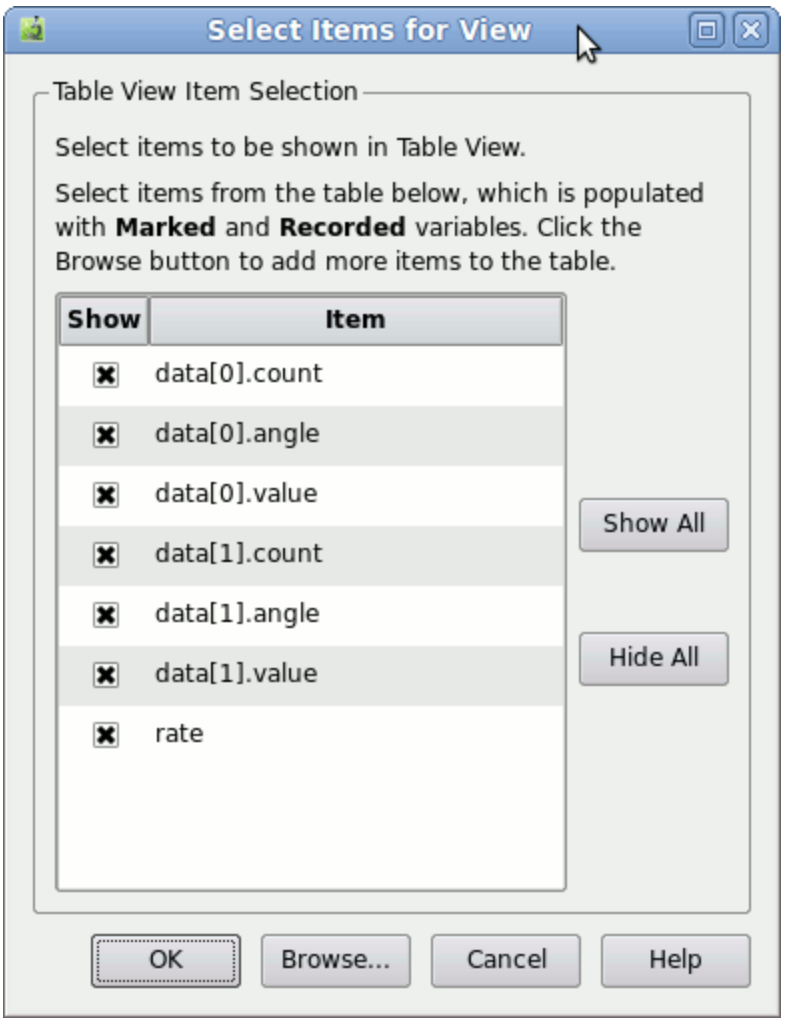


図 5-9. 項目選択ダイアログ

本ダイアログでは Mark または Record 属性が設定された項目を選択することが可能です。

デフォルトで、ダイアログは前述の変数を表示するための初期値を構成します。

- data[1]コンポーネントの全ての要素を Show 列のそれらの行をクリックして非表示にしてください
- OK ボタンを押して下さい。

テーブルは現在 5 つの列があり、サンプル番号用に 1 つ、前の手順で選択した各々の変数に 1 つずつとなります。

- Automatic Sampling チェックボックスを ON にして下さい。

Automatic Sampling チェックボックスの右にあるスピンボックスに定義されたレートでサンプルがテーブル内の変数に取り込まれます。

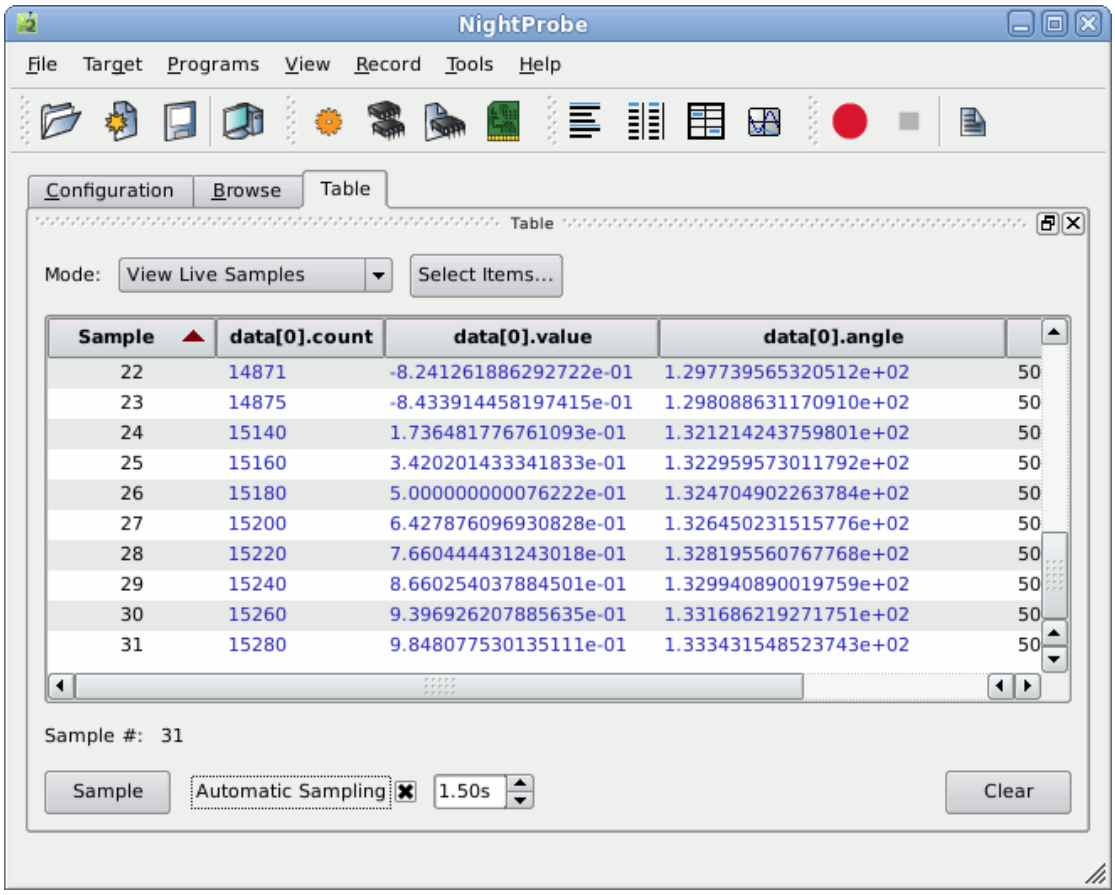


図 5-10. 自動サンプリング・モードの Table

値が前回のサンプリングから変わった場合は青で表示されます。

列見出しをクリックすることで変数値でソートすることが可能です。

- Automatic Sampling チェックボックスを OFF にして下さい。
- data[0].value の列見出しをクリックしてから再度クリックするとテーブルは最大値から最小値にソートされます。

十分なサンプルが取り込まれた場合はトップに表示される値はほぼ 1.0 となるはずです (data[0].value の値は正弦波の値です)。

5-5 ページの「変数の変更」で説明されているのと同じ方法で Table ビューを使い変数を修正することが可能です。ここでの違いはクリックしたセルは既に取り込んだサンプルの値であることです。セルの値を変更した場合、変数の値はプログラム内で直ちに変わりますが、セルは以前のサンプリング値に戻ります。

新たなサンプルは修正の効果を示します。

- 最小から最大にソートされるまで Sample 列見出しをクリックして下さい。

- Automatic Sampling チェックボックスを ON にして下さい。
- スクロールバーのボックスをクリックしてスクロールバーの底までドラッグしたら離して下さい。

新しい値が再びテーブルの底に表示されます。

グラフ・ビュー

Graph パネルは個々の変数が単独の線としてグラフ上に描かれます。

- View メニューから Add New Page オプションを選択して下さい。
- View メニューから Graph オプションを選択して下さい。

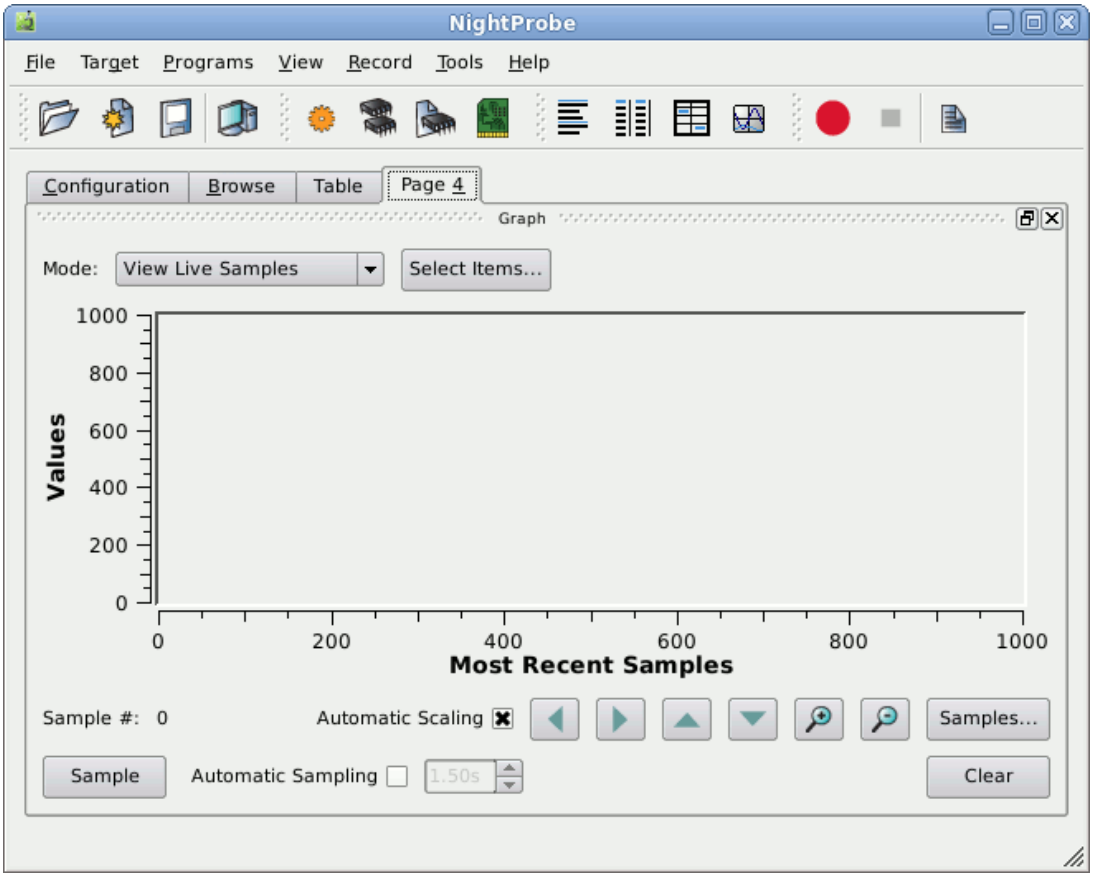


図 5-11. Graph パネル

最初はテーブルは空です。

- Select Items...ボタンを押して下さい。

テーブル・ビューとは異なり、Select Item ダイアログ内のどの項目も選択されずに表示されます。通常は、1 つだけまたはごく少数の項目を 1 つのグラフ上に表示します。

- Show 列の data[0].value と data[1].value 項目のそれぞれの行をクリックして印をつけて下さい。
- OK ボタンを押して下さい。
- Automatic Sampling チェックボックスが ON であることを確認して下さい。
- Automatic Sampling チェックボックスの右側のスピンボックスで更新レートを 1.0 秒に変更して下さい。

下図のように 2 つの線の描画が始まります。

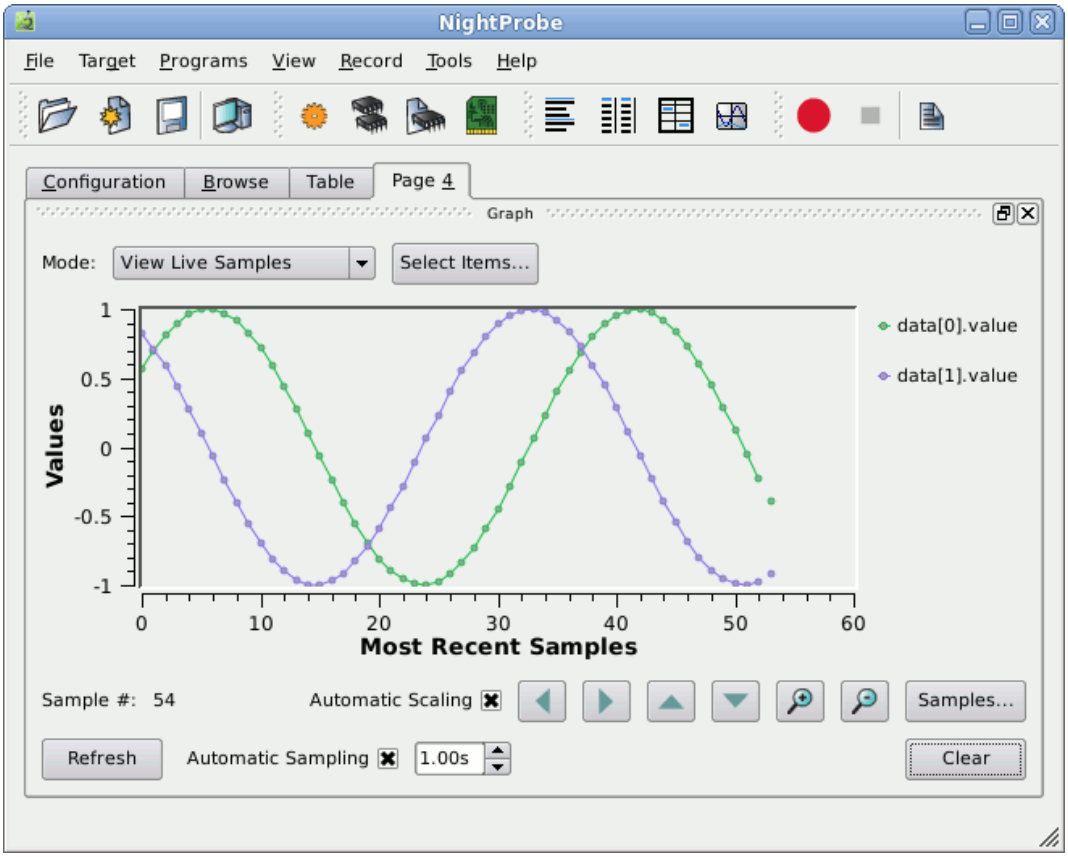


図 5-12. 活発に値を表示する Graph パネル

- グラフ・パネルの右側にある凡例のいずれかの value 項目の(右クリックで起動する)コンテキスト・メニューから Edit...オプションを選択して下さい。

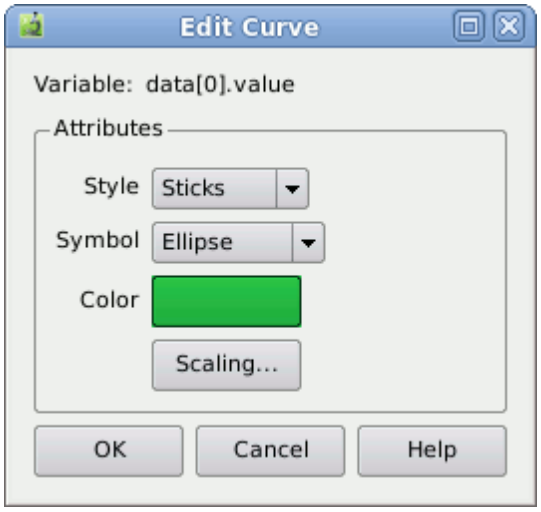


図 5-13. Edit Curve 属性ダイアログ

- Style オプション・リストから Sticks を選択して下さい。
- 色のついたブロックをクリックして色選択ダイアログを起動し色を変更して下さい。
- OK ボタンを押して色選択ダイアログを閉じて下さい。

- OK ボタンを押して Edit Curve 属性ダイアログを閉じて下さい。

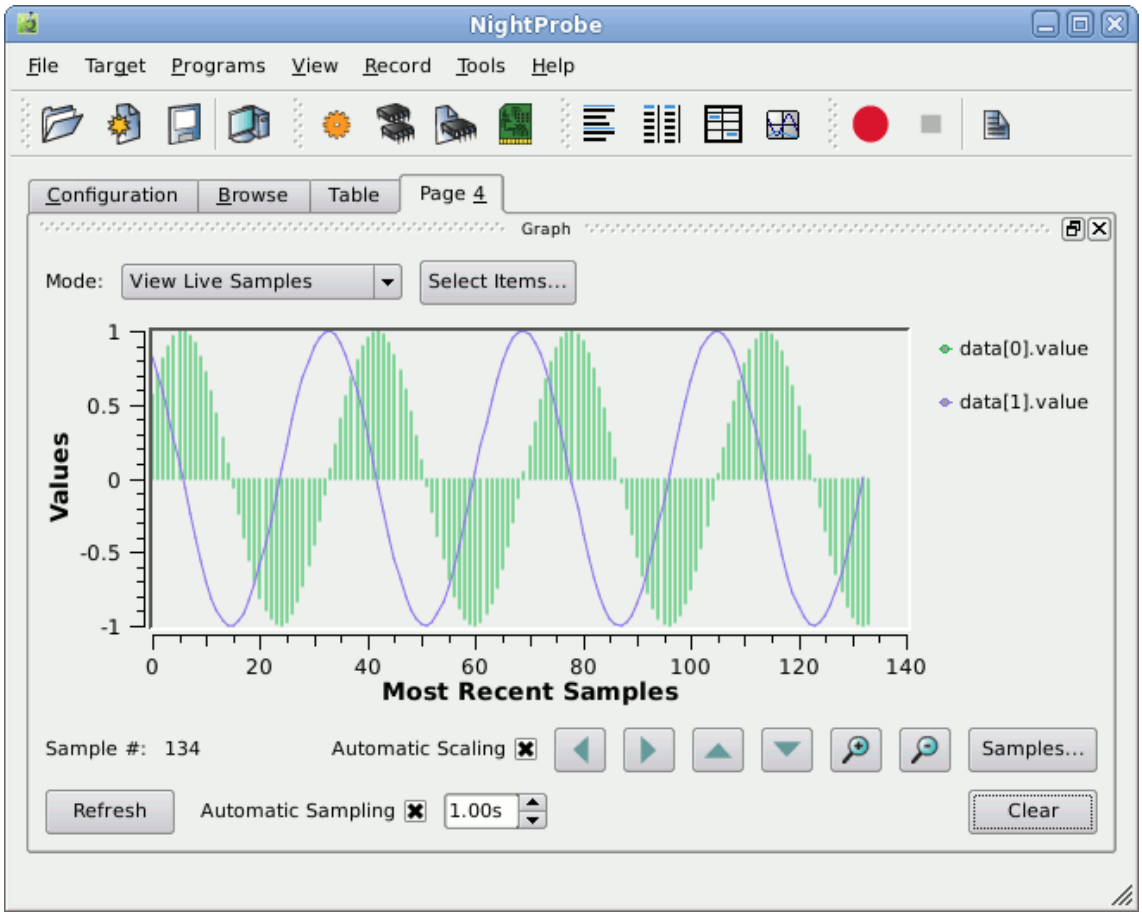


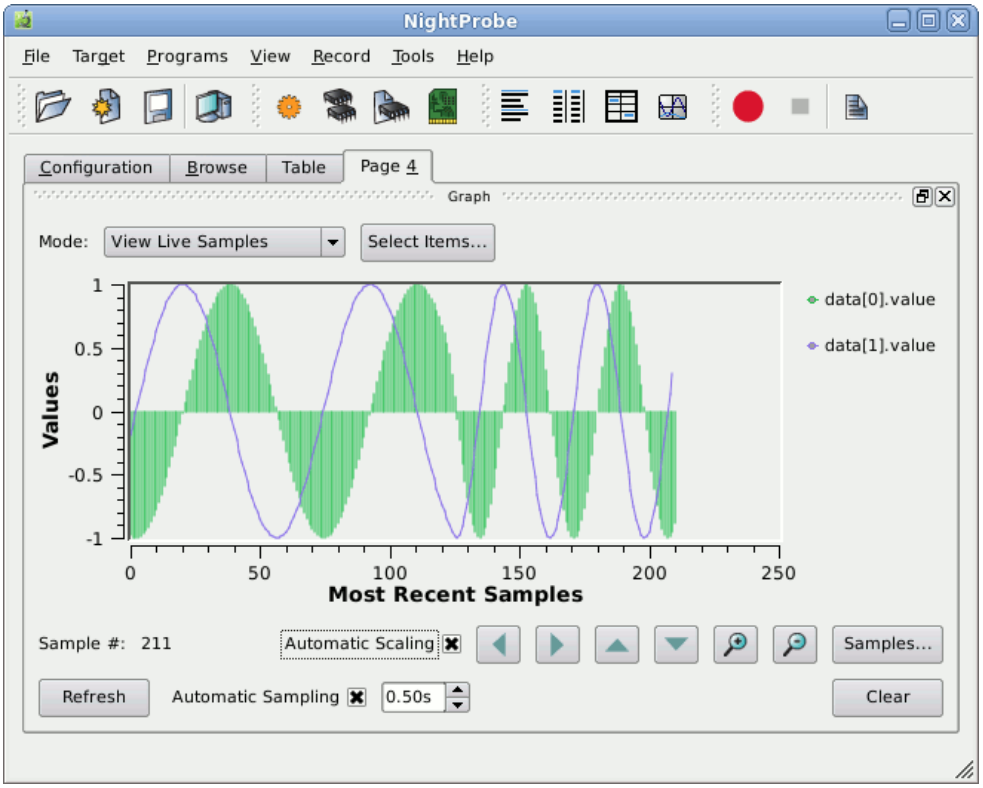
図 5-14. 変更された曲線を含む Graph パネル

- Automatic Scaling チェックボックスを ON にして下さい。
- refresh rate を 0.5 秒に変更して下さい。

プログラムはスレッドがそれぞれの計算を行うために起動する頻度を rate 変数を使用して決定します。

- Browse ページまたは Table パネルを使って `rate` 変数を 50000000 から 25000000 に変更して下さい。

この変更は実際にスレッドが動作する頻度が 2 倍になるので、正弦波と余弦波の外形が変わります。



他のプログラムへ取得データを送信

データの値はその後の処理用にファイルへ記録する、またはライブ処理用に記録して NightTrace に流し込むことが可能です。

同様に記録したデータを任意のプロセスに送信することが可能です。

- Configuration ページのタブをクリックして選択して下さい。

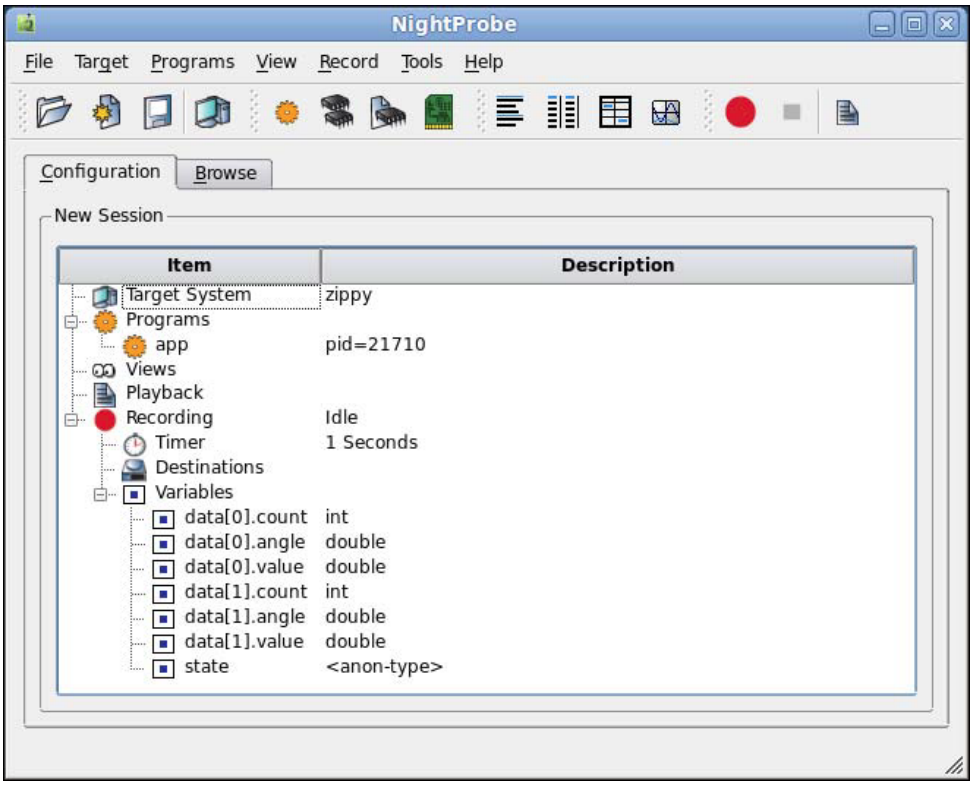


図 5-15. Configuration ページの Recording 領域

構成ツリーの Recording 部分は記録用 Timing ソース、記録先 Destinations、Record 属性が設定された変数のリストを表示します。

- Recording ツリーの Timer 項目を右クリックして Timer サブメニューから Clock...オプションを選択して下さい。

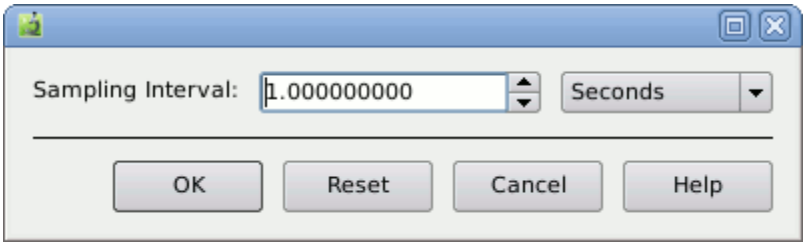


図 5-16. クロック選択ダイアログ

本ダイアログは記録するサンプルを取り込むレートを制御します。

- 単位を Sampling Interval オプション・リストから Milliseconds に変更して下さい。
- Sampling Interval の値を 100.0 に変更して下さい。

- OK ボタンを押して下さい。

ツリー内の Timer 項目の説明がこの操作を反映して変更されます。

記録先はユーザー・アプリケーションになります。

- Destinations 項目を右クリックして To Program...を選択して下さい。

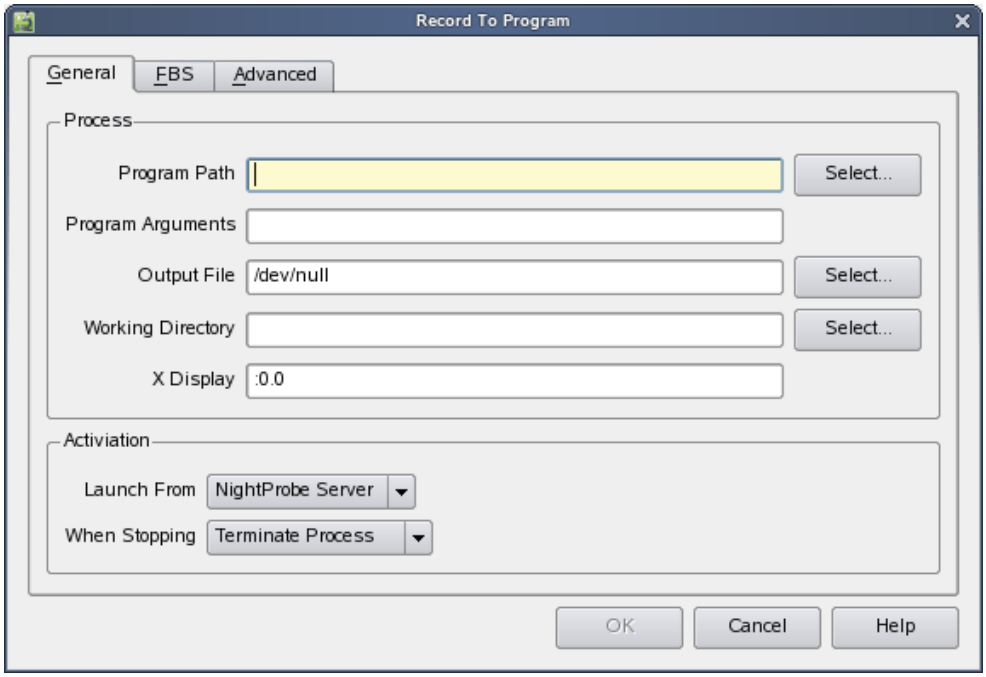


図 5-17. Record To Program ダイアログ

- Program Path テキスト・フィールドに **api** と入力して下さい。
- Output File テキスト・フィールドのテキスト**/dev/null** を以下に置き換えて下さい。

```
/tmp/api.out
```

- OK ボタンを押して下さい。

記録したサンプルの値を取り込んで表示するための NightProbe API を使用する簡素なアプリケーションは、1-4 ページの「チュートリアル・ディレクトリの生成」で **tutorial** ディレクトリにコピーされています。

- ターミナル・セッションで以下のコマンドを入力してプログラムをビルド下さい：

```
cc -g -o api api.c –lnprobe
```

Configuration ページの Recording 領域は以下のように見えているはずです。

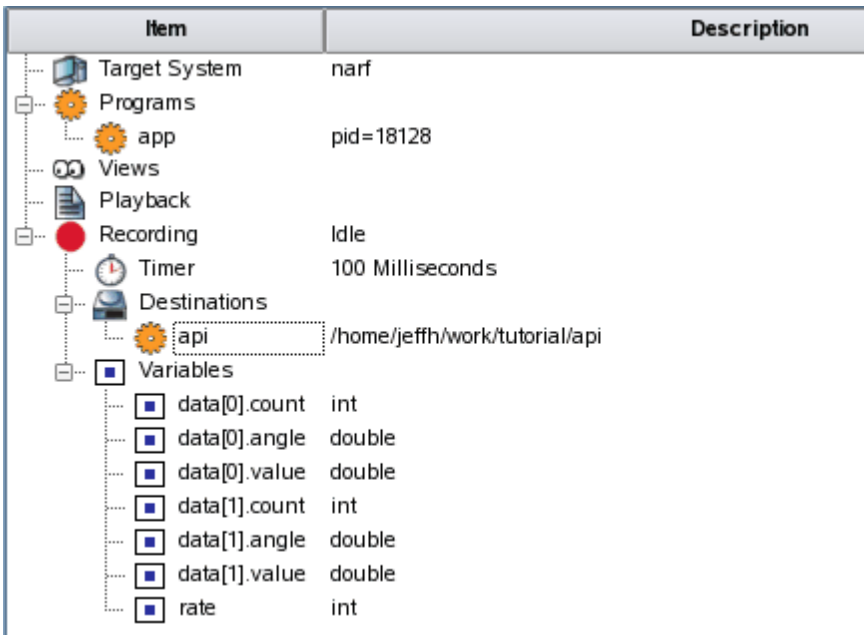


図 5-18. 宛先を含む Configuration ページの Recording 領域

これで記録する変数、記録するタイミング・ソース、記録先を選択しましたので、サンプルを記録してそれらを **api** アプリケーションへ流すことが可能です。

- ツールバーの Record アイコンを押して下さい。

- サンプルを記録してそれを渡しながら api プログラムの出力を表示して下さい。
- ターミナル・セッションで以下のコマンドを入力して下さい：

```
tail -f /tmp/api.out
```

プログラムは以下のような出力を生成します：

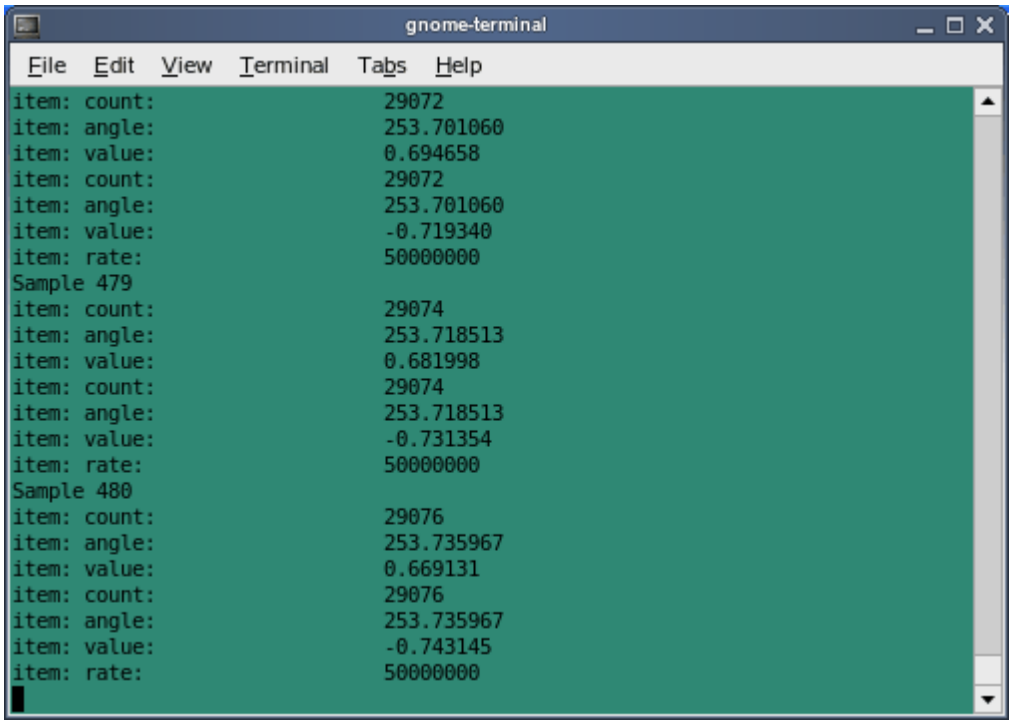


図 5-19. api プログラムの出力例

- Recording ツールバーの Stop アイコンを押して記録処理を停止して下さい：

NightProbe API に関する詳細については、*NightProbe User's Guide* の「NightProbe API」章を参照して下さい。

プログラム変数の変更に Datamon を利用

Data Monitoring Application Programming Interface は NightStar RT ツール・セットの一部です。

変数名を指定して選択した変数の値を監視、取得、修正、およびアドレス、型、サイズのような変数に関する情報を取得するために Data Monitoring は Ada, C 言語、Fortran の変数を含む実行可能プログラムを指定することが可能です。

NOTE

Ada プログラムは、適切な DWARF デバッグ情報を生成する Concurrnet MAXAda コンパイラーでコンパイルされた場合にのみサポートされます。

Data Monitoring は豊富な API を備える強力な機能です。これはプログラム・ファイル内の変数に関する詳細なシンボリックおよび属性の情報を取得することも可能です。しかしながら、本書では 1 つの変数の値を変更するとても簡素なプログラムを提供します。

Data Monitoring の詳細については *Data Monitoring Reference Manual* を参照して下さい。

set_rate プログラムのソース・コードは次のとおりです：

```
#include <stdlib.h>
#include <stdio.h>
#include <datamon.h>

#define check(x) ¥
if((x)) {fprintf(stderr, "%s¥n", dm_get_error_string());exit(1);}

main(int argc, char * argv[])
{
    program_descriptor_t pgm;
    object_descriptor_t obj;
    char buffer[100];

    if (argc != 2) {
        fprintf (stderr, "Usage: set_rate integer-value¥n");
        exit(1);
    }

    check(dm_open_program("app",0,&pgm));
    check(dm_get_descriptor("rate",0,pgm,&obj));
    check(dm_get_value(&obj,buffer,sizeof(buffer)));
    check(dm_set_value(&obj,argv[1]));

    printf ("rate: old_value=%s, new_value=%s¥n", buffer, argv[1]);
}
```

dm_open_program 関数は指定されたプロセスの名称と PID(0 の場合、指定した名前と一致するプロセスを使用するための呼び出しを指示しています)で Data Monitoring を初期化します。

dm_get_descriptor 関数は指定した変数名を探してその変数に関する情報を返します。また、これは **app** プロセスの変数の基礎となるメモリ・ページを監視プロセスにマッピングします。

dm_get_value と dm_set_value 関数はダイレクト・メモリの読み取りおよび書き込みを使って変数の値を戻すおよび設定します。**app** プロセスは rate 変数の値が変更されること以外に影響は及びません。

set_rate.c ソース・ファイルは 1-4 ページの「チュートリアル・ディレクトリの生成」の操作で現在の作業ディレクトリにコピーされています。

- 次のコマンドを使ってプログラムをコンパイルして下さい：

```
cc -g -o set_rate set_rate.c -ldatamon -lccur_rt
```

本チュートリアルのこの部分は少しも NightProbe 自体に依存しませんが、Datamon API を使って rate 変数の変更の影響を見るために NightProbe を使用します。

- NightProbe の Page 4 のラベルが付いたタブをクリックして Graph パネルを選択して下さい。
- グラフ・パネルの Pan Right ボタンを使って表示域をグラフの終端に移動して下さい(グラフの終端が見えるまで繰り返しボタンをクリックして下さい) :

- 次のコマンドを実行して **app** プロセスの rate 変数の値を変更して下さい :

```
./set_rate 123456789
```

前述のソース・コードが示すとおり、プログラムは変数の前の値を出力して、その後 **set_rate** への引数として指定された値に設定します。

正弦波と余弦波は Graph パネルで見られるように形が変わります。

終了 – NightProbe

NightProbe の操作を終了するには以下の手順を実行して下さい :

- File メニューから Exit Immediately オプションを選択して下さい。

これで NightStar RT チュートリアルの NightProbe の部は終了です。

6

NightTune の利用

NightTune はシステムの動作状況を解析および調整するためのグラフィカル・ツールです。

本章はプログラムが既にビルドされていて実行中であることを想定しています。プログラムをビルドしていない場合は 1-4 ページ「プログラムのビルド」の指示に従いビルドして、進める前にアプリケーションを実行して下さい：`./app &`

NightTune の起動

NightTune はコマンド・プロンプトで次のコマンドを使い起動することが可能です：

```
ntune &
```

または NightTune デスクトップ・アイコンをダブルクリックして起動することも可能です。

本チュートリアルのある状況に関しては、NightTune を **root** ユーザーとして実行する、またはユーザー・アカウントが適切な権限を持っていることを確保する必要があります。詳細については 1-2 ページの「ユーザー権限の設定」を参照して下さい。

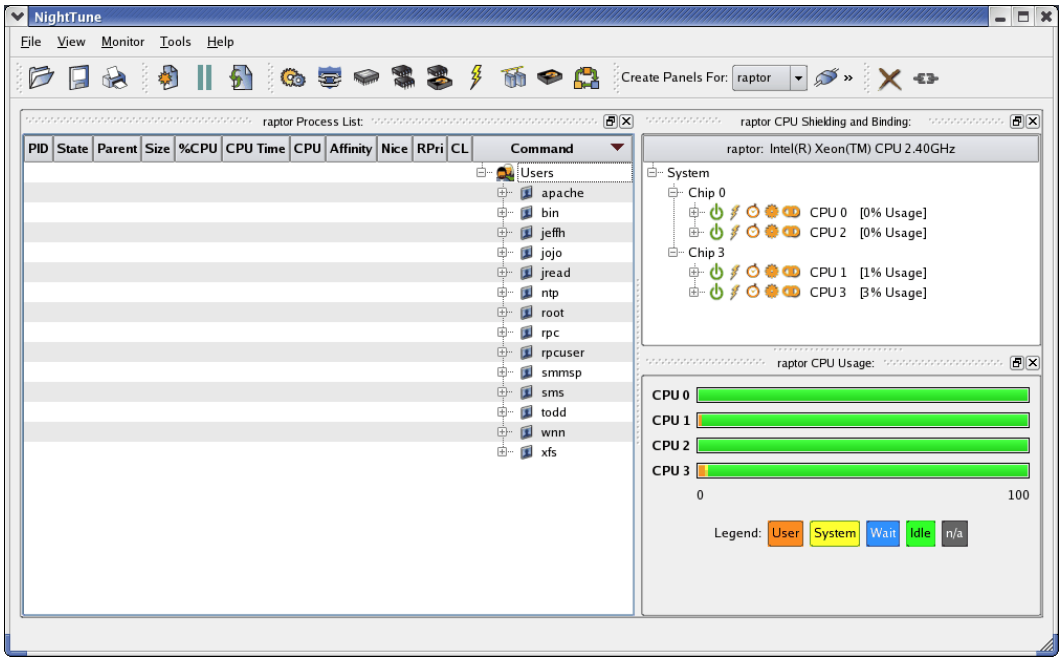


図 6-1. NightTune の初期パネル

NOTE

以前に NightTune を使用したことがある場合、NightTune 内の項目のレイアウトをカスタマイズしたかもしれません。本セッション中は本チュートリアルで使用されているような既定のレイアウトにリセットしたほうが良いかもしれません。その場合、File メニューから Load System Default Configuration を選択して下さい。

プロセスの監視

最初に実行中の **app** プロセスを監視します。

- Process List パネルにてパネル内のどこでもクリックしてから Ctrl-F を押下して下さい。
- Find バーがパネル下部に現れます。**^app** と入力するとプロセス・リストが自動的に展開され単語 app で始まるプロセス名称の最初のプロセスが選択されます。

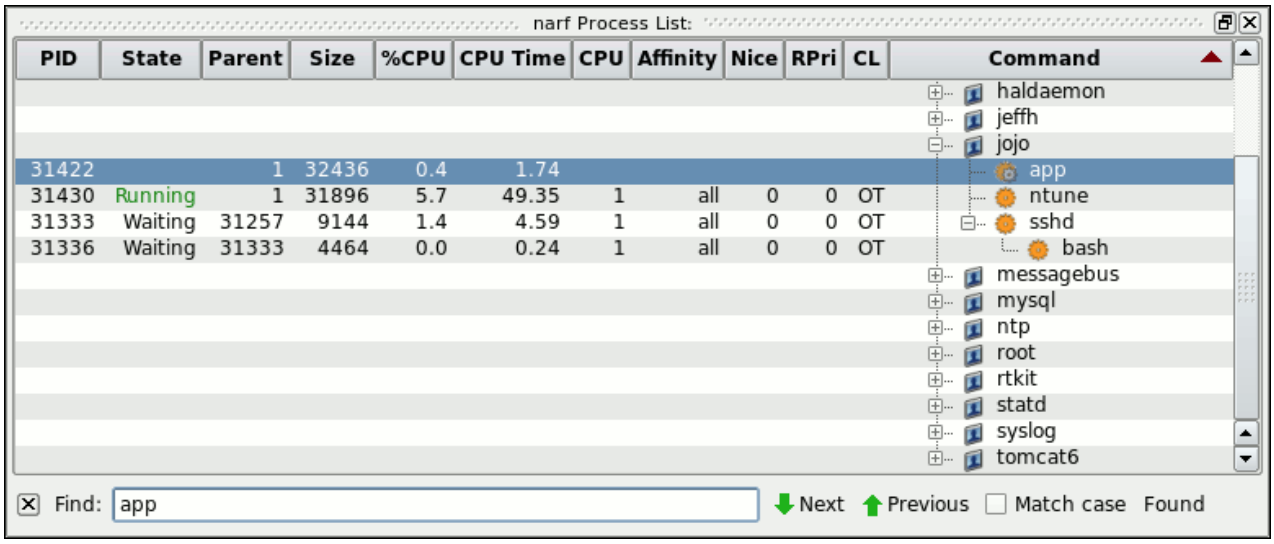


図 6-2. 展開された Process List

選択されたプロセスが **app** プロセスではない場合、正しいプロセスが選択されるまで Find バーの Next アイコンをクリックして下さい。

app プロセスに関連付けられたアイコンは橙色のプロセス・アイコンに小さな灰色の歯車が重なっていることに注目して下さい。これはプロセスがマルチスレッドであることを示しています。

- **app** プロセスに関連付けられたコンテキスト・メニューから Show Threads オプションを選択して下さい。

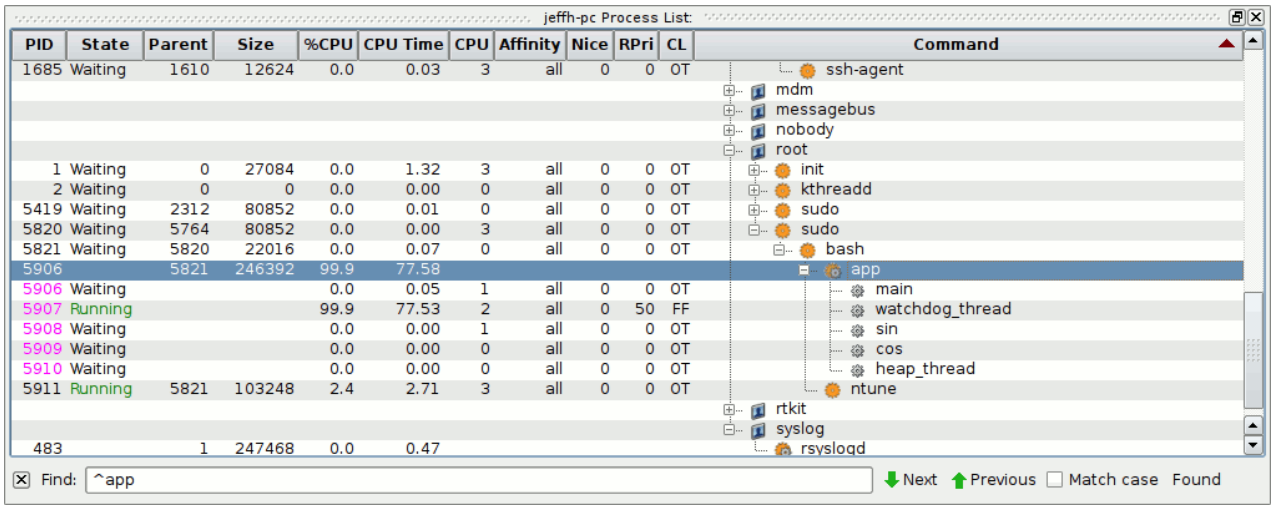


図 6-3. スレッドを含む Process List

パネルは各スレッドおよびプロセス全体の指標を示します。具体的には次を含みます：

- プロセスの仮想メモリサイズ
- 各スレッドおよびプロセス全体で使用する CPU 時間の割合および総量
- 直近で各スレッドが実行された CPU
- 各スレッドの CPU アフィニティ(スレッドが実行を許可されている CPU のセット)
- 各スレッドのスケジューリング特性
- NightView でデバッグ中である場合、またはアプリケーションが NightTrace API を使用中で **trace_set_thread_name(3x)** の呼び出しを介してスレッドに名前を付けている場合はスレッド名称

表示されている列のセットは、パネルのコンテキスト・メニューの Display Fields オプションをクリックした後、メニュー項目をチェックまたはチェックを外して個々の領域を選択することで変更することが可能です。

システム・コールのトレース

NightTune はプロセスで使用されるシステム・コールをトレースする便利なインターフェースを提供します。これは NightTune が検索および管理することが可能なダイアログへの出力を提供することを除いては基本的に strace(1)コマンドの使用と同じです。

- **app** プログラムの sin スレッドに関連付けられたコンテキスト・メニューから Trace System Calls...オプションを選択して青緑色の右矢印の開始ボタンを押して下さい。

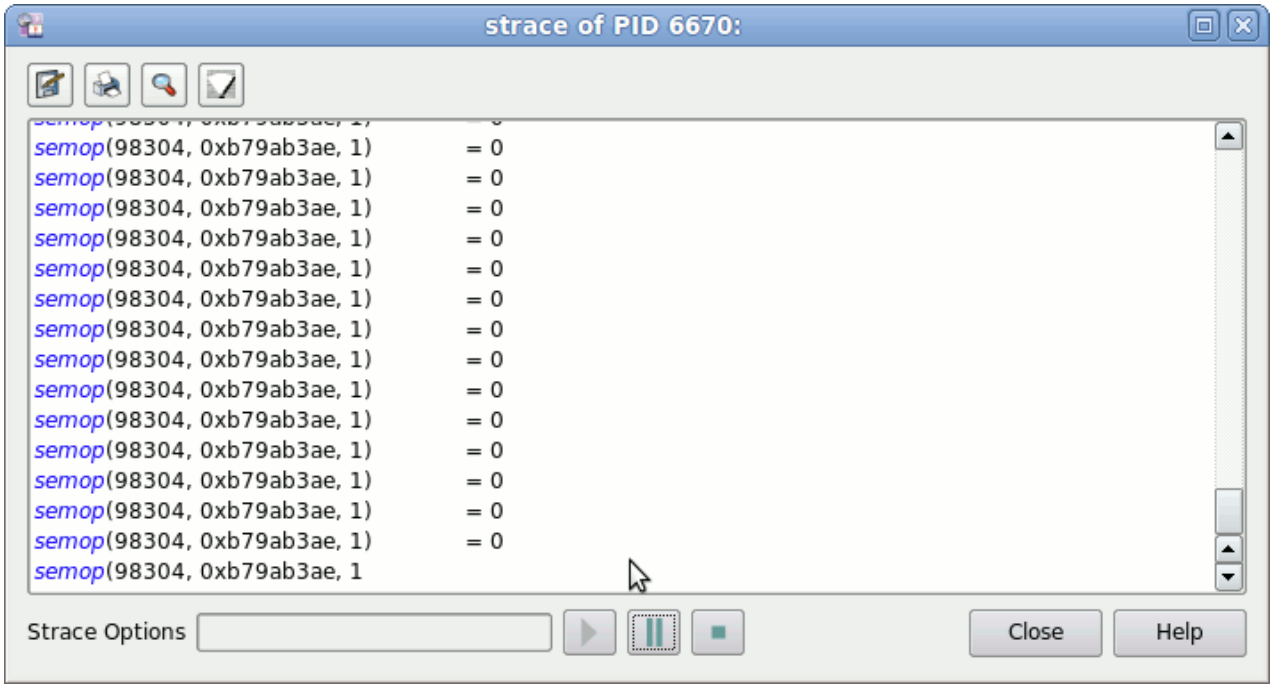


図 6-4. スレッドの strace 出力

上図で示されるように選択されたスレッドは以下のコード・セグメント内で示す通り **api.c** の 51 行目に関わる **semop(2)** 以外のシステム・コールは呼びません：

```
41 void *
42 sine_thread (void * ptr)
43 {
44          control_t * data = (control_t *)ptr;
45          struct sembuf wait = {0, -1, 0};
46          work(1);
47
48          trace_set_thread_name (data->name);
49
50          for (;;) {
51                    semop(sema, &wait, 1);
52                    data->count++;
53                    data->angle += data->delta;
54                    data->value = sin(data->angle);
55          }
56 }
```

- システム・コール・トレースを終了してダイアログを閉じるには Close ボタンを押して下さい。

プロセス詳細

NightTune はプロセス属性の詳細な分析を提供します。

- **app** プログラム内の任意のスレッドのコンテキスト・メニューから Process Details...オプションを選択して下さい。

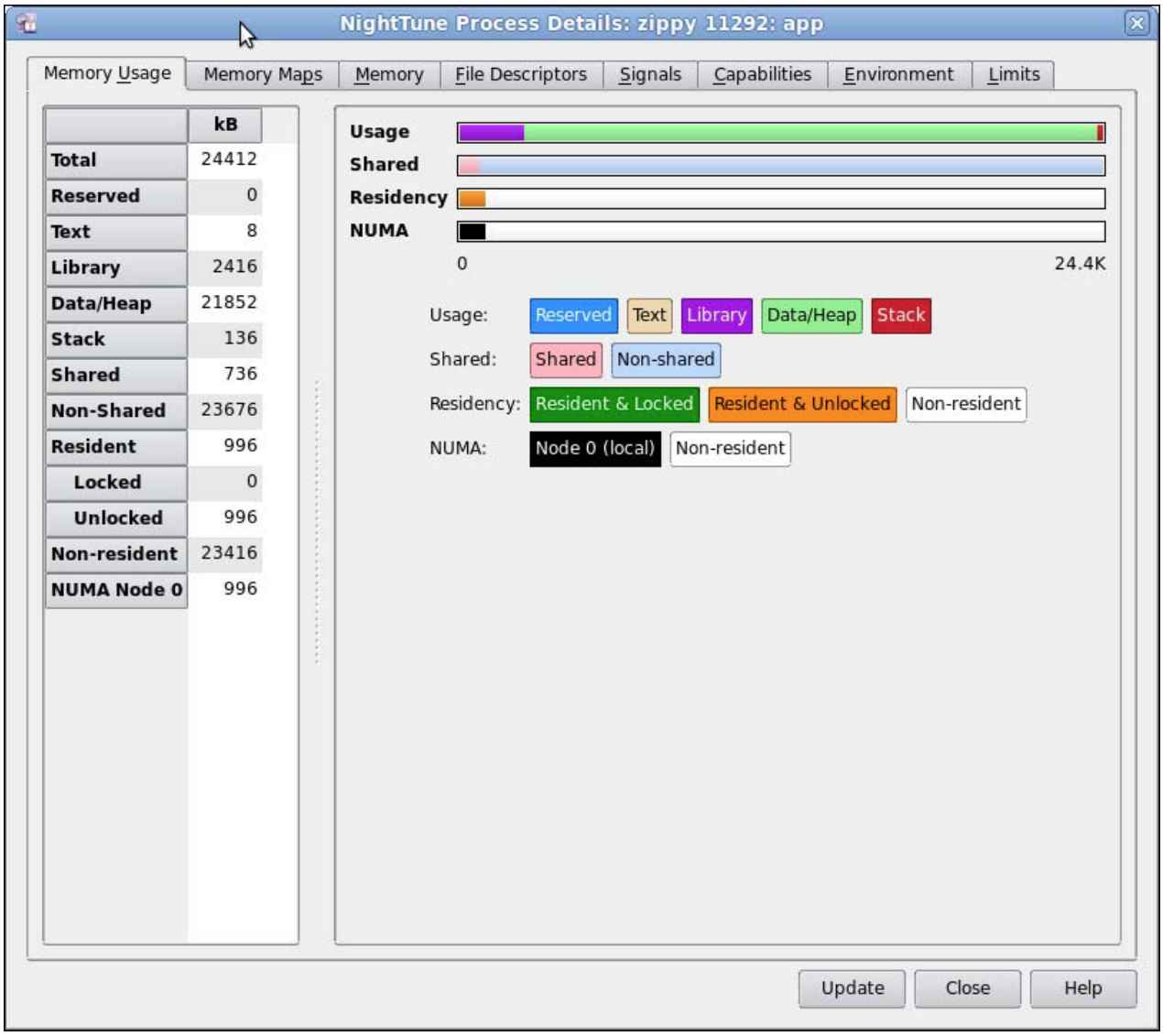


図 6-5. Process Details ダイアログ

本ダイアログに表示された全ての情報は本質的に読み取り専用です。Memory タブ上のメモリ・ページのロックを除き、本ダイアログを使ってプロセスの属性を変更することは出来ません。

7 つのタブ・ページは以下を含むプロセスに関する詳細な情報を提供します：

- Memory Usage
- Memory Maps
- Memory Details
- File Descriptors
- Signals
- Capabilities

- Environment
- Limits

Memory Usage ページは仮想および常駐メモリの使用状況のサマリー情報をテキストとグラフィックの枠の両方で提供します。

プロセス詳細 - Memory Details

- ページを表示するには Memory タブをクリックして下さい。

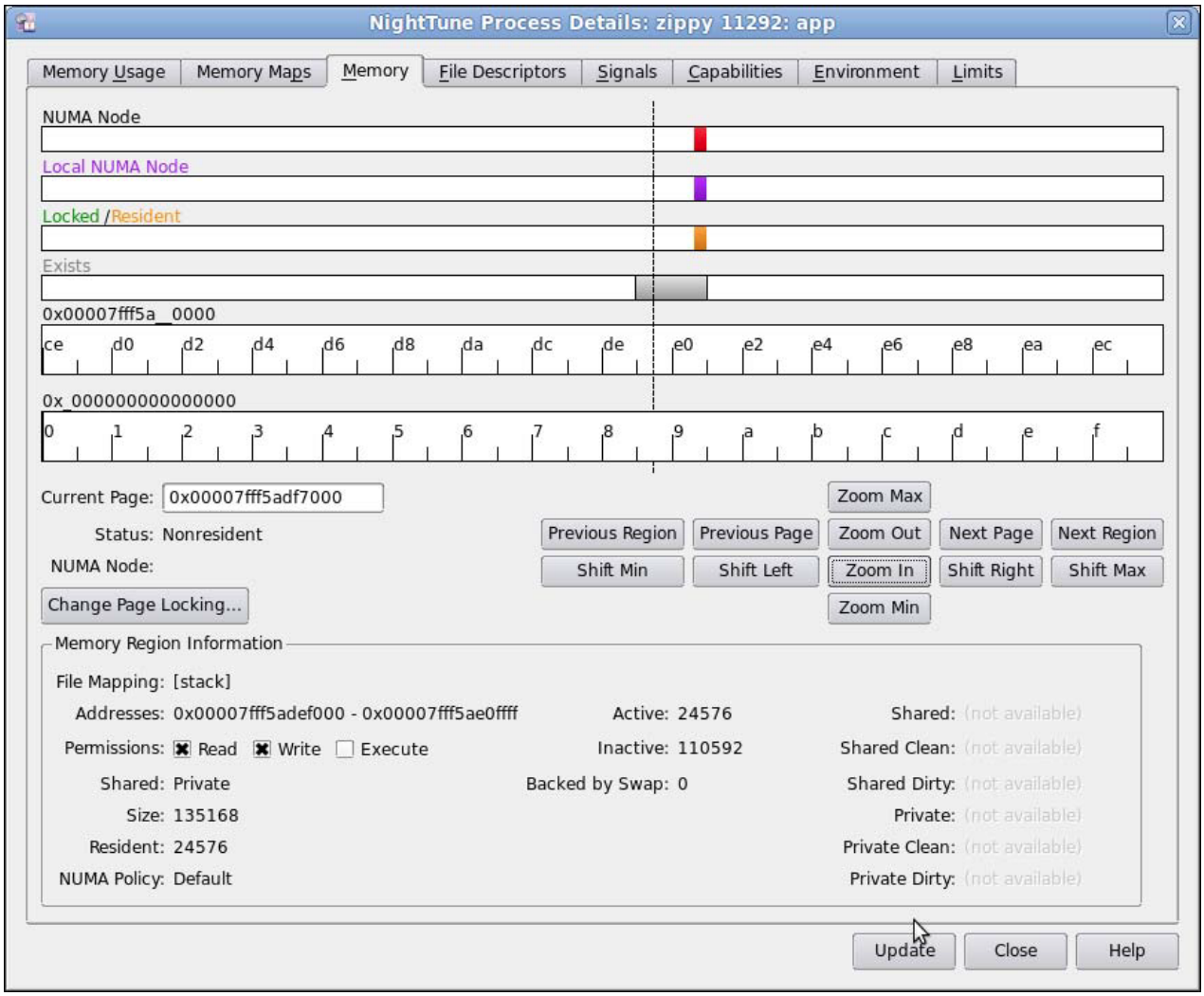


図 6-6. プロセス・メモリ詳細ページ

本ダイアログはアドレス空間内の任意のセグメントまたはページに関する詳細なメモリ情報が取得可能なコントロールを提供します。

グラフィック部分のコントロールは本質的に NightTrace に似ています。

- 任意の場所またはルーラー上をクリックして下さい。
- 完全にズームアウトするには Alt+上矢印を押下して下さい

プロセスのアドレス空間全体が現在表示されています。そのプロセス内のページに関連するメモリ・アドレス空間の各セグメントは Exists 行に少なくとも 1 本の垂直の黒い線で示されます。

- 線が表示の中央部分に表示されていない場合、Previous Region をクリックして何かが見えるまで複数回 Zoom In をクリックして下さい。
- 十分な詳細が得られるまでマウス・ホイールまたは Zoom In ボタンを使ってズームインして下さい。

上図では、メモリ・セグメントは Exists 行に灰色の領域で表示されています。メモリ・セグメントの境界は垂直の黒い線で表示されています。ズーム係数が十分大きい場合、メモリ・セグメントは 1 本または 2 本だけの垂直の黒い線で描かれます。

メモリ・セグメントに関する詳細はページ下部にあるテキスト領域に表示されます。

他の行は NUMA プールやページの Locked/Resident 属性を含むページ単位の情報を表示します。

NOTE

Locked/Resident 情報は全てのオペレーティング・システムのバージョンで利用可能というわけではありません。NUMA 情報は Non-Uniform Memory Architecture をサポートするシステムだけに適用され、その情報は一部のオペレーティング・システムにのみ提供されます。

あるいは、Current Page テキスト領域に入力することで特定のアドレスを選択することが可能です。

Memory ページに関する詳細については NightTune User's Guide を参照して下さい。

プロセス詳細 - File Descriptors

File Descriptors はそのプロセスに関連する全てのオープン済みファイル記述子を記載して、それぞれの説明を提供します。

下図は **ntune** プロセスで使用中のファイル記述子を示しています。

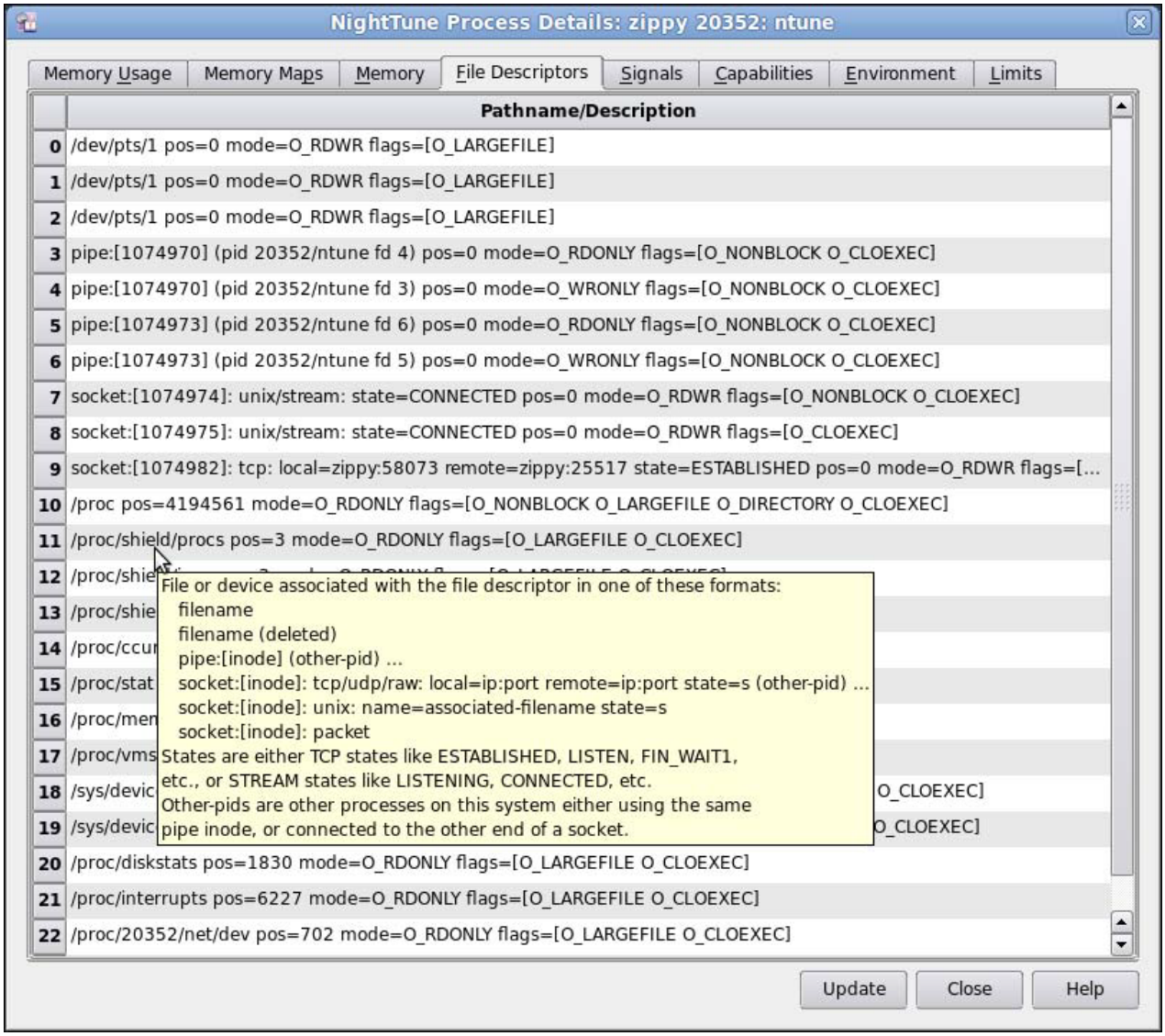


図 6-7. File Descriptors ページ

説明は(関連がある場合)ファイル記述子に関連するファイル名称やソケットに関する接続情報を含んでおり、パイプやソケットを使用している他のプロセスが同じシステム上にある場合はそれらを特定します。

プロセス詳細 – Signals

Signals タブはシグナルの属性を表示します。

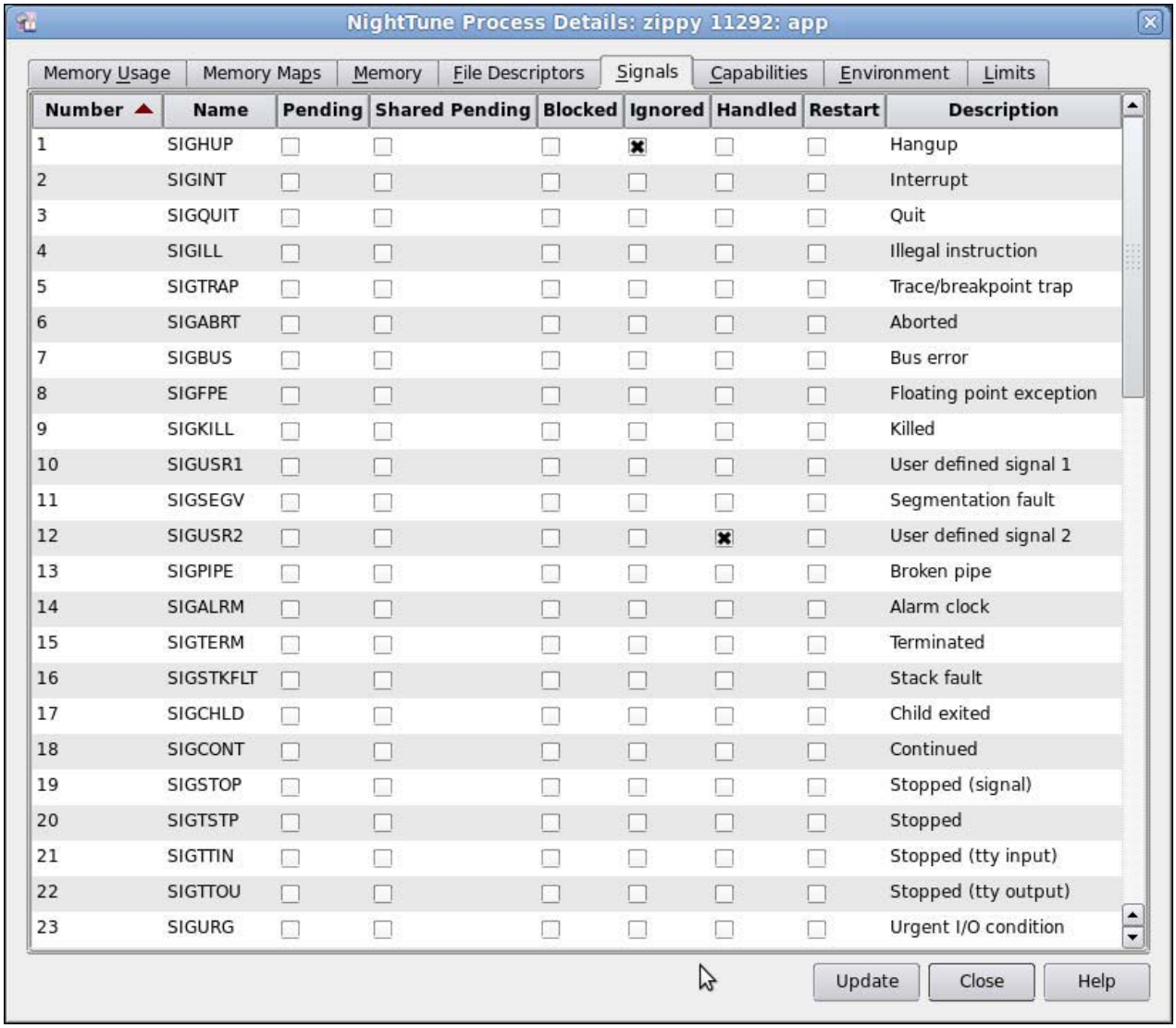


図 66-8. Signals ページ

表示される情報はアプリケーションによって現在保留またはブロックされているシグナルのインジケータ、およびアプリケーションがシグナル用にインストールされたハンドラーを持っているかどうかを含みます。

上図では、アプリケーションは SIGUSR2 用に登録されたハンドラーを所持しており、SIGHUP を無視しています。

- ダイアログを閉じるには Close ボタンをクリックして下さい。

プロセスのスケジューリング・パラメータを変更

どのようにアプリケーションの挙動が変わるかを見るには、動作中にスレッドまたはプロセスのスケジューリング特性を変更することが望ましい場合があります。例えば、あるスレッドが他のスレッドにより CPU 時間が不足しているとします。その場合、そのスケジューリング・クラスをリアルタイム・クラスへ、優先度をより高い優先度へ変更することを推奨します。

- **app** プロセスの sin スレッドに紐づくコンテキスト・メニューの Process Scheduler...オプションを選択して下さい。

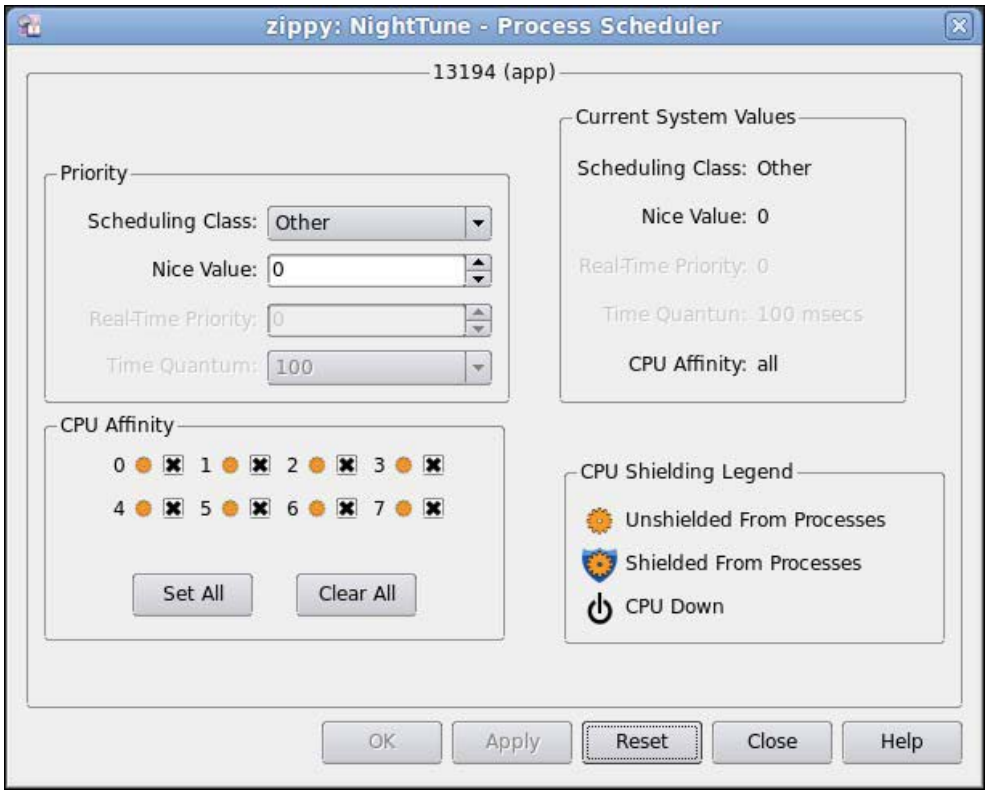


図 6-9. Process Scheduler ダイアログ

このダイアログでは、Scheduling Class, Nice Value, Real-time Priority, Time Quantum を変更することが可能です。マルチ・プロセッサー・システムでは、CPU Affinity も同様に変更することが可能です。プロセスまたはスレッドが実行を許可されている CPU については、CPU 番号のチェックボックスがチェックされているはずです。本トピックに関する詳細は 6-11 ページの「プロセスの CPU アフィニティを設定」を参照して下さい。

NOTE

Scheduling Class を Round Robin に変更して Real-time Priority を変更するには、NightTune を **root** ユーザーまたは 1-2 ページの「ユーザー権限の設定」に記述されているように適切な権限を持つユーザー・アカウントで実行されている必要があります。

- Scheduling Class をドロップダウン・リストから選択して Round Robin に変更して下さい。

- Real-time Priority を 3 に変更して下さい。
- OK ボタンを押して下さい。

Process List パネルはすぐにスレッドへのこれらの変更を反映します。

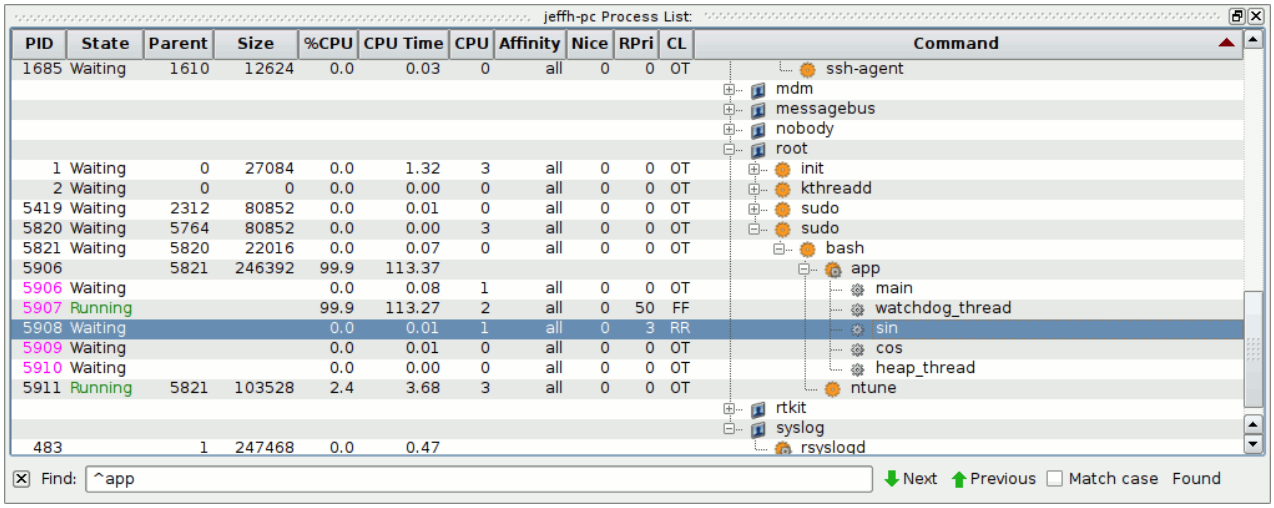


図 6-10. 変更されたスレッドを含む NightTune の Process List

変更されたスレッドについては、CL(Scheduling Class)領域は RR(Round Robin)が表示され、RPri(Real-time Priority)領域は 3 が表示されます。

プロセスの CPU アフィニティを設定

本項だけは NightTune が動作中のシステムがマルチプロセッサー・システムの場合に適用されます。そうではない場合、6-17 ページの「終了 - NightTune」に進んで下さい。

CPU Shielding and Binding パネルは CPU 階層、シールド状況(Concurrent RedHawk Linux のみ)、CPU 使用率、バインドしているプロセスと IRQ を表示します。

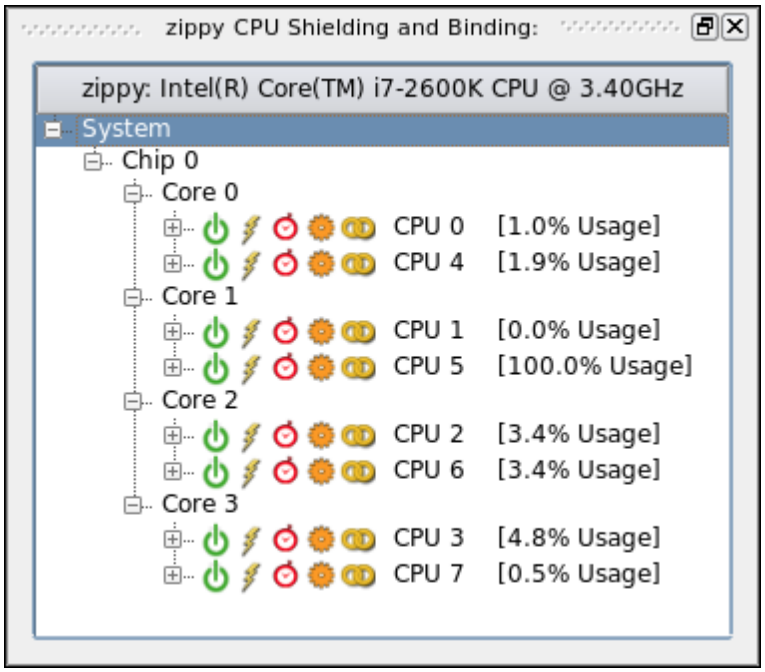


図 6-11. CPU Shielding and Binding パネル

この階層は論理 CPU の関係性、特にハイパースレッドとマルチコア・チップの存在を視覚化するので便利です。

上図では、1 つのチップにハイパースレッド化された 4 つの物理コアで合計 8 個の論理 CPU が含まれています。ハイパースレッド化された CPU はその間で一部の物理リソースを共有しますが、独立したプロセッサーとして全てユーザーが見れる方法で動作します。マルチコア CPU もシブリング間で物理リソースを共有していますが、ハイパースレッド技術と比較して非常に少なくなっています。

プロセスまたはスレッドは CPU アフィニティを持っており、それは実行する可能性のある CPU セットを決定します。1 つの CPU 上で実行するように制限されることもあります。大抵これはプロセスまたはスレッドの*バインド*と呼ばれます。6-10 ページの「プロセスのスケジューリング・パラメータを変更」で CPU アフィニティを変更する 1 つの方法を説明しました。更に CPU Status パネルはプロセスまたはスレッドを直ぐにバインドするために使用することが可能です。

- パネル内の System 項目に関連するコンテキスト・メニューから Expand All を選択して下さい。

ツリーは各 CPU にバインドされたプロセスや割り込みに関するツリーを展開します。

- カーソルが **app** プロセスのスレッドの 1 つの上にある間に*左*マウス・ボタンを押したままにして、次に CPU Shielding and Binding パネル内の CPU の 1 つにスレッドをドラッグしてマウスボタンを離して下さい。

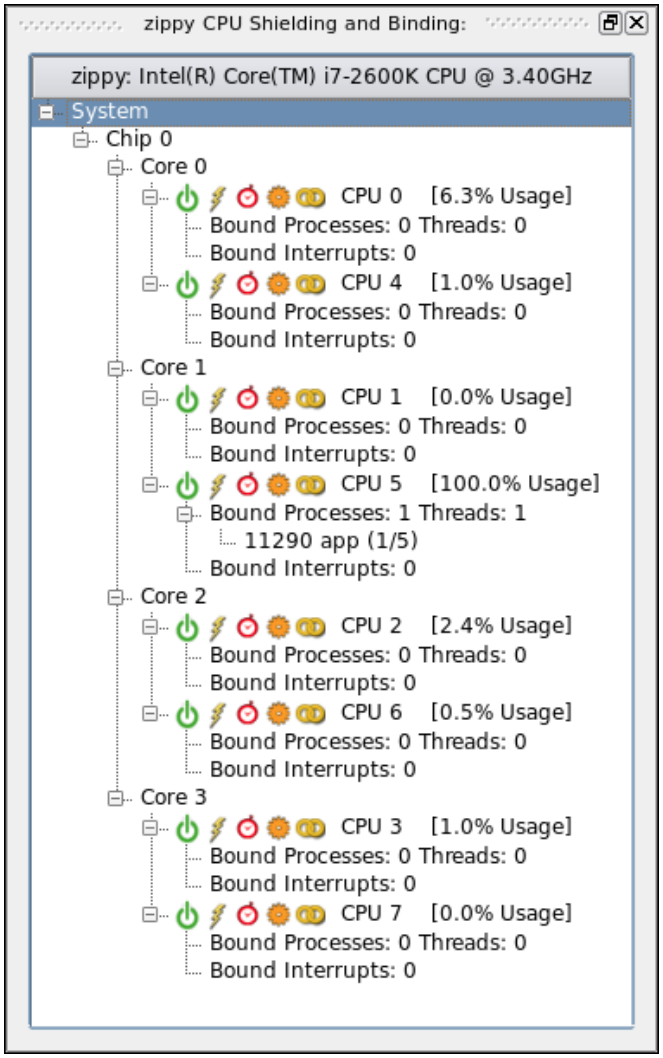


図 6-12. バインドされたスレッドを含む CPU Shielding and Binding パネル

この動作は特定の CPU に対し選択したスレッドをバインドします。つまり、その CPU アフィニティは 1 つの CPU だけを含むように設定されます。プロセスまたはスレッドの CPU アフィニティが 1 つの CPU のみ含んでいる場合、そのプロセスまたはスレッドは CPU Shielding and Binding パネル内の特定の CPU のプロセス・ツリーの下に表示されます。上図では、sin スレッドを CPU にバインドしたので CPU 5 の下に 1 つのエントリがあります。この例では 1 つのスレッドだけが CPU 5 にバインドされているので、そのエントリはサフィックス(1/5)を含んでおり、5 つのスレッドのうち 1 つだけがその CPU にバインドされていることを示しています。

NOTE

お手持ちのシステムでは選択した CPU に追加のプロセスや割り込みがバインドされているかもしれません。

スレッドの新しい CPU アフィニティも Process Monitor パネルの Affinity 領域に反映されます。

この領域は 16 進数のビット・マスクで表示され、その最下位ビットは CPU 0 を表し、その隣は CPU 1, CPU 2,･･･を表します。今回のケースで値 0x20 は CPU 5 を示しています。

NightTune は直ぐにプロセスをアンバインドすることも可能です。

- カーソルがパネルのスレッド・エントリ上にある間に*左*マウス・ボタンを押したままにして、次にウィンドウの右上にある(壊れた鎖に似ている)Unbind アイコンにアイテムをドラッグしてマウスボタンを離して下さい。

Process List パネルはスレッドが再度アンバインドされたことを反映します。

NightTune の中からプログラムを終了することも可能です。

- Process List パネルで **app** プログラムをクリックし、キル・ボタン(赤い X)上に来るまでドラッグしてマウスを離して下さい。

割り込みの CPU アフィニティを設定

本項で説明する機能だけは NightTune が **root** ユーザーまたは 1-2 ページの「ユーザー権限の設定」に記述されているように適切な権限を持つユーザー・アカウントで実行されている場合に利用可能です。そうではない場合、6-17 ページの「終了 - NightTune」に進んで下さい。

プロセスの CPU アフィニティを設定できることに加えて、NightTune は割り込みの CPU アフィニティを制御することも可能です。

割り込みの CPU アフィニティを変更することが望ましい場合があります。例えば、割り込みが頻繁に発生している場合、それを処理する CPU によっては同じ CPU 上で実行中のアプリケーションを邪魔している可能性があります。

- タイトル・バーの右端のボックスをクリックして Process List パネルを閉じて下さい。
- 代わりに Monitor メニューから Interrupt Detail Activity オプション、続いてそのサブメニューから Text Pane オプションを選択して Interrupt Detail Activity パネルを開いて下さい。

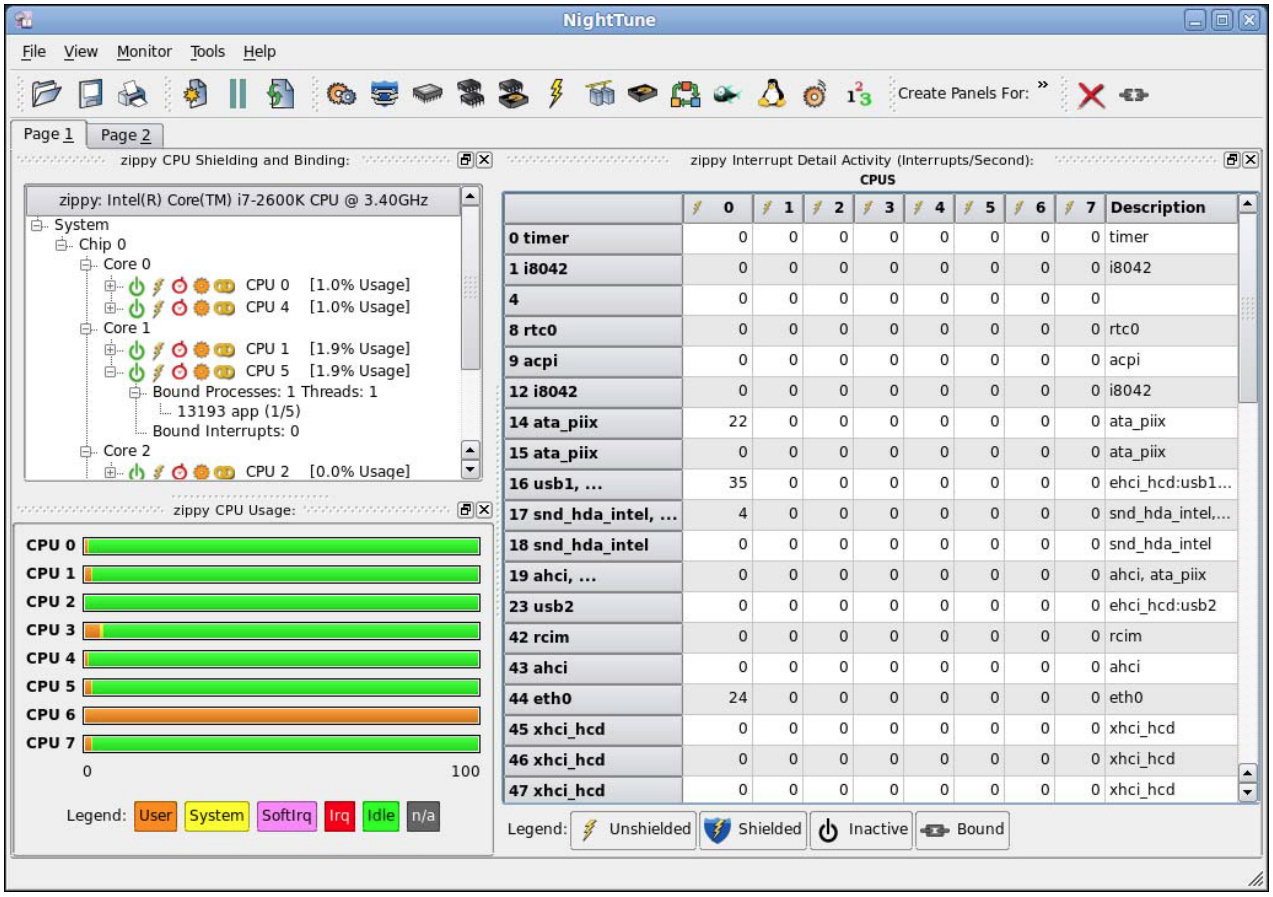


図 6-13. Interrupt Detail Activity パネルを含む NightTune

パネルは(マルチプロセッサー・システムの場合)各 CPU で処理された各割り込みについて 1 秒毎の割り込み数を表示します。

Interrupt Detail Activity パネルの鎖アイコンは割り込みが特定の CPU で処理されたことを示しています。一方、割り込みが全ての CPU で処理される場合はその割り込みに対してアイコンは現れません。同じ情報が CPU Shielding and Binding パネル内の各 CPU の Bound Interrupts items に表示されます。

一部のシステムでは IRQ アフィニティを時間と共に自動的に変更する IRQ バランシングを使用している可能性があります。これは割り込みアフィニティを手動で制御する試みに干渉します。本チュートリアルの目的のため、root ユーザーで次のコマンドを実行して IRQ バランシングが現在無効になっていることを確実にして下さい：

```
/sbin/service irqbalance stop
```

1 つの CPU に割り込みをバインドするには、プロセスと同じような方法でドラッグすることが可能です。

カーソルが Interrupt Detail Activity パネルの割り込みの上にある間、割り込み行の(タイトル以外の)任意のデータ・セル上を左マウス・ボタンを押したままにして、次に CPU Shielding and Binding パネルの特定の CPU に割り込みをドラッグして下さい。同様にカーソルが CPU Shielding and Binding パネル内にある CPU の Bound Interrupts リストの割り込みの上にある

間、左マウス・ボタンを押したままにして、次に CPU Shielding and Binding パネルの異なる CPU に割り込みをドラッグすることが可能です。

割り込みのアフィニティを複数の CPU または 1 つ以上を除いて割り当てるように変更するには、パネル内のいずれの割り込み行のコンテキスト・メニューから Set CPU Affinity...オプションを選択して下さい。

NOTE

root ユーザーして実行していないまたは適切な権限が不足するユーザーの場合、Set CPU Affinity...オプションはコンテキスト・メニューに表示されません。

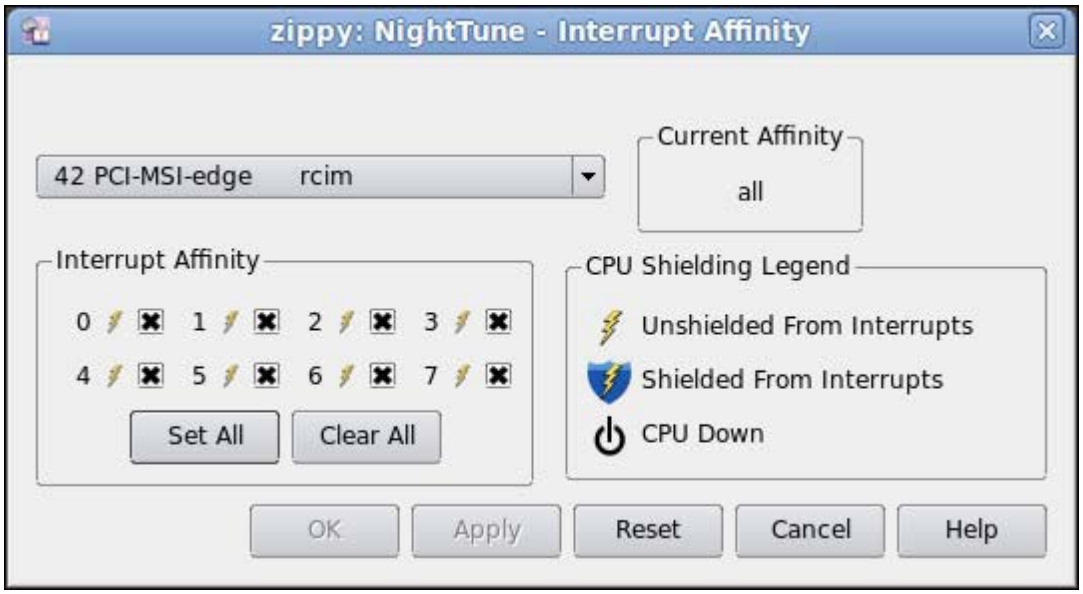


図 6-14. Interrupt Affinity ダイアログ

割り込みを処理することを許可する各 CPU に対して、その CPU 番号のチェックボックスをチェックする必要があります。変更は OK もしくは Apply ボタンを押すと有効になります。

NOTE

NMI のような一部の割り込みについては、その CPU アフィニティを制御することは不可能です。

最大のデターミニズムと性能のために CPU をシールド

NightTune はプロセス、割り込み、他の CPU による共有リソースの干渉から特定の CPU を簡単にシールドすることが可能です。

これについては本章の NightSim の項の一部として実演しています。詳細については 7-9 ページの「オーバーラン検知とシステム・チューニング」を参照して下さい。

終了 – NightTune

チュートリアルの残りの部分は **app** プログラムの実行とは関係がありません。以下の手順を実行してプログラムを終了して下さい：

- 左マウスボタンを使い Process List パネルから **app** プロセスをツールバー上の Kill アイコンにドラッグして離して下さい。

- File メニューから Exit を選択して NightTune を終了して下さい。

これで NightStar RT チュートリアルの NightTune の部は終了です。

7

NightSim の利用

NightSim は複数のプロセスを同期させる方法でスケジューリングしその実行を監視するためのグラフィカル・ツールです。

NightSim は Frequency Based Scheduler ユーティリティのグラフィカル・インタフェースを提供します。

お手持ちのシステムに Frequency Based Scheduler がインストールされていない場合、チュートリアルのこの章は適用でません。次のコマンドを使って Frequency Based Scheduler がインストールされているかどうかを確認して下さい：

```
rpm -q ccur-fbsched
```

チュートリアルの本章では、殆どの Concurrent iHawk システムで標準装備されている Real-Time Clock and Interrupt Module (RCIM)からのリアルタイム・クロック割り込みソースも使用します。お手持ちのシステムに RCIM が含まれていない場合、チュートリアルのこの章は適用でません。次のコマンドを使って RCIM がインストールされているかどうかを確認して下さい：

```
cat /proc/driver/rcim/status
```

上述のファイルが存在しない場合は RCIM がお手持ちのシステムにはない、またはご使用のカーネルは RCIM サポートが削除されています。

本項の一部の状況については、root ユーザーとして NightSim および NightTune を実行、またはユーザー・アカウントが適切な権限を持っていることを確認する必要があります。詳細については 1-2 ページの「ユーザー権限の設定」を参照して下さい。

FBS アプリケーションの生成

NightSim を介してスケジュールされる周期アプリケーションを修正するのは些細なことです。

1 つの API コールが必要となります。

単純化した **work** アプリケーションのソース・コードは次のようになります：

```
#include <fbsched.h>
int workload = 1000;
main()
{
    int data = 0;
    int i;
    volatile double d = 1.0;
    while (fbswait()>=0) {
        data = !data;
        for (i=0; i<workload; ++i) d = d/d;
    }
}
```

fbswait()の呼び出しは次にスケジュールされたサイクルまでそれが戻るポイントでプロセスをブロックします。その後プロセスはそのワークロードを実行して次にスケジュールされたサイクルまで fbswait()でブロックすることを繰り返します。

work.c ファイルは本チュートリアルの早い段階で現在の作業ディレクトリに **/usr/lib/NightStar/tutorial** からコピーされています。

次のコマンドを使ってアプリケーションをコンパイルおよびリンクして下さい。

```
cc -g -o work work.c -lccur_fbsched -lccur_rt
```

NightSim の起動

NightSim の設定ファイルは本チュートリアルのために用意されており、ページ 1-4 の「チュートリアル・ディレクトリの生成」項の作業中に現在の作業ディレクトリにコピーされているはずです。

以下に示すように設定ファイルを指定して NightSim を開始して下さい：

```
nsim -c nsim.nsc &
```

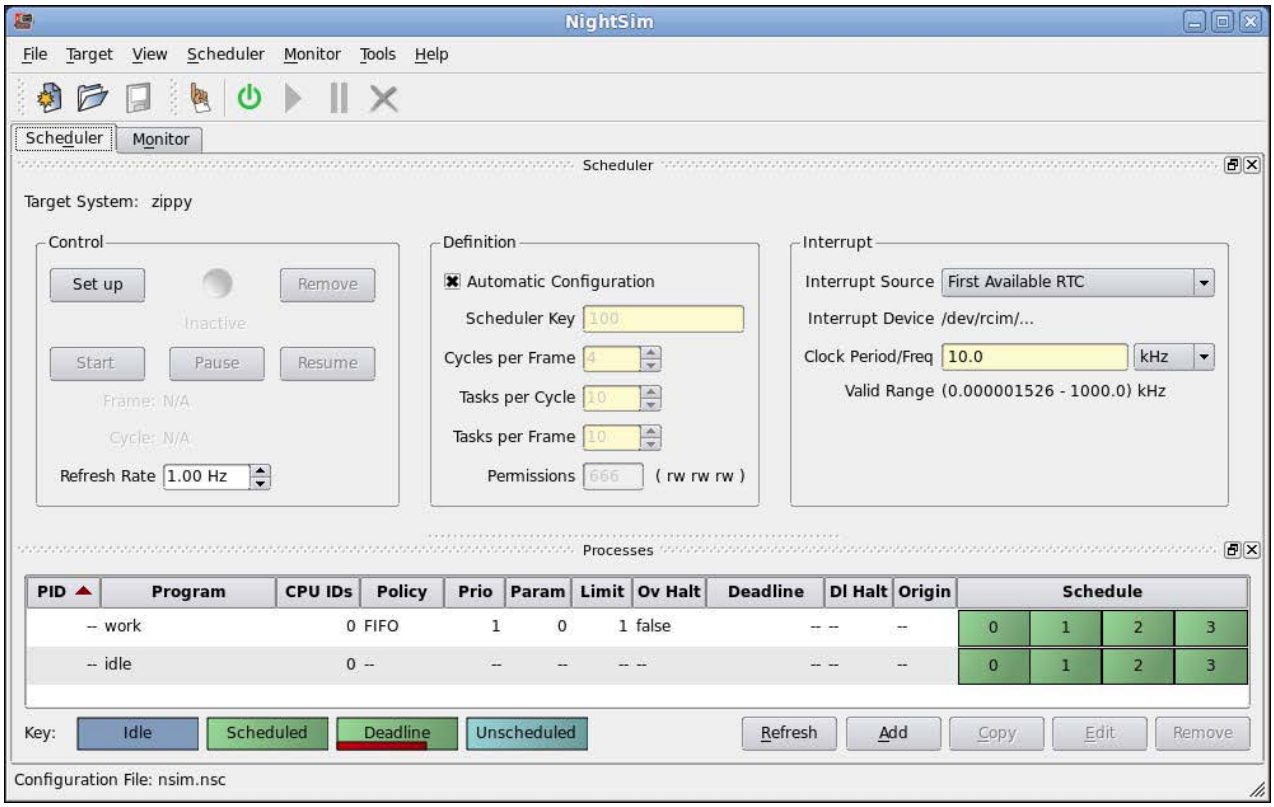


図 7-1. NightSim の初期ウィンドウ

NOTE

NightSim がターゲット・システムに接続できないエラーを提示する場合、お手持ちのシステムの **hostname** が**/etc/hosts** ファイルに含まれており適切な IP アドレスであることを確認して下さい。

スケジューラの生成

NightSim は以下のパラメータを使って複数のプロセスのスケジューリングを定義することが可能です：

- スケジューリング・ソース(通常は外部割込み)
- (クロック・ベースの割り込みについては)割り込みを発生させる割合
- プロセスがスケジュールされる間隔
- CPU アフィニティ、スケジューリング・ポリシー、スケジュールされたプロセスの優先度

共にこれらのパラメータは*スケジューラ*を定義します。

サイクルはスケジューリング・ソース(割り込み)間の時間として定義されます。

フレームは固定サイクル数で定義されます。フレームは、反復操作全体(フレーム)を繰り返す前に一連の別々の間隔(サイクル)を順番に実行する必要のある多くの周期的アプリケーションにおいて便利な概念です。

前項のコマンド・ラインで指定した **nsim.nsc** ファイルで構成されたスケジューラは、メイン・ウィンドウ上に見える以下の属性を持つスケジューラを定義しました：

- Cycles Per Frame -- フレームあたり 4 サイクル
- Timing Source -- Real-Time Clock and Interrupt Module (RCIM)の First Available RTC (リアルタイム・クロック)を使用する割り込みソース
- Clock Period -- 100 マイクロ秒のサイクル時間(10.0kHz)
- Processes -- 1 つのプロセス(**work**)をフレームの各サイクルで実行するスケジュール

スケジュールされたプロセスの詳細な属性を表示するには、Processes パネルの下部にあるプロセス領域の**./work** プロセスを選択してパネルの右下にある Edit...ボタンを押して下さい。

Add Process ダイアログが表示されます

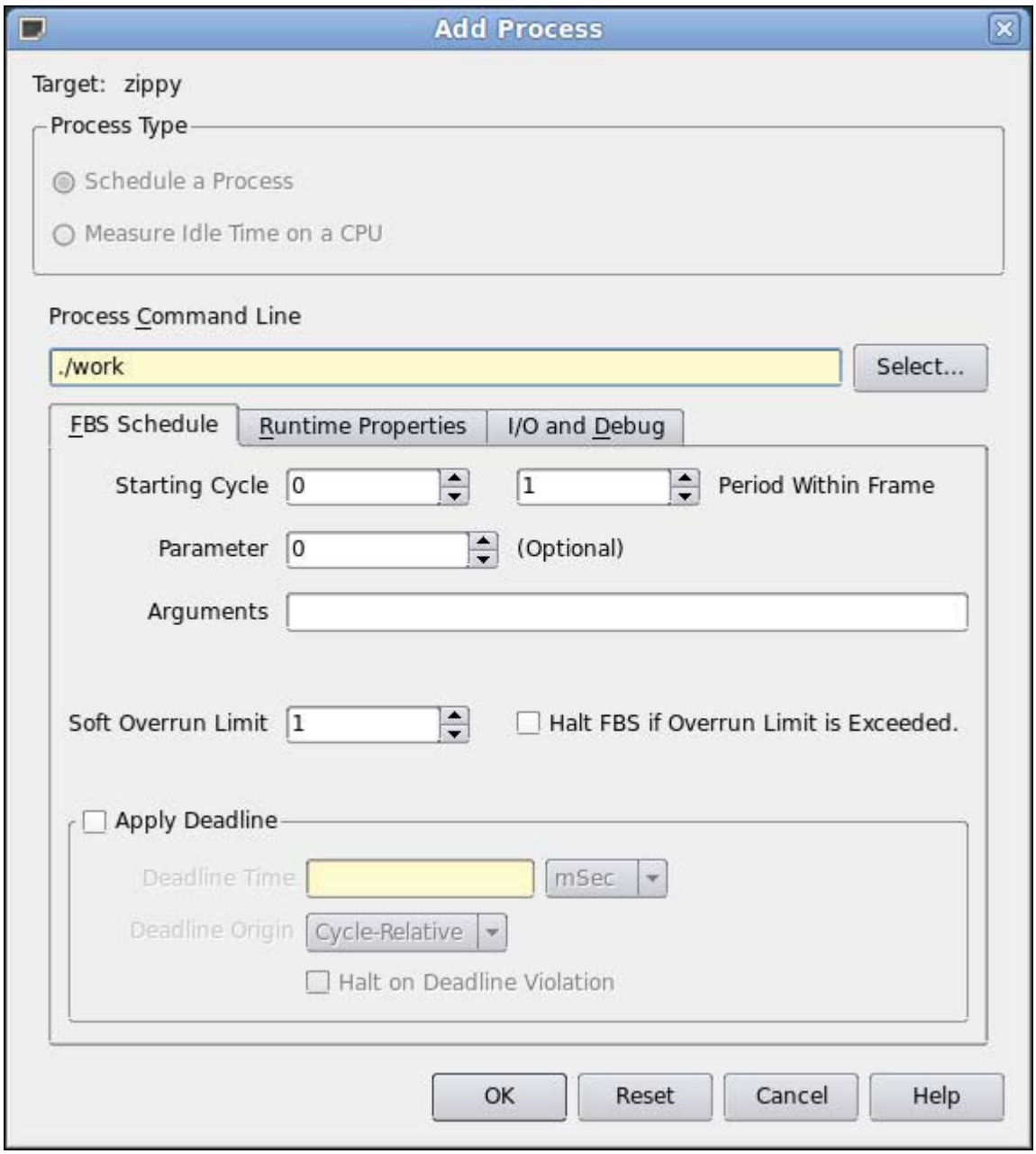


図 7-2. NightSim の Add Process ダイアログ

FBS Schedule タブは **work** プロセスの開始サイクルと間隔を表示します。Starting Cycle はプロセスが実行を開始するフレームのサイクルを定義します。Period は実行の頻度をサイクルで定義します。Period の値が 1 の場合はフレーム内の各サイクルでアプリケーションを実行します。

ダイアログの Runtime Properties タブをクリックして下さい。

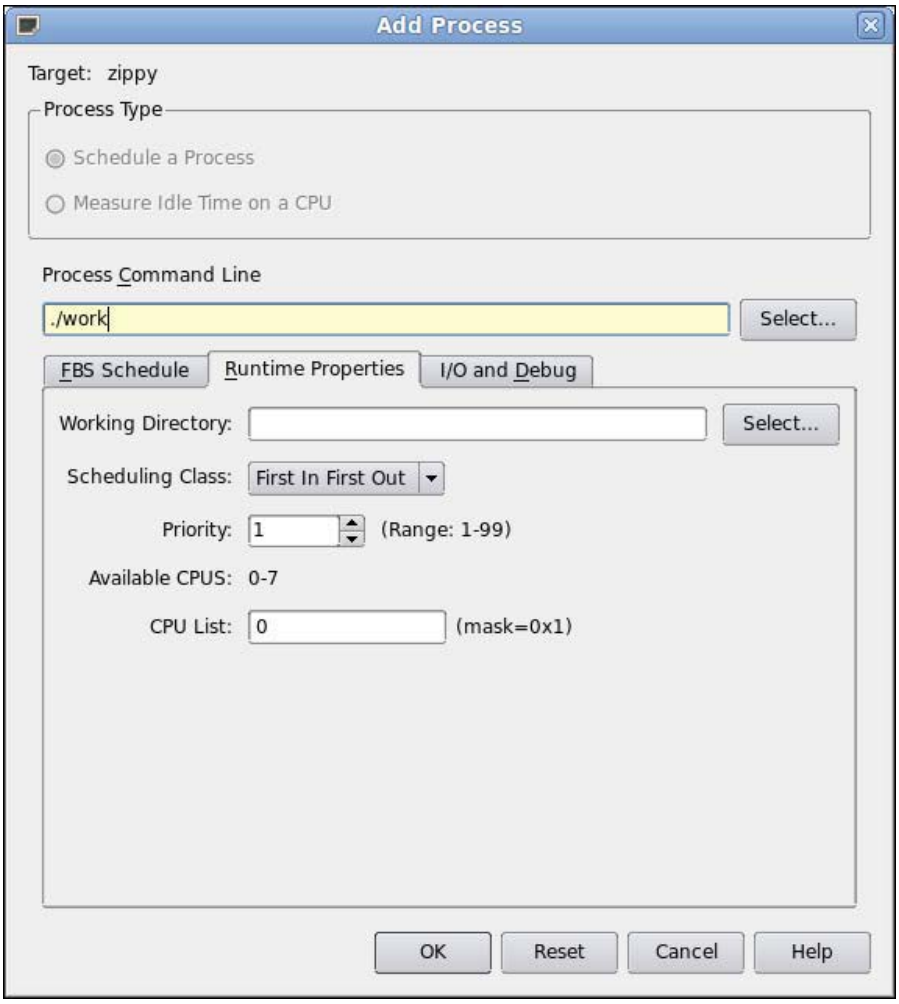


図 7-3. Runtime Properties タブ

NOTE

Runtime Properties タブ内の Available CPU 説明領域はお手持ちのシステムの CPU 数によって変わる可能性があります。

タブは実行が許可されている CPU、プロセスのスケジューリング・ポリシーやスケジューリング優先度を選択することが可能です。

Cancel ボタンを押してウィンドウを閉じて下さい。

work プロセスに加え、NightSim ウィンドウのスケジューリング領域に **idle** プロセスが表示されていることに注意して下さい。各サイクルで使用可能なアイドル時間の量を後で監視するために **idle** プロセスを登録しています。**idle** プロセスはスケジュールされたプロセスではなく、むしろアイドルサイクルを表すために使用される代理です。

スケジューラの実行

プロセスのスケジューリングを開始するには、Control 領域内の Setup ボタンに続いて Start ボタンを押して下さい。

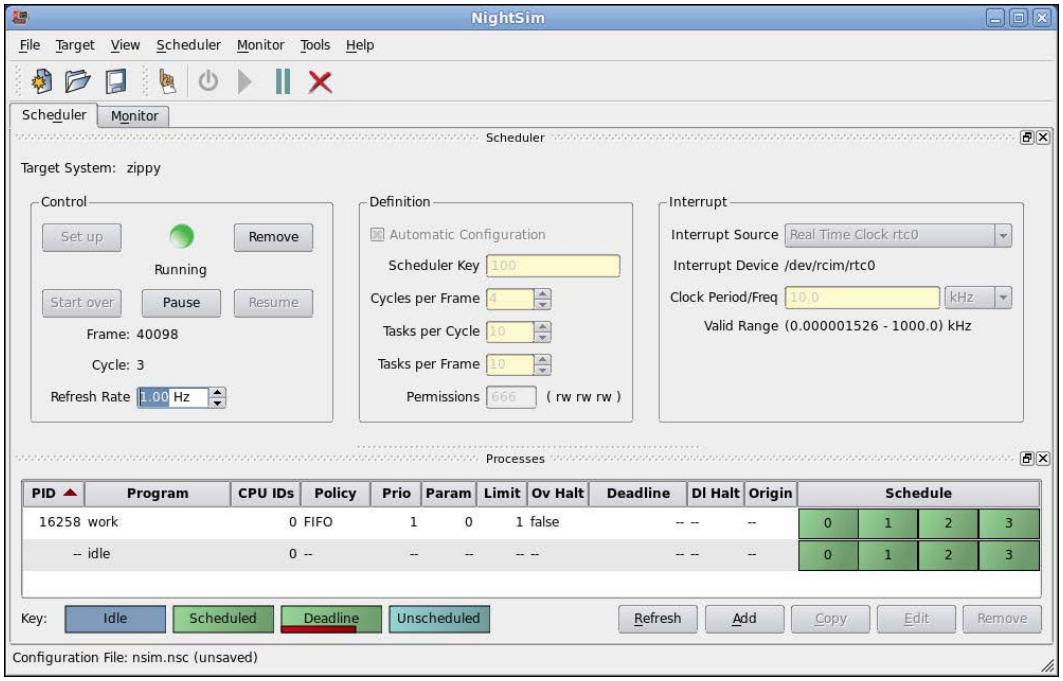


図 7-4. 開始されたスケジューリング

Cycle が 0 と 3 の間で振れるにつれて Control 領域の Frame カウントが増え始めることに注目して下さい。

プロセスの実行を監視するには、ウィンドウ上部にある Monitor タブをクリックして下さい。

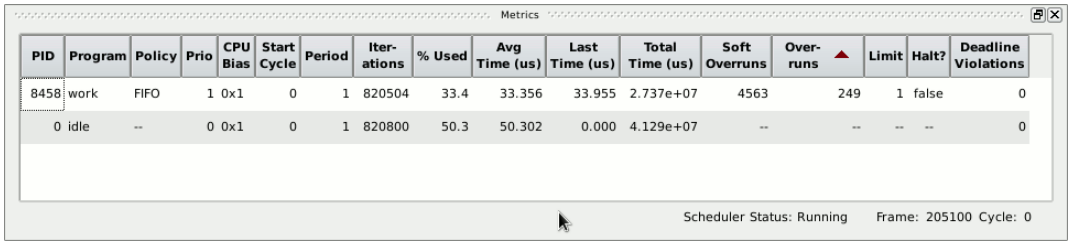


図 7-5. NightSim の Monitor ページ - Metrics パネル

本マニュアルでは、前述の図は他の NightSim ウィンドウから Metrics パネルを分離してパネルをより見易くなるようにしています。

NightSim Monitor Metrics パネルはスケジューラ上の個々のプロセスに関する統計値を提供します。それには PID、プログラム名、CPU バイアス、実行サイクル数、サイクル毎実行に関わる CPU 時間、オーバーラン回数、各プロセスで使用されたフレームの平均パーセンテージが含まれます。更なる統計値を表示するにはテーブルのコンテキスト・メニューの Select Fields...オプション項目を介して選択することが可能です。

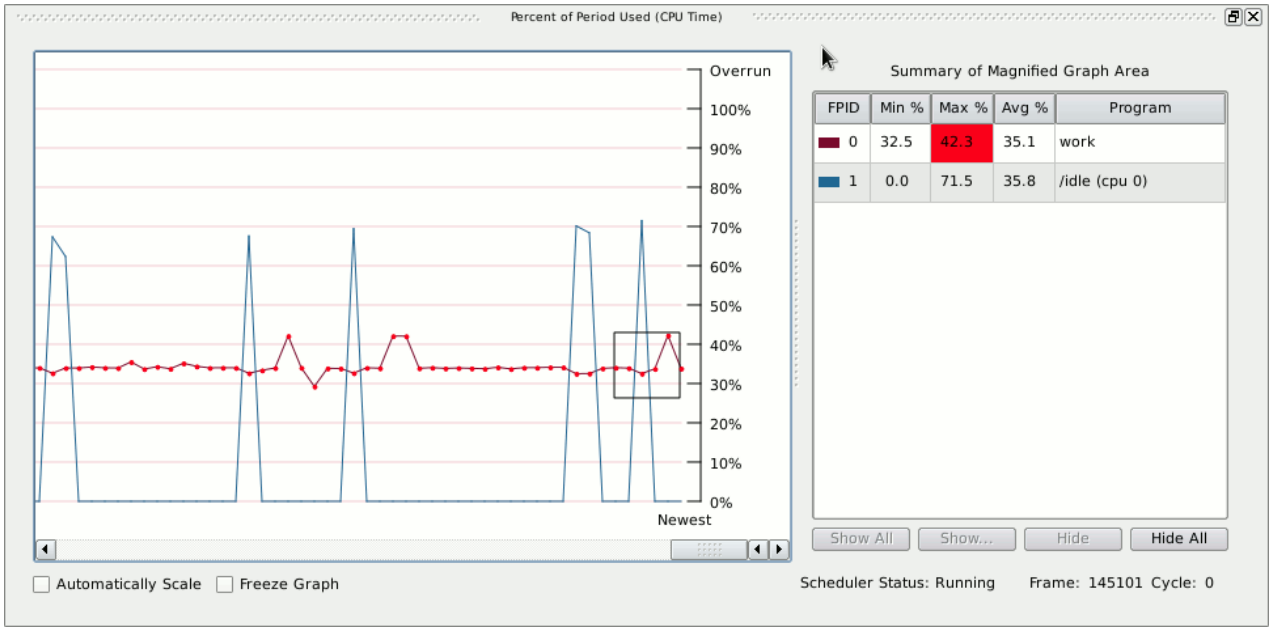


図 7-6. NightSim の Monitor ページ - Percent of Period Used パネル

ページの下半分には上図に抜き出した Percent of Period Used (CPU Time)グラフが表示されます。スケジューラー上の各プロセスに対して行が存在し、最後のサイクルで使用された時間のパーセンテージ(**CPU time**)が時間と共に描画されます。アプリケーションがその時間帯をオーバーランした場合、グラフ上に赤い点が表示されます。

四角い拡大鏡に含まれる点については右側のテーブルに詳述されます。

Important

プロセスが割り当てられた CPU 時間の 100%以上を使用していなくてもデッドラインをオーバーランする可能性はあります(他のプロセスが干渉する、I/O を待機している等)。むしろ、これは最高性能のシステムに調整する前には頻繁にある状況で、本章の後半で行います(7-9 ページの「オーバーラン検知とシステム・チューニング」を参照)。

Last Time 列を見て下さい。表示されるこの値は最後のサイクルの実行で各プロセッサーで使用されたマイクロ秒単位での CPU 時間です。**idle** プロセスに起因するこの値はサイクル内の利用可能な残りの CPU 時間を示します。

work プロセスのワークロードを調整して NightSim Monitor ウィンドウに表示されるその効果を確認します。

プログラム変数の変更に Datamon を利用

Data Monitoring Apprication Programming Interface は NightStar RT ツールの一部です。

データ監視は、変数名の指定によって選択された変数の値を監視、取得、変更するために Ada, C, Fortran 変数を含む実行可能プログラムを指定して、アドレス、型、サイズのような変数に関する情報を取得することが可能です。

データ監視は豊富な API を備えた強力な機能です。しかしながら、今回の目的のために 1 つの変数の値を変更するとても単純なプログラムを作成します。

データ監視に関する詳細については *Data Monitoring Reference Manual* を参照して下さい。

set_workload プログラムに関するソース・コードは次のようになります：

```
#include <stdlib.h>
#include <stdio.h>
#include <datamon.h>

#define check(x) ¥
    if((x)) {fprintf(stderr, "%s¥n", dm_get_error_string());exit(1);}

main(int argc, char * argv[])
{
    program_descriptor_t pgm;
    object_descriptor_t obj;
    char buffer[100];

    if (argc != 2) {
        fprintf (stderr, "Usage: set_workload integer-value¥n");
        exit(1);
    }

    check(dm_open_program("work",0,&pgm));
    check(dm_get_descriptor("workload",0,pgm,&obj));
    check(dm_get_value(&obj,buffer,sizeof(buffer)));
    check(dm_set_value(&obj,argv[1]));

    printf ("workload: old_value=%s, new_value=%s¥n", buffer, argv[1]);
}
```

dm_open_program 関数は指定されたプロセス名称と PID(この場合はゼロで、指定した名称に一致するプロセスを使用する呼び出しを指示)でデータ監視を初期化します。

dm_get_descriptor コールは指定された変数名を探して変数に関する情報を返します。また、**work** プロセスの変数の基礎となるメモリ・ページを監視プロセスにマップします。

dm_get_value と dm_set_value ルーティンは直接メモリを読み書きするのに使用する変数の値を返すおよび設定します。**work** プロセスはワークロード変数の値を変更すること以外に影響はありません。

set_workload.c ソース・ファイルは 1-4 ページの「チュートリアル・ディレクトリの生成」の操作で現在の作業ディレクトリにコピーされています。

次のコマンドを使ってプログラムをコンパイルして下さい：

```
cc -g -o set_workload set_workload.c -ldatamon -lccur_rt
```

次のコマンドを実行して **work** プロセス内の workload 変数の値を変更して下さい：

```
./set_workload 0
```

前述のソース・コードで示すようにプログラムは workload 変数の以前の値を出力して、続いて **set_workload** への引数として指定された値を設定します。

./work の Last Time 列は NightSim Monitor ウィンドウに表示されているように減少したワークロードに影響を受けます。

./work の平均の Last Cycle 時間が約 50 マイクロ秒になるまで **set_workload** プログラムを使い様々な workload の値を試して下さい。調整後に Monitor メニューから Clear Performance Data を選択、もしくは単にグラフを見て work の行が 50%近くになった時に調整を停止しても構いません。

Overrun Detection and System Tuning

プロセスの次のサイクルが始まっても前のサイクルの実行が終了していない場合、スケジューリングの*オーバーラン*が発生します。

NightSim Monitor ウィンドウには各プロセスのオーバーラン回数が含まれています。

何回かオーバーランが **work** プロセスで発生している可能性があります。

NOTE

まだオーバーランが発生していない場合、システムに追加の負荷を加えて下さい。別のターミナル・セッションで次のコマンドを実行すると必要とする効果が得られるはずです：

```
find / -print
```

前章で説明した NightStar ツールはプロセスのオーバーランの明確な原因を究明するために適しています。NightStar のカーネル・トレーシングはプロセス・コンテキスト・スイッチ、割り込み、システムコール、マシン例外を含む全ての CPU に関するシステムの動作の詳細な表示を提供します。

簡略化のため、オーバーランの原因がスケジューラーとは無関係の追加の動作が **work** を実行している CPU 上で発生していることに起因しているものとみなします。

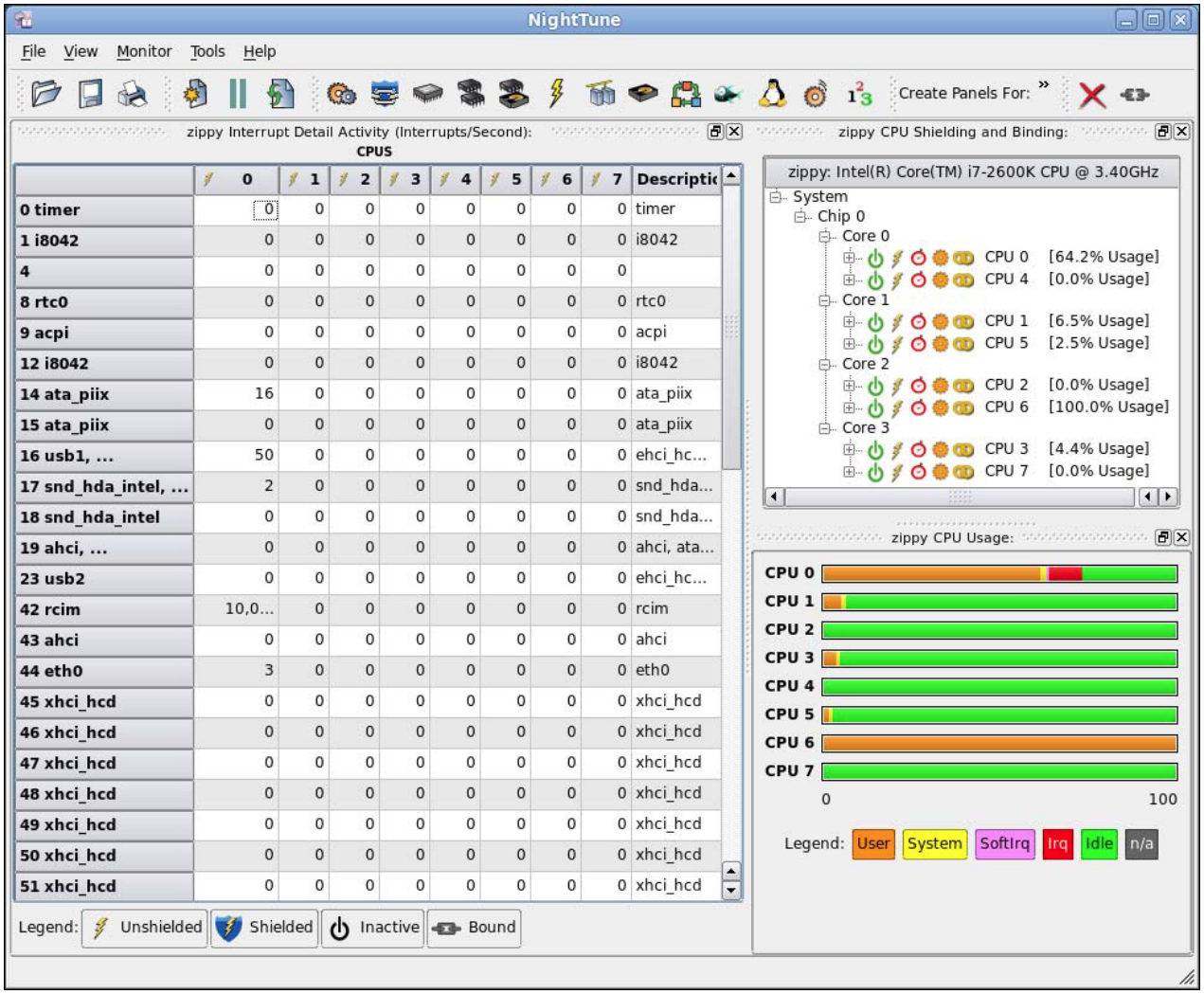
スケジューラーに関連する CPU を他の動作からシールドするために NightTune を使用します。

NOTE

お手持ちのシステムがシングル CPU である場合、本項の残りの部分は応用できません。その場合は 7-14 ページの「スケジューラのシャット・ダウン」へ進んで下さい。

1-4 ページの「チュートリアル・ディレクトリの生成」の操作で現在の作業ディレクトリにコピーされた **ntune.config** ファイルを使って NightTune を起動して下さい：

```
ntune -c ./ntune.config &
```



	0	1	2	3	4	5	6	7	Descriptic
0 timer	0	0	0	0	0	0	0	0	timer
1 i8042	0	0	0	0	0	0	0	0	i8042
4	0	0	0	0	0	0	0	0	
8 rtc0	0	0	0	0	0	0	0	0	rtc0
9 acpi	0	0	0	0	0	0	0	0	acpi
12 i8042	0	0	0	0	0	0	0	0	i8042
14 ata_piix	16	0	0	0	0	0	0	0	ata_piix
15 ata_piix	0	0	0	0	0	0	0	0	ata_piix
16 usb1, ...	50	0	0	0	0	0	0	0	ehci_hc...
17 snd_hda_intel, ...	2	0	0	0	0	0	0	0	snd_hda...
18 snd_hda_intel	0	0	0	0	0	0	0	0	snd_hda...
19 ahci, ...	0	0	0	0	0	0	0	0	ahci, ata...
23 usb2	0	0	0	0	0	0	0	0	ehci_hc...
42 rcim	10,0...	0	0	0	0	0	0	0	rcim
43 ahci	0	0	0	0	0	0	0	0	ahci
44 eth0	3	0	0	0	0	0	0	0	eth0
45 xhci_hcd	0	0	0	0	0	0	0	0	xhci_hcd
46 xhci_hcd	0	0	0	0	0	0	0	0	xhci_hcd
47 xhci_hcd	0	0	0	0	0	0	0	0	xhci_hcd
48 xhci_hcd	0	0	0	0	0	0	0	0	xhci_hcd
49 xhci_hcd	0	0	0	0	0	0	0	0	xhci_hcd
50 xhci_hcd	0	0	0	0	0	0	0	0	xhci_hcd
51 xhci_hcd	0	0	0	0	0	0	0	0	xhci_hcd

図 7-7. Interrupt と CPU Shielding & Binding パネルを含む NightTune

NightTune ウィンドウが現れて割り込み発生状況と全 CPU のシールドとバウンド状況が表示されます。

CPU Shielding & Binding パネルの System アイコンを右クリックしてコンテキスト・メニューから Expand All を選択して下さい。

work プロセスが CPU 0 の Bound Processes リストに載っていることに注目して下さい。

RCIM の割り込みを CPU 0 にバインドして他の全ての動作から CPU 0 をシールドするには以下の操作を行ってください：

- Interrupt Detail Activity パネルの Description 列に rcim という単語を含んでいるセルを探して下さい。

NOTE

Description ヘッダを表示するには NightTune ウィンドウまたは Interrupt Detail Activity パネルのサイズを変える必要があるかもしれません。

- カーソルが Interrupt パネルの **rcim** を含んでいる Description 列のセルの上にある間に左マウス・ボタンを押したままにして、次に CPU Shielding and Binding パネル内の CPU 0 の行に割り込みをドラッグしてマウスボタンを離して下さい。**rcim** の割り込みは直ちに CPU 0 にバインドされます。

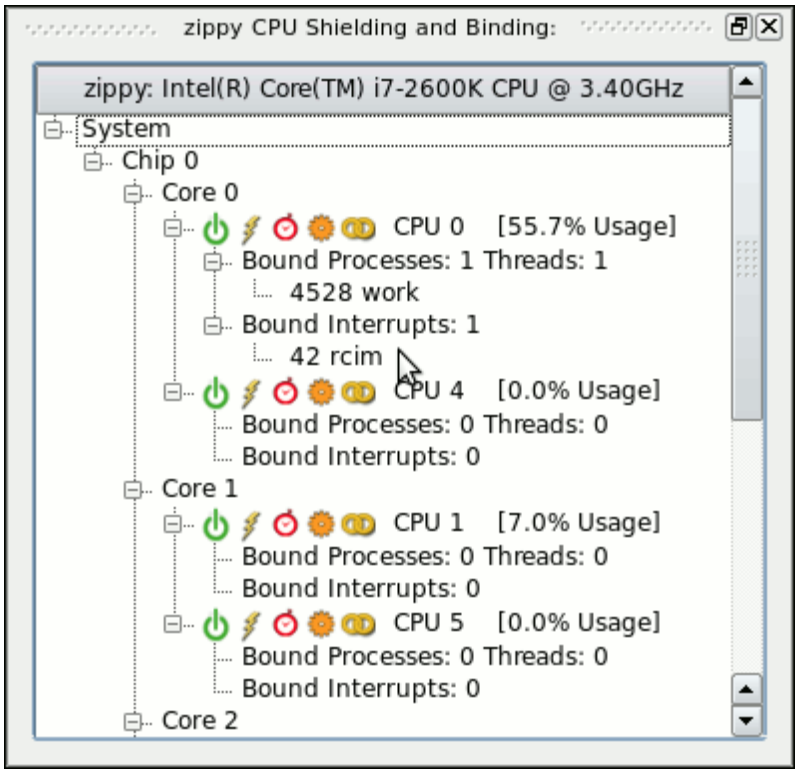


図 7-8. CPU 0 にバインドされたプロセスと割り込み

- CPU Shielding and Binding パネルの任意の場所で右クリックしてコンテキスト・メニューから Change Shielding...オプションを選択して下さい。

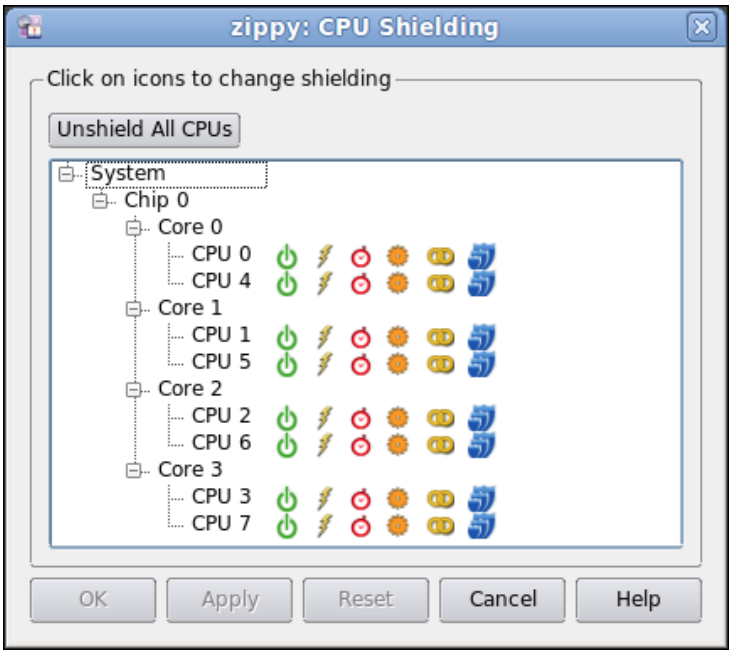


図 7-9. CPU Shielding ダイアログ

- CPU 0 の並びの Maximize Shielding アイコンをクリックして下さい(Maximize Shielding アイコンは盾の絵が 3 つ重なっている右端のアイコン)。

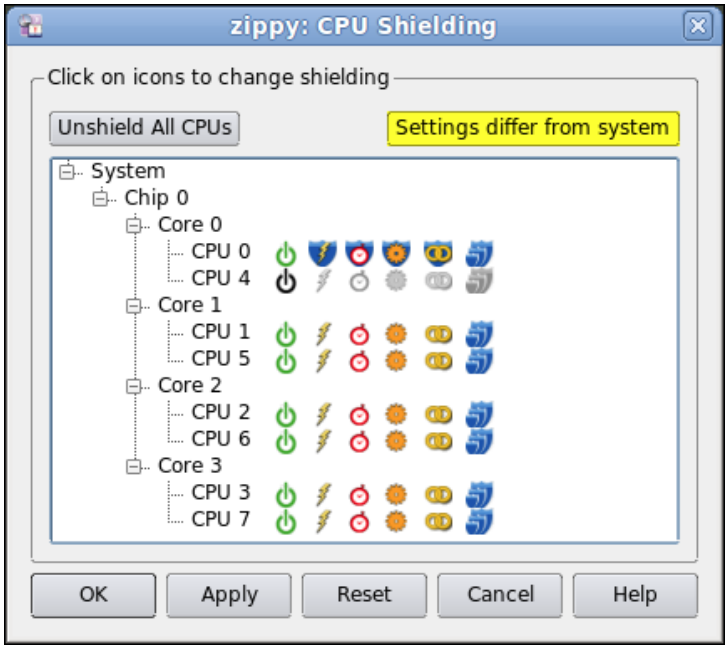


図 7-10. シールド変更中

CPU 0 の並びは **work** と **rcim** 以外の全てのプロセスと割り込みは CPU 0 からシールドされることを示す表示に変わります。更に、ハイパースレッドのシブリング CPU(この場合、CPU 0 の下に表示される CPU 4)は、CPU 上のハイパースレッド化された実行が CPU 0 を干渉しないようにダウンのマークになります。

NOTE

ハイパースレッド化された CPU0 のシブリングは CPU 4 以外の論路 CPU 番号である可能性があります。

NOTE

CPU がハイパースレッド・グループに表示されていない場合、お手持ちのシステムがハイパースレッドをサポートしていない、またはハイパースレッドが有効ではない可能性があります。

NOTE

お手持ちのシステムがハイパースレッド CPU である場合に CPU0 のシブリングをダウンさせることが出来ないことが起こり得ます。これはシブリング CPU にバインドされた他のプロセスや割り込みがある場合に発生する可能性があります。この場合、それらを CPU Shielding and Binding パネルのコンテキスト・メニューを使ってアンバインドすることが可能ですが、一部の割り込みはアンバインドできないことを理解して下さい(例えば、一部のシステムの hpet 割り込み等)。

- シールドの変更を有効にするには OK ボタンを押して下さい。

NightSim Monitor ウィンドウに戻って Overrun の列を見て下さい。オーバーランが発生しなくなっていることが期待できます。Monitor メニューから Clear Performance Data オプション項目を選択してオーバーラン回数をクリアして下さい。この操作は全ての統計値をゼロにリセットします。

オーバーランがまだ発生しているかどうかを見るには Overrun 列を注視して下さい。

システムが適切に構成されている場合、スケジューラーはシールドされた CPU 上ではオーバーランせずに実行し続けるはずです。

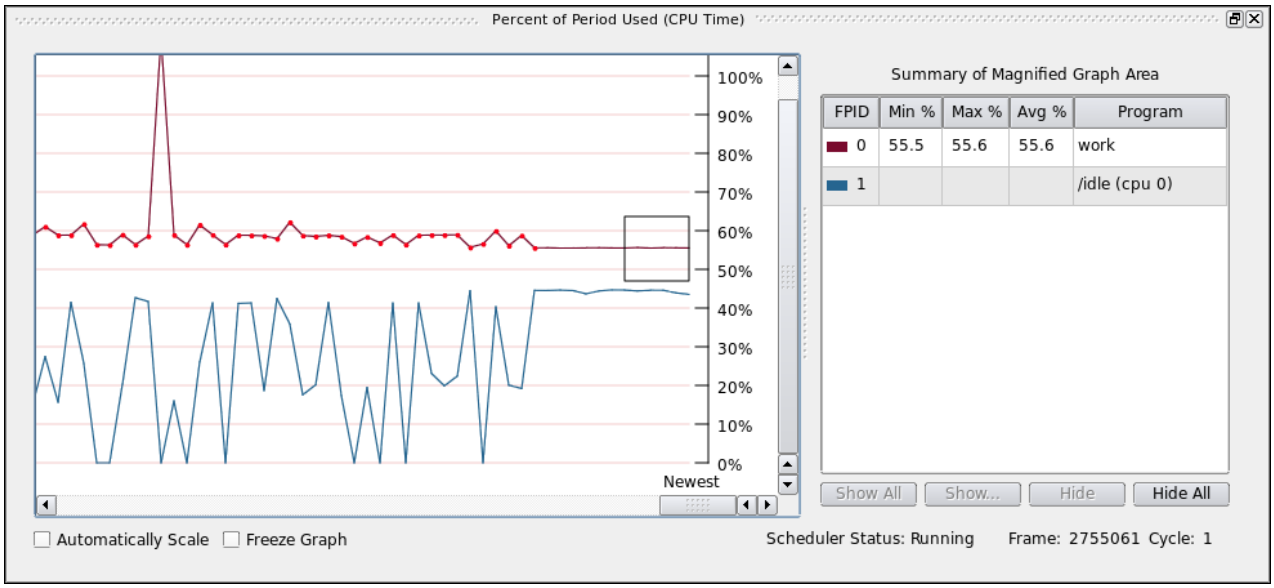


図 7-11. NightSim の Percentage of Period パネル - シールド CPU

上図では、NightTune で行ったシールドの操作によってプロセスのオーバーランが止まったのを見ることが出来ます。

スケジューラのシャット・ダウン

Scheduler ページに戻って、スケジューラーを止めるために Remove ボタンを押して下さい。スケジューラーに関連するプロセスを終了するかどうかを尋ねるダイアログが現れたら Yes を押して下さい。

File メニューから Exit メニュー項目を選択して NightSim を終了して下さい。**nsim.nsc** への変更を保存するかどうかを尋ねるダイアログが現れた場合、No を押して下さい。

CPU 0 へのシールド属性をクリアしたい場合は NightTune を使って以前の状態にシステムを戻して下さい。

File メニューから Exit を選択して NightTune を終了して下さい。

これで NightStar RT チュートリアルの NightSim の部は終了です。

A

チュートリアル・ファイル

以下のセクションでは *NightStar RT Tutorial* で使用されるファイルのソース・リストを示します。

- api.c
- app.c
- function.c
- report.c
- set_workload.c
- set_rate.c
- work.c
- worker.c

api.c

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <errno.h>
#include <string.h>
#include <nprobe.h>

int cycles = 0;
int overruns = 0;
char * sample;

// Perform the work of consuming a single Data Recording sample from NightProbe.
//
int
work (FILE * ofile, np_handle h, np_header * hdr) {
    np_item * i;
    int status;
    int which;
```

```
    // Read one sample, which may contain data for multiple processes
    // and variables.
    //
    status = np_read (h, sample);
    if (status <= 0) {
        return status;
    }

    cycles++;

    fprintf (ofile, "Sample %d¥n", cycles);
    for (i = hdr->items; i; i = i->link) {
        char buffer [1024];
        sprintf (buffer, "item: %s:", i->name);
        fprintf (ofile, "%-30s", buffer); // Nice formatting :-)

        // Display the value of each item.
        // For arrays, format each individual item.
        //
        for (which = 1; which <= i->count; ++which) {
            char * image = np_format (h, i, sample, which);

            if (image != NULL) {
                fprintf (ofile, " %s", image);
            } else {
                fprintf (ofile, "¥n<error: %s>¥n", np_error (h));
                return -1;
            }

            free (image);
        }
        fprintf (ofile, "¥n");
    }
    fflush (ofile);

    return 1;
}

int
main (int argc, char *argv[])
{
    np_handle h;
    np_header hdr;
    np_process * p;
    np_item * i;
    int fd;
    int status;
    FILE * ofile = stdout;
```

```
fd = 0; // stdin

status = np_open (fd, &hdr, &h);

if (status) {
   fprintf (stderr, "%s¥n", np_error (h));
   exit(1);
}

sample = (char *) malloc(hdr.sample_size);
if (sample == NULL) {
   fprintf (stderr, "insufficient memory to allocate sample buffer¥n");
   exit(1);
}

for (p = hdr.processes; p; p = p->link) {
   if (p->pid >= 0) {
      fprintf (ofile, "process: %s (%d)¥n", p->name, p- >pid);
   } else {
      fprintf (ofile, "resource: %s (%s)¥n", p->name, p- >label);
   }
}
fprintf (ofile, "¥n");

for (i = hdr.items; i; i = i->link) {
   fprintf (ofile, "item: %s (%s), size=%d bits, count=%d, type=%d¥n",
            i->name, i->process->name, i->bit_size, i- >count, i->type);
}
fprintf (ofile, "¥n");

for (;;) {
   status = work (ofile, h, &hdr);
   if (status <= 0) break;
}

fprintf (ofile, "Data Recording done: %d cycles fired, %d overruns¥n",
         cycles, overruns);

if (ofile != stdout) {
   fclose (ofile);
}

if (status < 0) {
   fprintf (stderr, "%s¥n", np_error (h));
}

np_close (h);
```

```
        // At this point, file descriptor 0 remains open, but is no
        // longer a NightProbe Data File/Stream.
    }
```

app.c

```
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <pthread.h>
#include <errno.h>
#include <ntrace.h>
#include <math.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/prctl.h>

static void * heap_thread (void * ptr);
static void * watchdog_thread (void * ptr);
static int add_link (void);
static int nosighup (void);
extern void work (int control);

typedef struct {
    char * name;
    int count;
    double delta;
    double angle;
    double value;
} control_t;

control_t data[2] = { { "sin", 0, M_PI/360.0, 0.0, 0.0 },
                      { "cos", 0, M_PI/360.0, 0.0, 0.0 } };
enum { run, hold } state;
int rate = 50000000;
int sema;

extern double
FunctionCall(void)
{
    return data[0].value + data[1].value;
}

void *
sine_thread (void * ptr)
{
    control_t * data = (control_t *)ptr;
    struct sembuf wait = {0, -1, 0};
    work(1);

    trace_set_thread_name (data->name);

    for (;;) {
```

```
        semop(sema, &wait, 1);
        data->count++;
        data->angle += data->delta;
        data->value = sin(data->angle);
    }
}

void *
cosine_thread (void * ptr)
{
    control_t * data = (control_t *)ptr;
    struct sembuf wait = {0, -1, 0};
    work(1);

    trace_set_thread_name (data->name);

    for (;;) {
        semop(sema, &wait, 1);
        data->count++;
        data->angle += data->delta;
        data->value = cos(data->angle);
    }
}

int main (int argc, char * argv[])
{
    pthread_t thread;
    pthread_attr_t attr;
    struct sembuf trigger = {0, 2, 0};
    const char * data_file = strdup("/tmp/data");

    if (argc > 1) {
        data_file = argv[1];
    }
    trace_begin (data_file,NULL);

    pthread_attr_init(&attr);
    pthread_create (&thread, &attr, watchdog_thread, NULL);

    sema = semget (IPC_PRIVATE, 1, IPC_CREAT+0666);

    pthread_attr_init(&attr);
    pthread_create (&thread, &attr, sine_thread, &data[0]);

    pthread_attr_init(&attr);
    pthread_create (&thread, &attr, cosine_thread, &data[1]);

    pthread_attr_init(&attr);
    pthread_create (&thread, &attr, heap_thread, NULL);

    for (;;) {
        struct timespec delay = { 0, rate } ;
```

```
        nanosleep(&delay,NULL);
        work (random() % 1000);
        if (state != hold) {
            semop(sema,&trigger,1);
        }
    }

    trace_end () ;
}

void * ptrs[5];

static void *
heap_thread (void * unused)
{
    int i = 5;
    int scenario = -1;
    void * ptr;
    int * * iptr;
    extern void * alloc_ptr (int size, int swtch);
    extern void free_ptr (void * ptr, int swtch);
    trace_set_thread_name("heap_thread");

    for (;;) {
        sleep (5);
        switch (scenario) {
        case 1:
            // Use of freed pointer
            ptr = alloc_ptr(1024,3);
            free_ptr(ptr,2);
            memset (ptr, 47, 64);
            break;
        case 2:
            // Double-free
            ptr = alloc_ptr(1024,3);
            free_ptr(ptr,2);
            free(ptr);
            break;
        case 3:
            // Overwriting past end of an allocated block
#define MyString "mystring"
            ptr = alloc_ptr(strlen(MyString),2);
            strcpy (ptr,MyString); // oops -- forgot the zero- byte
            break;
        case 4:
            // Uninitialized use
            iptr = (int * *) alloc_ptr(sizeof(void*),2);
            if (*iptr) **iptr = 2778;
            break;
        case 5:
            // Leak -- all references to block removed
            ptr = alloc_ptr(37,1);
```

```
            ptr = 0;
            break;
        case 6:
            // Some more allocations we'll check on...
            ptrs[0] = alloc_ptr(1024*1024,3);
            ptrs[1] = alloc_ptr(1024,2);
            ptrs[2] = alloc_ptr(62,1);
            ptrs[3] = alloc_ptr(4564,3);
            ptrs[4] = alloc_ptr(8177,3);
            break;
        }

        (void) malloc(1);
        scenario = 0;
    }
}

void * func3 (int size, int count)
{
    return malloc(size);
}

void * func2 (int size, int count)
{
    if (--count > 0) return func3(size,count);
    return malloc(size);
}

void * func1 (int size, int count)
{
    if (--count > 0) return func2(size,count);
    return malloc(size);
}

void free3 (void * ptr, int count)
{
    free(ptr);
}

void free2 (void * ptr, int count)
{
    if (--count > 0) {
        free3(ptr,count);
        return;
    }
    free(ptr);
}

void free1 (void * ptr, int count)
{
    if (--count > 0) {
        free2(ptr,count);
        return;
    }
```

```
    free(ptr);
}

void * alloc_ptr (int size, int count)
{
    return func1(size,count);
}

void free_ptr (void * ptr, int count)
{
    free1(ptr,count);
}

void work (int control)
{
    volatile double calculations[2048];
    volatile double d = 0.0;
    int i;
    for (i=0; i<2048; ++i) {
        calculations[i] = 3.14159;
    }
        for (i=0; i<control*10; ++i) {
        d = d*d; calculations[i%2048] = d;
    }
}

struct node_t {
    int value;
    struct node_t * link;
};
struct node_t * head;
struct node_t * tail;

static int add_link (void)
{
    static int count;
    count++;
    if (count > 5 && count < 1000) {
        struct node_t * n = (struct node_t*)malloc(sizeof(struct node_t));
        n->value = count;
        n->link = NULL;
        if (tail) {
            tail->link = n;
        } else {
            head = n;
        }
        tail = n;
    }
}

#include <signal.h>
static int nosighup (void)
```

```
{
    struct sigaction ignore;
    ignore.sa_flags = 0;
    ignore.sa_handler = SIG_IGN;
    sigemptyset(&ignore.sa_mask);
    sigaction(SIGHUP,&ignore,NULL);
}
#include <time.h>
#include <sched.h>
#include <stdio.h>

void * watchdog_thread (void * unused)
{
    double deadline = 0.050;
    struct timespec ts;
    double last, now;
    int deadline_violation_fatal = 0;
    struct sched_param param;

    // prctl(PR_SET_NAME,"watchdog_thread"); or
    trace_set_thread_name("watchdog_thread");

    param.sched_priority = 50;
    if (sched_setscheduler(0,SCHED_FIFO,&param)) {
        printf("Warning: sched_setscheduler failed: %s¥n",strerror(errno));
    }
    clock_gettime(CLOCK_REALTIME,&ts);
    last = (double)ts.tv_sec + ((double)ts.tv_nsec)/ 1000000000.0;
    for (;;) {
        clock_gettime(CLOCK_REALTIME,&ts);
        now = (double)ts.tv_sec + ((double)ts.tv_nsec)/ 1000000000.0;
        if (now-last > deadline) {
            printf("!!!!!!!!!!!!!!!!!!!!!!!!!!!!¥n");
            printf("Deadline missed by %f seconds!!!!!¥n",(now- last)-deadline);
            printf("!!!!!!!!!!!!!!!!!!!!!!!!!!!!¥n");
            if (deadline_violation_fatal) break;
        }
        last = now;
    }
    for (;;) {
        sleep(1);
    }
}
```

function.c

```
double
FunctionCall(void)
{
    static double counter;
    return counter++;
}
```

report.c

```
#include <stdio.h>

void report (char * caller, double value)
{
    static int count;

    if (++count % 40) printf ("The value from %s is %f¥n", caller, value);
}
```

set_workload.c

```
#include <stdlib.h>
#include <stdio.h>
#include <datamon.h>

#define check(x) if((x)) {fprintf(stderr, "%s¥n", dm_get_error_string());exit(1);}

main(int argc, char * argv[])
{
    program_descriptor_t pgm;
    object_descriptor_tobj;
    char buffer[100];

    if (argc != 2) {
        fprintf (stderr, "Usage: set_workload integer- value¥n");
        exit(1);
```

```
    }

    check(dm_open_program("work",0,&pgm));
    check(dm_get_descriptor("workload",0,pgm,&obj));
    check(dm_get_value(&obj,buffer,sizeof(buffer)));
    check(dm_set_value(&obj,argv[1]));

    printf ("workload: old_value=%s, new_value=%s¥n", buffer, argv[1]);
}
```

set_rate.c

```
#include <stdlib.h>
#include <stdio.h>
#include <datamon.h>

#define check(x) if((x)) {fprintf(stderr, "%s¥n", dm_get_error_string());exit(1);}

main(int argc, char * argv[])
{
    program_descriptor_t pgm;
    object_descriptor_tobj;
    char buffer[100];

    if (argc != 2) {
        fprintf (stderr, "Usage: set_rate: integer-value¥n");
        exit(1);
    }

    check(dm_open_program("app",0,&pgm));
    check(dm_get_descriptor("rate",0,pgm,&obj));
    check(dm_get_value(&obj,buffer,sizeof(buffer)));
    check(dm_set_value(&obj,argv[1]));

    printf ("rate: old_value=%s, new_value=%s¥n", buffer, argv[1]);
}
```

work.c

```
#include <stdlib.h>

int workload = 1000;
```

```
int
main()
{
    int data = 0;
    int i;
    volatile double d = 1.0;

    while (fbswait()>=0) {
        data = !data;
        for (i=0; i<workload; ++i) d = d/d;
    }
}
```

work.c

```
#include <time.h>
#include <stdio.h>

static int elapsed(struct timespec*,struct timespec*);

int outer = 50;
int inner = 10;
int threashold = 200;
int usecs;
int overruns;

double
work(void)
{
    volatile double d = 0.0;
    int i,j;
    for (i=0; i<outer; ++i) {
        for (j=0; j<inner; ++j) {
            d *= d;
        }
    }
}

int
main()
{
    struct timespec start;
    struct timespec stop;
    for (;;) {
        struct timespec t = {0,10000000};
        nanosleep(&t,0);
        clock_gettime(CLOCK_REALTIME,&start);
            work();
        clock_gettime(CLOCK_REALTIME,&stop);
```

```
        usecs = elapsed(&stop,&start);
        if (usecs > threashold) {
            printf ("Overrun %d¥n",++overruns);
        }
    }
}


static
int
elapsed (struct timespec * stop, struct timespec * start)
{
    int sec = stop->tv_sec - stop->tv_sec;
    int nsec;
    if (stop->tv_nsec < start->tv_nsec) {
        sec--;
        nsec = 1000000000-(start->tv_nsec-stop->tv_nsec);
    } else {
        nsec = stop->tv_nsec-start->tv_nsec;
    }
    return sec * 1000000 + nsec/1000;
}
```