PTL2 — Portable Thread Library

PTL2-current-990204

安倍 広多
Kota Abe
k-abe@media.osaka-cu.ac.jp

Media Center, Osaka City University

1 イントロダクション

PTL – Portable Thread Library は, BSD UNIX の下でマルチスレッド環境を構築するためのライブラリです.

通常の UNIX プロセスでは, 制御の流れ (コンテキスト) は一つしかありませんが, このライブラリを用いることによって, 複数の制御の流れを作り出すことが出来ます. 一つの制御の流れのことを スレッド と言います. 複数のスレッドが一つの UNIX プロセスの中でコンテキストスイッチを行ないながら見かけ上並行に動きます.

複数のプロセスを fork() を用いて並行に走らせる場合と比べると, スレッドを用いた並行動作には以下のような利点があります.

- プロセスが増えないのでシステムの負荷がほとんど増えない. (プロセスは重い資源です.)
- スレッド間でアドレス空間を共有しているので, データのやりとりや同期を素早く行なえる.
- スレッドのスケジューリングを自分で行なえるので, 柔軟性が高い.

PTL は, スレッドを扱うためのインタフェースとして, POSIX 1003.1c という規格に準拠した関数群を提供しています.

このライブラリ, 及びマニュアルは, まだ完成されたものではありません. 改善すべき点や問題点, バグ等を発見した場合, 御連絡下さい. 次のリリースで改善したいと思います.

2 ライブラリについて

2.1 インストール

インストールに関しては, アーカイブに含まれる ‘`INSTALL`’ を参照して下さい.

2.2 ライブラリの使用法

PTL を使用するプログラムをコンパイルする時には, 必ず 以下のように PTL のインクルードディレクトリを先に見るようにコンパイラに指示を与えなければなりません. また, 全てのファイルで `pthread.h` を必ず include しなければなりません.

```
gcc -I/usr/local/PTL/include foo.c -o foo -L/usr/local/PTL/lib -lPTL
```

`libPTL.a` をコンパイルする際に使用した C コンパイラを用いることを推奨します.

PTL は, C++ から使用することも可能です. 大域変数のコンストラクタの実行の前に PTL の初期化を行なうためにトリッキーな方法を用いているため, 以下の C++ コンパイラでのみ動作を確認しています.

- gcc 2.*
- Sun C++ (SC1.0, SC2.0.1)

2.3 PTL に関する情報源

PTL の Web page を用意しています.

```
http://www.media.osaka-cu.ac.jp/~k-abe/PTL/
```

2.4 PTL の入手法

最新版の PTL は以下の anonymous ftp から入手できます.

```
ftp://ftp.media.osaka-cu.ac.jp/pub/PTL/
```

3 ライブラリの概要

この章では, スレッドを用いたプログラミングの概要について解説します.

3.1 スレッドの操作の概要

3.1.1 スレッドの生成

新たなスレッドを生成するためには, `pthread_create()` を用います. この関数は, 指定したアトリビュート (第 3.2.3 節 [スレッドアトリビュートオブジェクト], 7 ページ.) を用いてスレッドオブジェクト (ライブラリ内部で使用するオブジェクトで, スレッドを管理するために用いる) を生成し, 指定した関数 (スレッド開始関数) からスレッドの実行を開始します.

3.1.2 スレッドの終了

スレッドの実行は, 以下の場合に終了します.

- `pthread_create()` で指定した関数から復帰 (return) した場合.
- スレッドが `pthread_exit()` を呼び出した場合.
- (他の) スレッドが `pthread_cancel()` を呼び出すことによって, スレッドがキャンセルされた場合.

  `pthread_cancel()` は, 指定したスレッドの実行を中止するように要求する関数です. (第 3.7 節 [キャンセルの概要], 22 ページ.)

スレッドが終了する際, 以下の動作が行なわれます.

1. `pthread_cleanup_push()` 等によってプッシュされ, まだ`pthread_cleanup_pop()` 等によってポップされていない Cleanup ハンドラが, プッシュした逆の順序で実行されます. (第 3.7.3 節 [スレッド Cleanup], 24 ページ.)
2. `pthread_join()` を用いてそのスレッドの終了を待っている他のスレッドに, スレッドの返り値 (スレッド開始関数の返り値, あるいは`pthread_exit()` の引数) が渡され, それらのスレッドがアンブロックされます.
3. Thread-Specific データのデストラクタ関数が, 定義されない順序で呼ばれ, そのスレッドの Thread-Specific データを全て破棄します. (第 3.4 節 [Thread-Specific データの概要], 18 ページ.)
4. スレッドが既にデタッチされていれば, スレッドオブジェクトを回収します. (第 3.1.4 節 [スレッドの削除], 4 ページ.)

スレッドが終了したあと, 他にスレッドが存在しなければ, プロセスは exit します. このときの終了ステータスは, `pthread_set_exit_status_np()` によって設定されていればその値, 設定されていなかった場合は 0 になります.

3.1.3 スレッドの終了の Wait

他のスレッドの終了を待つには, `pthread_join()` を用います. この関数を呼び出したスレッドは, 指定したスレッドが終了するまでブロックします. 複数のスレッドが同一のスレッドに対して `pthread_join()` を呼び出した場合, 指定したスレッドが終了すると, 全てのスレッドがアンブロックされます.

デタッチされているスレッドに対して `pthread_join()` を呼んではいけません.

3.1.4 スレッドの削除

スレッドは, 終了してもデタッチされるまでは完全には削除されません. スレッドが完全に削除されるためには, 以下のいずれかの操作が必要です.

- `pthread_detach()` を呼び出す.
- `pthread_create()` を用いてスレッドを生成する際に detachstate 属性を 1 にしたスレッドアトリビュートを指定する. (第 3.2.3 節 [スレッドアトリビュートオブジェクト], 7 ページ.)
- `pthread_join()`を呼び出す.

スレッドが終了する前に `pthread_detach()` が呼ばれると, スレッドに「終了したら削除しても良い」というフラグを立てます.

デタッチされたスレッドに対して `pthread_join()` してはいけません.

メモリを有効に利用するため, スレッドはいつかはデタッチされるべきです.

3.1.5 スレッドのサスペンドの概要

スレッドから, 他のスレッドをサスペンド (一時停止) させることができます. また, スレッドを生成する際に, サスペンドされた状態で生成することもできます.

スレッドをサスペンドするには, `pthread_suspend_np()` を用います. またサスペンドされたスレッドを再開するには `pthread_resume_np()` を用います.

スレッドをサスペンドされた状態で生成するためには, サスペンドステート属性を 1 にしたスレッドアトリビュートオブジェクトを `pthread_create()` で指定します (第 3.2.3.8 節 [サスペンドステート], 11 ページ.) こうやって生成されたスレッドは `pthread_resume_np()` が呼ばれるまで実行されません.

3.1.6 パッケージの動的な初期化

プログラムには, 初期化ルーチンなど, 一度しか実行して欲しくない部分があります. `pthread_once()` を使ってこれを実現できます.

`pthread_once()` は, 指定した関数を一度だけ実行させるために用います. 2 度目以降の `pthread_once()` の呼び出し時に, 最初の `pthread_once()` の実行が終了していなかった場合は, 最初の `pthread_once()` の

実行が終了するまでブロックします. 2 度目以降の `pthread_once()` の呼び出しは, 最初の呼び出しが終了していた場合, 単に無視されます.

`pthread_first_np()`, `pthread_first_done_np()` は, 関数内の特定のブロック (初期化ブロック) を一度だけ実行するために用います. 関数内 static 変数の初期化等に有効です. `pthread_first_np()` は最初に実行すると 1 を返し, 2 度目以降の呼び出しには 0 を返します. `pthread_first_done_np()` は, 初期化ブロックの実行が終了したことを宣言します. 2 度目以降の `pthread_first_np()` 呼び出し時に, まだ `pthread_first_done_np()` が実行されていなければ, 実行されるまでブロックされます.

```
foo()
{
    static pthread_once_t once = PTHREAD_ONCE_INIT;
    static int need_initialized;
    if (pthread_first_np(&once)) {
        need_initialized = appropriate_value(); /* Initialize */
            :
            :
        pthread_first_done_np(&once);
    }
    ....
}
```

3.1.7 スレッドに対するネーミング

スレッドにはそれぞれ名前をつけることができます. これは, デバッグの時に役立ちます. スレッドを生成した時点では, 名前は付いていません.

スレッドに名前を付けるためには, `ptheread_setname_np()` を, 名前を得るためには, `pthread_getname_np()` を用います.

3.1.8 初期スレッドについて

`main()` 関数から始まるコンテキストのことを, 初期スレッドと言います. 初期スレッドのスケジューリング属性は, 以下の通りです.

スケジューリングポリシー
　SCHED_OTHER

スケジューリングプライオリティ
　SCHED_OTHER のプライオリティ範囲の中央値

初期スレッドがリターンすると, プロセス全体が exit します. 初期スレッドを終了したいが, 他のスレッドは動かしておきたい場合, `pthread_exit()` を使用してください.

また, 初期スレッドの Cancelability State については, 第 3.7.1 節 [Cancelability States], 23 ページ. を参照してください.

3.2 アトリビュートオブジェクトの概要

アトリビュートオブジェクト は, スレッド, Mutex, Condition Variable のさまざまな属性を記述するために用いられます. アトリビュートオブジェクトは, スレッド, Mutex, Condition Variable のそれぞれに対応してスレッドアトリビュートオブジェクト, Mutex アトリビュートオブジェクト, Condition アトリビュートオブジェクトがあります.

オブジェクト (スレッド, Mutex, Condition Variable) を生成する際には, 対応するアトリビュートオブジェクトを指定するか, デフォルトのアトリビュートを指定します.

3.2.1 アトリビュートオブジェクトの生成

アトリビュートオブジェクトの生成は以下の関数で行ないます.

スレッドアトリビュート
`pthread_attr_init()`

Mutex アトリビュート
`pthread_mutexattr_init()`

Condition アトリビュート
`pthread_condattr_init()`

これらの関数は, デフォルトの属性を持つアトリビュートオブジェクトを生成します. 属性を変更するためには, 第 3.2.3 節 [スレッドアトリビュートオブジェクト], 7 ページ., 第 3.2.4 節 [Mutex アトリビュートオブジェクト], 11 ページ., 第 3.2.5 節 [Condition アトリビュートオブジェクト], 12 ページ. を参照して下さい.

オブジェクトを生成する際に, (必要な) アトリビュートオブジェクトの内容はコピーされます. つまり, オブジェクトを生成した後にアトリビュートオブジェクトの内容を変更しても, 既に生成されたオブジェクトには影響を与えません.

3.2.2 アトリビュートオブジェクトの削除

アトリビュートオブジェクトの削除は以下の関数で行ないます.

スレッドアトリビュート
`pthread_attr_destroy()`

Mutex アトリビュート
`pthread_mutexattr_destroy()`

Condition アトリビュート
`pthread_condattr_destroy()`

オブジェクトを生成した後にアトリビュートオブジェクトを削除しても, 既に生成されたオブジェクトには影響を与えません.

3.2.3 スレッドアトリビュートオブジェクト

スレッドアトリビュートオブジェクト は, 新たに (これから) 生成するスレッドの属性を制御します. スレッドアトリビュートオブジェクトは以下のようにして使用します.

1. `pthread_attr_init()` を用いてスレッドアトリビュートオブジェクトを生成する.
2. 以下に述べる関数を用いてスレッドアトリビュートオブジェクトの各属性を変更する.
3. 生成したスレッドアトリビュートオブジェクトを指定して `pthread_create()` を呼び, スレッドを生成する.

あるいは, `pthread_create()` に与えるスレッドアトリビュートとして, NULL を与えることによってデフォルトのスレッドアトリビュートを用いることも出来ます.

スレッドアトリビュートオブジェクトで指定できる属性には以下のものがあります.

- スケジューリングポリシー
- スケジューリングプライオリティ
- Inherit スケジューリング
- コンテンションスコープ
- スタックプロパティ
- スタックサイズ
- デタッチステート
- サスペンドステート

PTL では, スレッドアトリビュートオブジェクトのデフォルトとして, それぞれの属性に以下の値を与えます. これらは, `pthread_attr_init()` で生成したスレッドアトリビュートの初期値, 及び `pthread_create()` でスレッドアトリビュートとして NULL を指定した場合に使用されます.

属性の値の意味に関しては後の章を参照してください.

属性	値
スケジューリングポリシー	SCHED_OTHER
スケジューリングプライオリティ	SCHED_OTHER のプライオリティ範囲の中央値
Inherit スケジューリング	PTHREAD_EXPLICIT_SCHED
コンテンションスコープ	PTHREAD_SCOPE_PROCESS
スタックプロパティ	PTHREAD_STACK_SAFE_NP
スタックサイズ	16Kbytes
デタッチステート	PTHREAD_CREATE_JOINABLE

サスペンドステート

0 (not suspended)

3.2.3.1 スケジューリングポリシー

スケジューリングポリシー は, スレッドがどのようにプロセスの中でスケジュールされるかを指定します. スケジューリングポリシーには以下の 3 種類があります.

`SCHED_FIFO`

最も高いプライオリティのスレッドが, ブロックするまで実行されます. 最も高いプライオリティに複数のスレッドが存在した場合, 最初のスレッドが, ブロックするまで実行されます.

`SCHED_RR`

最も高いプライオリティのスレッドが, ブロックするまで実行されます. 最も高いプライオリティに複数のスレッドが存在した場合, スレッドは途中で CPU を奪われ, そのプライオリティの次のスレッドが実行権を得ます (タイムスライス).

`SCHED_OTHER` (デフォルト)

SCHED_OTHER の全てのスレッドはタイムスライスによって CPU を奪われます. 高いプライオリティのスレッドほど CPU を多く握れますが, 低いプライオリティのスレッドにも実行権は回ります. SCHED_FIFO, SCHED_RR の実行可能なスレッドが存在すれば, このポリシーのスレッドの実行は行なわれません.

以下の方法で, スケジューリングポリシーを変更することが出来ます.

- `pthread_attr_setschedpolicy()` によってスレッドスケジューリングアトリビュートのスケジューリングポリシーを変更する. これによって, 新たに生成するスレッドの初期スケジューリングポリシーを変更できます.
- `pthread_setschedparam()` によって, 既存のスレッドのスケジューリングポリシーを変更する.

スケジューリングに関しての詳細は, 第 3.5 節 [スケジューリングの概要], 18 ページ. を参照して下さい.

3.2.3.2 スケジューリングプライオリティ

スケジューリングプライオリティ は, スレッドの実行の優先度を定めます. スケジューリングプライオリティの値の範囲は, スケジューリングポリシーによって異なります.

`sched_get_priority_max()`, `sched_get_priority_min()` によって, 各ポリシーのスケジューリングプライオリティの最大値, 最小値を得ることができます.

スケジューリングプライオリティは '`<sched.h>`' で定義される構造体 struct sched_param で指定します.

```
struct sched_param
    int sched_priority;
;
```

Scheduling Policy Attribute に関しての詳細は, 第 3.5 節 [スケジューリングの概要], 18 ページ. を参照してください.

3.2.3.3 Inherit スケジューリング

Inherit スケジューリング 属性は, `pthread_create()` で新たに生成されたスレッドが, 作成するスレッドのスケジューリング属性を継承するか, あるいはスレッドアトリビュートオブジェクトので指定したスケジューリング属性によって設定されるかを指定します.

Inherit スケジューリング属性の値は, `PTHREAD_EXPLICIT_SCHED` (デフォルト) か, `PTHREAD_INHERIT_SCHED` です. `pthread_create()` で指定したスレッドアトリビュートの Inherit スケジューリング属性が, PTHREAD_INHERIT_SCHED だった場合, 生成されるスレッドのスケジューリング属性 (スケジューリングポリシー, プライオリティ) は生成するスレッド (すなわち, `pthread_create()` を呼び出したスレッド) の値を引き継ぎます.

Inherit スケジューリング属性を変更するためには, `pthread_attr_setinheritsched()` 関数を使用します.

3.2.3.4 コンテンションスコープ

コンテンションスコープ 属性は, `PTHREAD_SCOPE_SYSTEM` か `PTHREAD_SCOPE_PROCESS` のいずれかで, スレッドのコンテンションスコープを定義します. コンテンションスコープとは, スレッドのスケジューリングの際に, プロセス内のスレッドだけを考慮する (`PTHREAD_SCOPE_SYSTEM`) か, あるいは, 他のプロセス内のスレッドも考慮する (`PTHREAD_SCOPE_LOCAL`) かを指定するもの (らしい) です.

PTL では, コンテンションスコープは `PTHREAD_SCOPE_SYSTEM` 固定で, 変更することはできません.

コンテンションスコープを設定するためには, `pthread_attr_setscope()` 関数を使用します.

3.2.3.5 スタックプロパティ

マルチスレッド環境では, スタックはスレッド毎に必要です. 通常の UNIX 上のプロセスではスタックが溢れた場合, カーネルが自動的にスタックセグメントを拡張します. しかし, スレッドスタックは, カーネルが管理していないため, この処理を PTL 側で行ないます.

PTL では, 大域変数 `int pthread_stack_expansion_np` が非 0 の場合, スレッドスタックの自動拡張および溢れ検出を行うことができます. 高速化のため, `pthread_stack_expansion_np` はデフォルトでは 0 になっています. この変数は, `main()` を実行する前に設定する必要があり, 実行中に変更してはいけません. 以下のように設定するのが正しい方法です.

```
int pthread_stack_expansion_np = 1;

int main()
{
  ...
}
```

PTL では, 現在のところ 3 種類のスタック (共有メモリスタック, Redzone Protect スタック, ヒープメモリスタック) を提供しています. (`pthread_stack_expansion_np` が 0 の場合, ヒープメモリスタックのみが使用されます) それぞれのスタックの特徴は以下の通りです.

共有メモリスタック
: 共有メモリスタックは OS が共有メモリ機構をサポートしている場合に使用可能で, スタック溢れを検出できます. スタックが溢れた場合, 自動的に拡張できる場合があります (アーキテクチャによる). スタックの確保に若干時間がかかるという欠点があります.

Redzone Protect スタック
: Redzone Protect スタックは OS が mprotect システムコールをサポートしている場合に使用可能で, スタック溢れを検出できます (拡張は出来ません). スタックの確保に若干時間がかかるという欠点があります.

ヒープメモリスタック
: ヒープメモリスタックは, どのアーキテクチャでも使用可能で, `malloc()` を用いてスタックを確保します. スタック溢れを検出できませんが, 三つの中で一番速く確保することが出来ます.

低レベルの実装を隠蔽するため, スタックを確保する際に, ユーザはこれらのスタックの種類を陽に指定することは出来ないようになっています. その代わり, ユーザは `pthread_attr_setstackprop_np()` を用いて使用したいスタックの性質 (スタックプロパティ) を伝えることが出来ます. 現在のところプロパティとして以下の 3 つが定義されています. 複数のプロパティを論理和で結合して `pthread_attr_setstackprop_np()` に渡すことが出来ます.

`PTHREAD_STACK_SAFE_NP`
: 安全なスタック (溢れ検出可能)

`PTHREAD_STACK_EXTENSIBLE_NP`
: 自動拡張可能なスタック

`PTHREAD_STACK_NONE_NP`
: 特に性質を指定しない

デフォルトでは, PTHREAD_STACK_SAFE_NP が使用されます. PTL は, なるべく指定したプロパティを満たすスタックを確保しようと試みますが, 失敗した場合は他の種類のスタックが割り当てられます.

これらのスタックは, スレッドが終了した後, スタックキャッシュに蓄えられ, 以後のスレッドの生成時に再利用されます.

また, `pthread_alloc_stack_cache_np()` を用いて指定した数のスタックを予めスタックキャッシュに用意しておくことができます. これによって, 一度に多数のスレッドを生成する際の時間を稼ぐことが出来ます.

`pthread_alloc_stack_cache_np` は PTL 独自の関数です. POSIX ではスタックの確保方法の指定について特に定めていません.

3.2.3.6 スタックサイズ

スタックサイズ 属性は, スレッドのスタックサイズを指定します. デフォルトでは 16K バイトです. 変更するためには `pthread_attr_setstacksize()` 関数を用います.

3.2.3.7 デタッチステート

デタッチステート 属性はスレッドが開始した際にデタッチされているかどうかの指定を行ないます.(第 3.1.4 節 [スレッドの削除], 4 ページ.) デフォルトではデタッチされていません. デタッチステートを変更するためには, `pthread_attr_setdetachstate()` 関数を用います.

3.2.3.8 サスペンドステート

サスペンドステート 属性は, スレッドが開始した際にサスペンドされているかどうかの指定を行ないます. (第 3.1.5 節 [スレッドのサスペンドの概要], 4 ページ.) デフォルトではスレッドは開始時はサスペンドされません. サスペンドステートを変更するためには, `pthread_attr_setsuspended_np()` 関数を用います.

3.2.4 Mutex アトリビュートオブジェクト

Mutex アトリビュートオブジェクト は, `pthread_mutex_init()` や `PTHREAD_MUTEX_INITIALIZER` で生成する Mutex のアトリビュートを指定するために用います.

Mutex アトリビュートオブジェクトで指定できる属性には以下のものがあります.

- プロセスシェアード属性
- Mutex プロトコル属性
- シーリング属性

PTL では, Mutex アトリビュートオブジェクトのデフォルトとして, それぞれの属性に以下の値を与えます. 属性の値の意味に関しては後の章を参照してください.

属性	値
プロセスシェアード属性	0 (not shared)
Mutex プロトコル属性	PTHREAD_PRIO_NONE
シーリング属性	SCHED_FIFO の最大プライオリティ

このデフォルト値は, 以下の場合に使用されます.

- `pthread_mutexattr_init()` で生成したスレッドアトリビュートオブジェクトの初期値
- `pthread_mutex_init()` の引数の Mutex アトリビュートオブジェクトが NULL の場合
- `PTHREAD_MUTEX_INITIALIZER` で作成された Mutex のアトリビュートとして

3.2.4.1 プロセスシェアード属性

プロセスシェアード属性 は, Mutex がプロセス間で共有されるかどうかを指定します. PTL ではこの属性はサポートしていません. 全ての Mutex はプロセスローカルです.

プロセスシェアード属性を変更するためには, `pthread_mutexattr_setpshared()` 関数を用います.

3.2.4.2 Mutex プロトコル属性

プライオリティの逆転 (第 3.3.1 節 [Mutex], 13 ページ.) を避けるため, Mutex をロックするスレッドのプライオリティを動的に変更することが出来ます. どのような方法で, プライオリティを変更するかを, *Mutex* プロトコル属性によって指定します. Mutex プロトコルには以下の 3 種類があります.

- PTHREAD_PRIO_NONE (デフォルト)
- PTHREAD_PRIO_INHERIT
- PTHREAD_PRIO_PROTECT (未実装)

Mutex プロトコル属性を変更するためには, `pthread_mutexattr_setprotocol()` 関数を用います.

詳しくは 第 3.3.1.1 節 [プライオリティの逆転の回避], 14 ページ. を参照してください.

3.2.4.3 シーリング属性

シーリング属性 は, Mutex プロトコル属性 (第 3.2.4.2 節 [Mutex Protocol Attribute], 12 ページ.) が PTHREAD_PRIO_PROTECT の場合に意味を持ちます. (PTL では, まだ PTHREAD_PRIO_PROTECT プロトコルを実装していません)

シーリング属性を変更するためには, `pthread_mutexattr_setprioceiling()` 関数を用います.

シーリングについては, 第 3.3.1.1 節 [プライオリティの逆転の回避], 14 ページ. を参照してください.

3.2.5 Condition アトリビュートオブジェクト

Condition アトリビュートオブジェクト は, `pthread_cond_init()` や `PTHREAD_COND_INITIALIZER` で生成する Condition Variable のアトリビュートを指定するために用います.

Condition アトリビュートオブジェクトで指定できる属性には以下のものがあります.

- プロセスシェアード属性

PTL では, Condition アトリビュートオブジェクトのデフォルトとして, それぞれの属性に以下の値を与えます. 属性の値の意味に関しては後の章を参照してください.

属性	値

プロセスシェアード属性
    0 (not shared)

このデフォルト値は, 以下の場合に使用されます.

- `pthread_condattr_init()` で生成したスレッドアトリビュートオブジェクトの初期値
- `pthread_cond_init()` の引数の Condition アトリビュートオブジェクトが NULL の場合
- `PTHREAD_COND_INITIALIZER` で作成された Condition Variable のアトリビュートとして

3.2.5.1 Condition Variable プロセスシェアード属性

プロセスシェアード 属性は, Condition Variable がプロセス間で共有されるかどうかを指定します. PTL ではこの属性はサポートしていません. 全ての Condition Variable はプロセスローカルです.

プロセスシェアード属性を変更するためには, `pthread_condattr_setpshared()` 関数を用います.

3.3 同期機構の概要

マルチスレッドプログラムでは, しばしばスレッド間で同期を取る必要があります. 例えば, スレッド間で共有している大域データにアクセスする場合や, 他のスレッドで行なわれている作業が完了するまで待つような場合です. Pthreads では, 同期の方法として, Mutex と Condition Variable を提供しています.

3.3.1 Mutex

Mutex は, *MUTual EXclusion* の略で, 複数のスレッドが同時にスレッド間で共有しているリソース (大域データ等) にアクセスできないようにするためのものです. Mutex には二つの状態 — ロックされている状態と, アンロックされている状態 — があります. スレッドから共有リソースにアクセスするためには以下のような手順を踏みます.

1. 共有リソースと結び付けられた Mutex をロック.
2. 共有リソースにアクセス.
3. ロックした Mutex をアンロック.

1. で Mutex をロックする際に, その Mutex が既に他のスレッドによってロックされていた場合には, 後からロックしようとしたスレッドの実行は Mutex がアンロックされるまで実行を停止します.

Mutex を用いて共有リソースを保護するのはプログラマの責任です.

Mutex は排他的にアクセスしたい共有リソース毎に Mutex を生成しなければなりません.

Mutex は, 「短い時間の」アクセスに限って使用されるべきです. 特に, Mutex をロックしている間にそのスレッドがブロックする可能性があるような場合, Mutex を使用すべきではありません. このような場合は, Condition Variable を使用します. 第 3.3.2 節 [Condition Variable], 15 ページ.

スレッドが終了する場合 (第 3.1.2 節 [スレッドの終了], 3 ページ.), ロックしている Mutex は自動的には開放されません. PTL では, 終了した際にロックしている Mutex があると警告を発します. (第 3.7 節 [キャンセルの概要], 22 ページ.)

低いプライオリティのスレッドが Mutex をロックしている間に, 他の高いプライオリティのスレッドが同一の Mutex をロックしようとすると, 高いプライオリティのスレッドが長期間ブロックされ得ます (プライオリティの逆転). これを避けるための機構が, Mutex には備わっています. 以下を参照してください.

3.3.1.1 プライオリティの逆転の回避

プライオリティの逆転を避けるため, Mutex には以下の 3 種類の 「プロトコル」が定義されています.

`PTHREAD_PRIO_NONE`

スレッドが PTHREAD_PRIO_NONE プロトコルを持つ Mutex をロックする際は, スレッドのプライオリティは Mutex の所有によって影響されることはありません (デフォルト). このプロトコルは, プライオリティの逆転を回避できません.

`PTHREAD_PRIO_INHERIT`

スレッドが PTHREAD_PRIO_INHERIT プロトコルを持つ Mutex をロックした後, より高いプライオリティのスレッドが同一の Mutex をロックしようとしてブロックした場合, Mutex をロックしているスレッドのプライオリティは, その Mutex を待ってブロックしている全てのスレッドの中で, 最も高いプライオリティのスレッドのプライオリティまで一時的に高められます. これによって, Mutex をロックしているスレッドが高いプライオリティで実行されることになります.

スレッドが Mutex をアンロックすると, プライオリティは元に戻ります.

`PTHREAD_PRIO_PROTECT`

スレッドが PTHREAD_PRIO_PROTECT プロトコルを持つ Mutex をロックすると, 直ちにそのスレッドのプライオリティは, その Mutex の シーリング の値まで高められます. スレッドが複数の PTHREAD_PRIO_PROTECT プロトコルを持つ Mutex をロックしている場合は, それらのうちの最も高いシーリングをプライオリティとします.

スレッドが Mutex をアンロックすると, プライオリティは元に戻ります.

プライオリティの逆転を防ぐためには, Mutex のシーリングは, Mutex をロックする可能性の有る全てのスレッドの最も高いプライオリティと等しいか, それ以上に設定されなければなりません.

シーリングは SCHED_FIFO スケジューリングポリシーで許されるプライオリティの範囲の値を取ります.

もし, スレッドが異なったプロトコルを持った複数の Mutex を同時に所有する場合, スレッドはそれらのプロトコルによって得られる最高のプライオリティで実行されます.

PTL では現在のところ PTHREAD_PRIO_PROTECT は実装していません.

3.3.1.2 **Mutex** の生成と破棄

Mutex を生成するためには `pthread_mutex_init()` か, `PTHREAD_MUTEX_INITIALIZER` を用います.

破棄するためには `pthread_mutex_destroy()` 関数を用います.

3.3.1.3 Mutex のロック

Mutex をロックするためには, `pthread_mutex_lock()` を用います. この関数は, Mutex がロックされていなければロックします. Mutex がロックされていれば, Mutex がアンロックされるまでブロックします. いずれにせよ, `pthread_mutex_lock()` からリターンしてきた際には Mutex はロックされています.

Mutex がロックされている場合にブロックして欲しくない場合のため, `pthread_mutex_trylock()` があります. これは, Mutex がロックされていた場合, EBUSY を返します.

Mutex をロックしているスレッドが更に同じスレッドをロックしようとすると, `pthread_mutex_lock()`, `pthread_mutex_unlock()` は EDEADLK とを返します.

3.3.1.4 Mutex のアンロック

Mutex をアンロックするためには `pthread_mutex_unlock()` 関数を用います.

Mutex をアンロックし忘れると, 簡単にデッドロックを引き起こすので注意してください. 特にキャンセル可能なスレッドでは, このことは重要です. 詳しくは 第 3.7 節 [キャンセルの概要], 22 ページ. を参照してください.

3.3.1.5 Mutex に対するネーミング

Mutex にはそれぞれ名前をつけることができます. これは, デバッグの時に役立ちます. Mutex を生成した時点では, 名前は付いていません.

Mutex に名前を付けるためには, `ptheread_mutex_setname_np()` を, 名前を得るためには, `pthread_mutex_getname_np()` を用います.

3.3.2 Condition Variable

Condition Variable は, 共有リソースが, 「特定の状態」になることを待つための仕組みです. Condition Variable は以下のように働きます.

スレッドは, 共有リソースが「自分の望む状態」になっているかどうかをチェックし, なっていなければ, ブロックします. 共有リソースの状態を変更したスレッドは, 状態を変更したことを他のスレッドに知らせます. 変更を通知された (ブロックしている) スレッドはアンブロックされ, またリソースが「自分の望む状態」かどうかを調べます.

例えば, 一つのスレッド (writer thread) が共有バッファに対して「要求」を書き込み, もう一つのスレッド (reader thread) が共有バッファから「要求」を読み取って処理を行なうような場合を考えます. バッファには数個の「要求」を格納することができるとします. この場合, writer thread は共有バッファを満たしてしまった場合, バッファに余裕ができるまで「バッファに空きがある」という Condition Variable でブロックします. 一方, reader thread はバッファから「要求」を読み取った後, バッファに空きが生じたので,「バッファに空きがある」という Condition Variable を「シグナル」します.

スレッドが共有リソース上のテスト (「自分の望む状態」かどうか ?) を行なうコードは, 共有リソースへのアクセスを行なわなければならないため, 共有リソースと結び付いた Mutex で保護されなければなりません. (第 3.3.1 節 [Mutex], 13 ページ.). 実際, Condition Variable は, Mutex と必ず結び付いて使用されます.

Condition Variable でブロックする際には, 他のスレッドが共有リソースをアクセスできるように Mutex はアンロックされ, また, Condition Variable からアンブロックする際に Mutex が自動的に再びロックされます. Condition Variable でのブロックと, Mutex のアンロックは不可分に実行されます.

典型的な疑似コードを以下に示します. スレッド A は, 共有リソース上の特定の<条件>が満たされた場合に処理 X を行ないます. スレッド B は共有リソースにアクセスするもう一つのスレッドで, 処理 Y を行ないます.

スレッド A のコード

```
/* 条件をテストするため, Mutex をロック */
pthread_mutex_lock(共有リソース_mutex);
while (<条件>が満たされていない) {
        /*
         * <条件>が変更されるまで, Condition Variable でブロック.
         *  Mutex は自動的にアンロックされる.
         */
        pthread_cond_wait(条件_cond , 共有リソース_mutex);
}
共有リソースに対する処理 X;
pthread_mutex_unlock(共有リソース_mutex);
```

スレッド B のコード

```
pthread_mutex_lock(共有リソース_mutex);
共有リソースに対する処理 Y;
/*
 * 変更を通知する.
 */
pthread_cond_signal(条件_cond);
pthread_mutex_unlock(共有リソース_mutex);
```

スレッド A は, 共有リソースにアクセスするために Mutex をロックし, <条件>をテストします. もし<条件>が満たされていなければ, Condition Variable で Wait します. その際に, Mutex は自動的にアンロックされます. 一方スレッド B では, 共有リソースにアクセスするために Mutex をロックし, 処理 Y を行ないます. その後, スレッド A が共有リソース上の Condition Variable で Wait している場合に備え, `pthread_cond_signal()` によって Condition Variable を「シグナル」します. これによって, スレッド A は `pthread_cond_wait()` の Wait からアンブロックされ, Mutex を再びロックして, <条件>をテストすることができます. スレッド B が `pthread_cond_signal()` を呼びだす際に, 他のスレッドが Condition Variable で Wait していなかった場合, 何も発生しません.

3.3.2.1 Condition Variable の生成と破棄

Condition Variable を生成するためには, `pthread_cond_init()` か, `PTHREAD_COND_INITIALIZER` を用います.

破棄するためには `pthread_cond_destroy()` を用います.

3.3.2.2 Condition Variable での Wait

Condition Variable で Wait するためには, `pthread_cond_wait()` を用います. この関数は, 自動的に指定された Mutex をアンロックし, Condition Variable が「シグナル」されるまでブロックします. (第 3.3.2.3 節 [Signaling Condition Variable], 17 ページ.) Mutex は必ずロックされていなければなりません. ロックされていない場合の動作は未定義です.

複数のスレッドから同一の Condition Variable で Wait を行なう際, `pthread_cond_wait()` には, 同一の Mutex が指定されなければなりません.

`pthread_cond_wait()` は, Condition Variable が「シグナル」されるまでブロックします. (第 3.3.2.3 節 [Condition Variable のシグナル], 17 ページ.)

指定した時刻が過ぎたら, Condition Variable での Wait から強制的に抜け出したい場合は, `pthread_cond_timedwait()` を用います. ここで指定する時刻は, 相対時刻 (何秒後) ではなく, 絶対時刻です. `pthread_cond_timedwait()` は, 時刻が過ぎると ETIMEDOUT を返します.

Condition Variable で Wait 中のスレッドにシグナルが配送された場合, シグナルハンドラを呼び出す前に Mutex がロックされます. ロックされた Mutex は, シグナルハンドラからのリターンの後に自動的にアンロックされます (シグナルハンドラから大域ジャンプを行なった場合を除く).

3.3.2.3 Condition Variable のシグナル

Condition Variable で Wait しているスレッドをアンブロックさせるためには, 2 種類の方法があります.

- Condition Variable で Wait しているスレッドの一つをアンブロックさせるためには `pthread_cond_signal()`.
- Condition Variable で Wait している全てのスレッドをアンブロックさせるためには `pthread_cond_broadcast()`.

`pthread_cond_signal()` では, アンブロックされるスレッドは, Condition Variable でブロックしているスレッドのうち, 最もプライオリティの高いスレッドです. (第 3.5 節 [スケジューリングの概要], 18 ページ.)

`pthread_cond_broadcast()` では, Condition Variable を待っている全てのスレッドがアンブロックされますが, 当然 Mutex をロックできるスレッドは (同時には) 一つだけです. 他のスレッドは Condition Variable の Wait からは開放されますが, Mutex を待ってブロックすることになります.

3.3.3 その他の同期機構

スレッドは, 他のスレッドの終了を待つことができます. 第 3.1.3 節 [スレッドの終了の Wait], 4 ページ. を参照してください.

3.4 Thread-Specific データの概要

特定の キー に対し, スレッド毎にデータを持つことが出来ます. 任意のポインタをスレッドに結び付けることが出来ます. 概念的には, Thread-Specific データは, 縦に キー, 横にスレッドが並んだ二次元配列と考えることが出来ます.

それぞれのスレッドは, 任意のデータを任意の値を自分自身の Thread-Specific データ領域に格納し, 取得することができます.

Thread-Specific データを用いるためには, まず, 全てのスレッドで共有する, ユニークな キー を生成しなければなりません. このためには, `pthread_key_create()` を用います. この関数は, キー を生成し, 存在する全てのスレッドの, キー と結び付いた Thread-Specific データの値を 0 に初期化します. また, この関数では, キー に対するデストラクタ関数を定義できます. この関数は, スレッドが終了する際に, Thread-Specific データを開放するために用いられます. (第 3.1.2 節 [スレッドの終了], 3 ページ.)

キーを削除するには, `pthread_key_delete()` を用います.

Thread-Specific データを格納するためには, `pthread_setspecific()` を用います.

Thread-Specific データを取得するためには, `pthread_getspecific()` を用います.

3.5 スケジューリングの概要

スレッドは, それぞれの Scheduling Policy Attribute(第 3.2.3.1 節 [Scheduling Policy Attribute], 8 ページ.) と Scheduling Priority Attribute(第 3.2.3.2 節 [スケジューリングプライオリティ], 8 ページ.) によってスケジュールされます.

スケジューリングポリシー は, スレッドがどのようにスケジューリングされるかを指定します.

スケジューリングポリシーとプライオリティの値の範囲の関係は以下のようになっています.

```
                                     プライオリティ大
SCHED_FIFO, SCHED_RR の範囲                  ^
                                              |
SCHED_OTHER の範囲                           |
                                              v
                                     プライオリティ小
```

スケジューリングは, 以下の規則によって行なわれます.

- スケジューリングポリシーとして SCHED_FIFO, SCHED_RR を持つスレッドが存在すれば, その中で最も高いプライオリティを持つスレッドを実行します. (同一のプライオリティで SCHED_FIFO と SCHED_RR の両方のスレッドが存在すれば SCHED_FIFO を優先します).
- SCHED_FIFO, SCHED_RR を持つスレッドが存在しなければ, SCHED_OTHER を持つスレッドを実行します. (つまりスケジューリングポリシーとして, SCHED_FIFO あるいは SCHED_RR を持つ走行可能なスレッドが存在すれば, SCHED_OTHER ポリシーを持つスレッドはスケジュールされません.) それぞれプライオリティ a, b を持つ二つの SCHED_OTHER のスレッド A, B が存在した場合, スレッド A は, スレッド B よりも平均的に a/b 倍の頻度で実行されます.

3.5.1 SCHED_FIFO

このポリシーを持つスレッドは CPU を横取り (preempt) されることはありません. `SCHED_FIFO` のスレッドは, 以下の場合に CPU を明け渡し, (存在すれば) 他のスレッドを実行します.

- Mutex や, Condition Variable 等でブロックした.
- `sleep()` 等の呼びだしでサスペンドした.
- `sched_yield()` を呼びだした.
- `pthread_create()` によって, 自分よりプライオリティの高いスレッドを生成した.

3.5.2 SCHED_RR

このポリシーは `SCHED_FIFO` と似ていますが, 同一のプライオリティに複数のスレッドが存在した場合, それらのスレッドの間で CPU の横取り (preemption) が行なわれます. つまり, タイムスライスを消費すると強制的にコンテキストスイッチが行なわれます.

3.5.3 SCHED_OTHER

このスケジューリングは, POSIX1003.1c では処理系依存とされています. PTL では, プライオリティに応じて CPU を与えられるスケジューリングに `SCHED_OTHER` を割り当てています. このポリシーの下では, スレッドにはプライオリティに比例して CPU 時間が与えられます. PTL では, このポリシーがデフォルトです.

`SCHED_RR`, `SCHED_OTHER` スケジューリングのためのタイムスライスは, 100msec です.

3.6 シグナルの概要

それぞれのスレッド毎に シグナルマスク が存在します. スレッドのシグナルマスクを取得, 変更するためには, POSIX.1 のインタフェースがそのまま用いられます. シグナルマスクはシグナルの集合で, マスクされたシグナルは, スレッドへの配送がブロックされます.

プロセスは, シグナル毎に, シグナルアクション を保持します. アクションはプロセス中の全てのスレッドで共有されます. シグナルアクションを取得, 変更するためには, POSIX.1 のインタフェースがそのまま用いられます.

シグナルの配送時に, シグナルアクションが, 終了 (termination), 停止 (stop), 続行 (continue) ならば, プロセス全体が終了, 停止, 続行します.

セグメンテーション違反のような, 特定のスレッドに起因するシグナルは, 同期シグナル と呼ばれます. 同期シグナルはそのシグナルを引き起こしたスレッドに配送されます.

一方, `kill()` や端末からのシグナル (`CONTROL-C` による SIGINT 等) は, 非同期シグナル と呼ばれます. 非同期シグナルを待つためには, `sigwait()` を使用します. 詳しくは, 第 3.6.1 節 [シグナルの配送], 20 ページ. を参照して下さい.

スレッドは, `pthread_kill()` によって, 同一プロセス内の特定のスレッドにシグナルを送ることが出来ます. もし, 受信したスレッドがそのシグナルの配送をブロックしていた場合, シグナルはそのスレッドでペンディングされます.

3.6.1 シグナルの配送

シグナルアクションは, プロセス毎に管理されます. アクションは, プロセス中の全てのスレッドで共有されます. シグナルが配送された時, そのシグナルに対応するアクションが振る舞いを決定します.

以下のシグナルは, 特別に取り扱われます.

- SIGKILL が配送されると, Cleanup ハンドラを実行することなくプロセス中の全てのスレッドはただちに消滅し, プロセスは終了します.
- SIGSTOP が配送されると, プロセス中の全てのスレッドは直ちに停止します.
- SIGCONT が配送されると, プロセス中の全てのスレッドの実行は続行されます.

シグナルが配送された場合, シグナルアクションが参照されます. シグナルアクションは 4 種類あり, `sigaction()`, `signal()` 関数で設定, 変更できます.

`SIG_DFL`	デフォルト動作. OS のデフォルトのシグナルアクションが行なわれます. これが全てのシグナルアクションのデフォルトです.
`SIG_IGN`	無視. シグナルは破棄されます.
`SIG_SIGWAIT_NP`	シグナルは, `sigwait()` へ渡されます. どのスレッドも `sigwait()` を実行中でない場合, シグナルはペンディングされ, スレッドが `sigwait()` を実行した際に渡されます. これは PTL の独自拡張です.
<シグナル捕捉関数>	配送されたシグナルに対して, `sigwait()` を実行しているスレッドが存在すれば, シグナルは `sigwait()` に伝えられます. そのシグナルに対して `sigwait()` を実行しているスレッドが存在しなければ, シグナルは プロセス中のシグナルをブロックしていないスレッドの一つ に配送され, シグナル捕捉関数の実行が開始されます. もし, プロセス中の全てのスレッドがシグナルをブロックしていた場合, どれかのスレッドがシグナルの配送をアンブロックするか, 対応するシグナルアクションを無視 (SIG_IGN) に設定するまで, シグナルはプロセスでペンディングされます. シグナル捕捉関数がリターンしたら, スレッドは割り込まれた場所から実行を再開します. シグナル捕捉関数を実行するスレッドは無作為に選ばれます.

複数のスレッドが同一のシグナルに対して `sigwait()` を実行していた場合, 単一のスレッドの `sigwait()` のみがリターンします. その際, `sigwait()` がリターンするスレッドは無作為に選ばれます.

3.6.1.1 スレッドへ向けたシグナル

以下のようなシグナルは, スレッドへ直接送られます.

- セグメンテーション違反のような, スレッドの活動に起因するシグナルは, シグナルを引き起こしたスレッドへ送られます.
- `pthread_kill()` によって特定のスレッドに送られたシグナルは, 指定されたスレッドへ送られます.
- `alarm()` の結果発生する SIGALRM シグナルは, アラームを要求したスレッドへ送られます.

シグナルを受信したスレッドが, シグナルの配送をブロックしていた場合, スレッドがシグナルの配送をアンブロックするか, 対応するアクションを無視 (SIG_IGN) に設定するまで, シグナルはスレッドでペンディングされます. スレッドでペンディングされているシグナルが再度発生した場合, 後のシグナルは破棄されます.

一度スレッドでシグナルがペンディングされると, それは他のスレッドに配送されることはありません.

3.6.1.2 プロセスへ向けたシグナル

プロセス中の全てのスレッドがシグナルをブロックしていた場合, スレッドがシグナルの配送をアンブロックするか, 対応するシグナルアクションを無視 (SIG_IGN) に設定するまで, シグナルはプロセスでペンディングされます. プロセスでペンディングされているシグナルが再度発生した場合, 後のシグナルは破棄されます.

3.6.2 シグナルの状態の継承

`pthread_create()` によって新たなスレッドが生成される際, シグナルの状態は以下のように `pthread_create()` を呼び出したスレッドから継承されます.

1. シグナルブロックマスクは, 作成するスレッドから継承されます.
2. シグナルペンディング集合はクリアされます.

3.6.3 同期シグナルリスト

PTL では, 以下のシグナルを同期シグナルとして扱います.

> SIGILL, SIGFPE, SIGBUS, SIGSEGV, SIGSYS, SIGPIPE, SIGEMT

(SIGPIPE を同期シグナルとして扱うのは問題があるかも知れません)

これら以外のシグナルはすべて非同期シグナルとして扱われます.

3.6.4 Async Safe 関数

Not yet written for PTL2.

3.6.5 内部で使用しているシグナル

以下のシグナルはライブラリ内部で使用しているため, ユーザは使用できません. kill(1) 等によってプロセスにシグナルが送られた場合の動作は不定となります.

SIGUSR2, SIGSEGV, SIGILL, SIGALRM

ただし, SIGALRM は, `pthread_alarm_np()` によって疑似的に発生させることが出来ます.

3.6.6 シグナルハンドラ

スレッドのシグナルハンドラは, 以下のような引数を受け取ります.

```
struct siginfo {
    int si_signo;       /* signal number */
    int si_code;        /* code (machine dependent) */
};

handler(int sig, struct siginfo *info) {
   ...
}
```

`siginfo` 構造体の, `si_signo` メンバーは, シグナルの番号を保持しています. `si_code` メンバーは, アーキテクチャ依存の数が入ります. これは, BSD UNIX のシグナルハンドラの第 2 引数の値です. `pthread_kill()`, `raise()` 等によって起動されたシグナルハンドラでは, `si_code` は 0 になります.

3.6.7 errno

PTL では, 大域変数 `errno` は, スレッド毎に保持され, コンテキストスイッチのたびに待避, 復元されます. シグナルハンドラを実行する際も, `errno` は待避, 復元されるので, ユーザはシグナルハンドラの内部で `errno` を保存する必要はありません.

3.7 キャンセルの概要

キャンセル は, 実行中のスレッドを他のスレッドから中止させるための機構です. スレッドキャンセル機構によって, コントロールされた方法でプロセス内の他のスレッドの実行を終了させることができます. スレッドは他のスレッドからのキャンセルの要求を一時的に保留したり, キャンセルの際に Cleanup ハンドラを実行することが出来ます.

それぞれのスレッド毎に 2 ビットで表される *Cancelability State* が存在します. 1 ビットは, スレッドがキャンセル要求を受け付ける (デフォルト) か否かを表します. 残りの 1 ビットは, スレッドが同期キャンセルモード (デフォルト) か, 非同期キャンセルモードかを表します. (第 3.7.1 節 [Cancelability States], 23 ページ.)

スレッドにキャンセル要求を行なうためには, `ptread_cancel()` を用います. この関数を呼び出しても, 対象のスレッドが直ちにキャンセルされるわけではありません. また, 一度キャンセル要求を行なった場合, 取り消すことは出来ません.

キャンセルが実行されるためには, 対象のスレッドがキャンセル要求を受け付けるモードであることが必要で, かつ, 対象のスレッドが割り込みポイント (第 3.7.2 節 [キャンセルポイント], 24 ページ.) を通るか, 対象のスレッドが非同期キャンセルモードの時にのみ発生します. キャンセル要求のなされたスレッドがこれらの条件を満たせば, スレッドは次にスケジューリングされた時にキャンセルされます.

開放されなければならないリソース (Mutex 等) を保持している間に非同期キャンセルモードを用いることは, キャンセルによってリソースの開放が出来なくなる可能性があるため, 非常に危険です. また, Cleanup スタックはスレッドが同期キャンセルモードか, キャンセルを禁止した状態の時のみ, 安全に操作 (プッシュ, ポップ) することができます. (第 3.7.3 節 [スレッド Cleanup], 24 ページ.) 非同期キャンセルモードは, 長時間かかる計算のループなどを中止するためだけに用いるべきです.

3.7.1 Cancelability States

Cancelability State は, スレッドがキャンセルの要求を受け取った際の動作を決定します.

それぞれのスレッドは, 2 ビットの *Cancelability States* を持っています.

キャンセル許可フラグ

キャンセル許可フラグは, キャンセル要求を受け付ける (PTHREAD_CANCEL_ENABLE, デフォルト) か否か (PTHREAD_CANCEL_DISABLE) を制御します. キャンセル許可モードを変更するためには `pthread_setcancelstate()` を用います.

キャンセルは, 対象のスレッドが PTHREAD_CANCEL_ENABLE の場合にのみ発生します. PTHREAD_CANCEL_DISABLE の場合, キャンセル要求はスレッドでペンディングされ, 後にキャンセル許可モードが PTHREAD_CANCEL_ENABLE に変更されたときにキャンセルが実行されます.

キャンセルタイプ

キャンセルタイプは, スレッドが特定の操作を実行中にのみキャンセルされる (`PTHREAD_CANCEL_DEFERRED`) –同期キャンセルモード (デフォルト)– か, 「何を実行していても」キャンセルされる (`PTHREAD_CANCEL_ASYNCHRONOUS`) – 非同期キャンセルモード– かを制御します.

キャンセル許可フラグが `PTHREAD_CANCEL_ENABLE` で, キャンセルタイプが `PTHREAD_CANCEL_DEFERRED` ならば, キャンセル要求は, スレッドの実行がキャンセルポイントに達するまでペンディングされます. (第 3.7.2 節 [Cancellation Point], 24 ページ.)

キャンセル許可フラグが `PTHREAD_CANCEL_ENABLE` で, キャンセルタイプが `PTHREAD_CANCEL_ASYNCHRONOUS` ならば, 新たな, あるいはペンディングされていたキャンセル要求は直ちに処理されます.

キャンセル許可フラグが `PTHREAD_CANCEL_DISABLE` ならば, 全てのキャンセル要求はペンディングされるのでキャンセルタイプの設定は直ちには効果はありませんが, キャンセル許可フラグが ENABLE になると, 設定されたキャンセルタイプは効果を持ちます.

新たに生成されるスレッドと, 初期スレッドのキャンセル許可フラグとキャンセルタイプの初期値はそれぞれ `PTHREAD_CANCEL_ENABLE`, `PTHREAD_CANCEL_DEFERRED` です.

3.7.2 キャンセルポイント

以下はキャンセルポイントとなります.

- `pthread_cond_wait()`, `pthread_cond_timedwait()`).
- `pthread_join()`.
- `pthread_testcancel()`.
- `sigwait()`.
- `sleep()`, `nanosleep()`, `usleep()`.

スレッドのキャンセル許可モードが PTHREAD_CANCEL_ENABLE で, スレッドがこれらの関数を呼び出してブロックしている間に, 他のスレッドからキャンセル要求がなされると, スレッドはキャンセルされます.

明示的にキャンセルの要求があったかどうかをテストするためには, `pthread_testcancel()` を用います. スレッドがキャンセル許可モード ENABLE で, そのスレッドにキャンセル要求がなされている時に, スレッドが `pthread_testcancel()` を実行すると, キャンセルが実行されます. `pthread_testcancel()` からはリターンしません.

キャンセルポイントで待っているイベントが起こった後, スレッドがスケジューリングされる前に, キャンセル要求が発生した場合, キャンセルが実行されるか, あるいはキャンセル要求がペンディングされて, スレッドが通常の実行を開始するかどうかは, 不定です.

3.7.3 スレッド Cleanup

スレッドがキャンセルされる際に, 後処理をするルーチン (*Cleanup* ハンドラ) を設定できます. スレッドは Cleanup ハンドラ中でリソースを自動的に開放したりすることができます.

スレッドは Cleanup ルーチンのリストを保持しています. ルーチンを登録するためには, `pthread_cleanup_push()`, `pthread_cleanup_push_f_np()` を, ルーチンを開放するためには, `pthread_cleanup_pop()`, `pthread_cleanup_pop_f_np()` を用います.

キャンセルが実行される際, リスト上のルーチンは LIFO (Last In–First Out) の順に, 一つずつ実行されます. すなわち, 最後にリストにプッシュされた関数が最初に実行されます. Cleanup ルーチンを実行するスレッドは最後の Cleanup ルーチンがリターンするまでキャンセル許可モードは DISABLE になります. 最後の Cleanup ルーチンがリターンすると, スレッドの実行は終了し, 終了ステータスとして PTHREAD_CANCEL が join しているスレッドに返されます.

Cleanup ルーチンはスレッドが `pthread_exit()` を呼びだした場合にも実行されます.

Condition Variable での Wait 中のキャンセルでは副作用として, 最初の Cleanup ルーチンが呼ばれる前に, Mutex が再び獲得されます. さらに, Condition Wait 中にキャンセル要求のあったスレッドは Condition で Wait しているとは見なされず, スレッドは `pthread_cond_signal()` の対象とはなりません.

3.7.4 非同期 Cancel-Safe 関数

非同期キャンセルモードのスレッドが安全に呼び出すことが出来る関数は *Async-Cancel-Safe* 関数 といいます. また, Async-Cancel-Safe 関数は, 割り込まれた時にも呼ぶことが出来ます. Async-Cancel-Safe 関数は, 大域データを書き換えたりすることが無く, 関数の実行が途中で中止されても安全な関数です.

3.8 ログ機能について

(この機能は現在メンテナンスされていません. 有効にするためには, PTL の'`src/spec.h`' で `#define LOGGING` する必要があります)

マルチスレッドのプログラムをデバッグすることは, 簡単ではありません. デバッグを助けるため, PTL にはスレッドの振る舞いを記録し, ログファイルに書き出す機能があります. 出力されたログファイルは, 専用のログビューアを用いて X Window 上で見ることができます.

ログビューアは, '`contrib`' ディレクトリ中の'`xptllogXXX.tar.gz`' にあります. 詳細は, アーカイブ中のドキュメントを参照してください.

環境変数 `PTHREAD_LOG` に, ログファイル名をセットして, プログラムを走らせると, 環境変数で指定されたファイルにログが出力されます. 環境変数 `PTHREAD_LOG` が存在しなければ, ログは出力されません.

ログファイルはホストのバイトオーダーに依存しないようになっています.

ログには, 以下の情報が記録されます.

- スレッドの生成 (create), 終了 (exit), デタッチ, join
- コンテキストスイッチ
- Mutex の init, destroy, lock, unlock, trylock
- Condition Variable の wait, timedwait, signal, broadcast
- キャンセルの要求, Cleanup ハンドラの実行
- シグナルの発生, ハンドラの起動/復帰, sigwait の復帰

スレッド, Mutex, Condition Variable に名前を付けるとログビューアで表示されるため, デバッグを効率的に行なうことが出来ます.

ユーザが任意の ASCII 文字列をログに出力することも可能です. これは従来の printf デバッグに対応するものと考えて良いでしょう. このためには, 関数 `pthread_log_np` を用います. ここで指定された文字列はログビューアの画面上で表示されます.

ログはある程度たまる毎にファイルに書き出されますが, この処理の間, 他のスレッドの実行は停止されます. ログファイルの書きだしには比較的時間がかかるため, ログをファイルに書き出す直前に, PTL は文字列 `[log flushing]` を書き出して, 書き出し中であることをユーザが認識できるようにしています.

3.9 データ型

Pthreads では, 以下のデータ型を定義しています. これらはすべて ‘`<sys/types.h>`’ で定義されます.

`pthread_t`	スレッド
`pthread_attr_t`	スレッドアトリビュートオブジェクト
`pthread_mutex_t`	Mutex
`pthread_mutexattr_t`	Mutex アトリビュートオブジェクト
`pthread_cond_t`	Condition Variable
`pthread_condattr_t`	Condition アトリビュート
`pthread_key_t`	Thread-Specific データキー
`pthread_once_t`	動的なパッケージの初期化

PTL では, `pthread_mutex_t` と `pthread_cond_t` を除く全ての型は int と同一で, 値をハンドルとして用いています. このため, エラーチェックが可能となっています. `pthread_mutex_t` と `pthread_cond_t` は構造体です. これは高速化のためです.

3.10 関数の戻り値

特に断わりの無い限り, Pthread の関数 (pthread_*) は正常に終了した場合 0 を返し, 失敗した場合エラーコードを返します. errno には設定しませんので注意してください.

3.11 注意点

ここでは, マルチスレッド環境で注意すべき点について述べます.

3.11.1 大域変数の保護

CPU がタイムスライスによって奪われるスケジューリング (`SCHED_RR`, `SCHED_OTHER`) の下では, 複数のスレッドから同時にアクセスされる大域変数や関数内 static 変数は, 排他制御されなければなりません (第 3.3 節 [同期機構の概要], 13 ページ.). これに対し, auto 変数は通常, 排他制御する必要は有りません.

3.11.2 デッドロック

マルチスレッド環境では, 不注意によって簡単にデッドロックが起きます. 例えば以下のようなコードはデッドロックを引き起こします. (Thread1 が mutex1 をロックした後, Thread2 が mutex2 をロックすると, Thread2 は mutex1 をロックできずにブロックし, Thread1 も mutex2 をロックできずにブロックしてしまいます.)

```
Thread.1                                Thread.2
while(1) {                                while(1) {
  pthread_mutex_lock(&mutex1);             pthread_mutex_lock(&mutex2);
  ....                                       ....
  pthread_mutex_lock(&mutex2);             pthread_mutex_lock(&mutex1);
  ....                                       ....
  pthread_mutex_unlock(&mutex2);           pthread_mutex_unlock(&mutex1);
  pthread_mutex_unlock(&mutex1);           pthread_mutex_unlock(&mutex2);
}                                         }
```

複数の Mutex をロックする場合, デッドロックを避けるために, プロセス内で Mutex をロックする順序を統一しなければなりません.

3.11.3 既存のライブラリの使用

既存のライブラリのほとんどは, マルチスレッド環境から呼ぶには適していません. それらのほとんどは, (適切なロック無しに) 大域データにアクセスしています.

これらのライブラリを使用するためには, いくつかの方法があります.

- ライブラリを単一のスレッドからしか呼ばない. こうすると, ライブラリは並列に呼ばれることはありません.
- 排他制御する. ライブラリを呼ぶ前に Mutex をロックし, リターンしてきた後でアンロックすることによって, ライブラリの呼び出しを, 「逐次化」することができます.

PTL では, いくつかの既存のライブラリ関数のリエントラント版を用意しています. 第 4.9 節 [リエントラント関数], 51 ページ. を参照してください.

PTL の, スレッド対応標準入出力ライブラリ (stdio) は, ファイルデスクリプタ毎に用意した Mutex をロックすることによって実現しています.

3.11.4 スレッドスタック

スレッドのスタックは, プロセス中のデータ領域あるいは共有メモリ領域に確保されます.

共有メモリスタック及び Redzone プロテクトスタックは, 溢れると

```
[Thread 6]Stack Type: shared memory
Stack overflow (cannot extend) stack size 16384, bottom 0x20400000
```

のように表示して停止します. これに参考に, スタックのサイズを変更して下さい.

3.11.5 入出力

I/O システムコールは, ほとんどがライブラリによってオーバーライドされています.

それぞれのスレッドは ネットワーク I/O やプロセス間通信を並行して実行できます. I/O 操作がすぐに完了しない場合, その I/O を実行しようとしたスレッドのみがブロックされ, 他のスレッドの実行は続行されます. (ただし, ファイル I/O は, ファイル I/O が完了するまで全てのスレッドをブロックさせます.)

これらは, PTL 内部でファイルデスクリプタを非ブロックモードで使用することによって実現しています. 現在のところ, ユーザは, 非ブロックモードを使用することは出来ません.

ただし, 標準入力, 標準エラー出力だけはブロックモードで使用しています.

TLI (Transport Layer Interface), ファイルロッキング 等はサポートされていません. これらを利用した場合, どうなるかは不明です.

3.11.6 ジョブコントトロール

PTL は, ジョブコントロール制御には以下のように対応しています.

プロセスが停止 (サスペンド) された場合は, 全てのスレッドが停止します. (第 3.6.1 節 [シグナルの配送], 20 ページ.)

プロセスが background で走行中に, スレッドが `read()` 等によって制御端末からの入力が発生すると, そのスレッドは, プロセスが foreground に回されて入力が完了するまでブロックします.

プロセスが background で走行中に, `write()` 等によって制御端末への出力が発生すると, 端末がバックグラウンドジョブからの出力を禁止している場合でも (例えば `stty tostop` を実行している等), 現在のところ出力は行なわれます.

PTL ではプロセスが, backgroud から foreground に回ったことを知るために, SIGCONT シグナルが送られることを前提としています. これは, 大抵のシェルには当てはまりません (確認した限りでは最近の tcsh (6.03 以降?) が SIGCONT を送るようです). 動いているプロセスに SIGCONT を送ることは特に害が無いため, シェルに手を加えてプロセスが foreground に回った際に SIGCONT を送るようにすべきです. bash に関しては foreground に回した際に SIGCONT を送るようにするパッチが ‘`patches`’ ディレクトリにあります.

PTL では, 標準出力を, 非ブロックモードで使用しています. これは, シェルを混乱させる場合があります (例えば version 1.13.4 より前の bash). PTL では, シェルが混乱しないように最小限の対策を行なっていますが, プロセスを停止 (サスペンド) するために, SIGTSTP でなく SIGSTOP を送ったり, プロセスが捕捉できないシグナルを受け取って死んでしまうと, シェルは混乱します.

3.11.7 移植性

ライブラリで定義している関数, 定数のうち, `_np`, あるいは`_NP` で終わるものは独自のもので, 他の POSIX 1003.1c ベースのライブラリでは提供されません. これらの関数の使用は移植性を低下させます. 注意してください.

NP は, Non Portable の略です.

PTL では ‘`<pthread.h>`’ 中に, `__PTL__` を定義しています. 移植性が問題となる場合, `#ifdef` によってコードを切り分けてください.

4 リファレンス

この章は, PTL のリファレンスマニュアルです.

4.1 スレッド管理のための関数

4.1.1 スレッド属性の操作

```
#include <pthread.h>

int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);
int pthread_attr_getstacksize(pthread_attr_t *attr, size_t *stacksize);
int pthread_attr_setstackaddr(pthread_attr_t *attr, void *stackaddr);
int pthread_attr_getstackaddr(pthread_attr_t *attr, void **stackaddr);
int pthread_attr_setdetachstate(pthread_attr_t *attr, int *detachstate);
int pthread_attr_getdetachstate(pthread_attr_t *attr, int *detachstate);
int pthread_attr_setstackprop_np(pthread_attr_t *attr, int property);
int pthread_attr_getstackprop_np(pthread_attr_t *attr, int *property);
int pthread_attr_setsuspended_np(pthread_attr_t *attr, int suspendstate);
int pthread_attr_getsuspended_np(pthread_attr_t *attr, int *suspendstate);
```

`pthread_attr_init`

`pthread_attr_init()`は, *attr* で指定されたスレッドアトリビュートオブジェクトを初期化します.

生成されたスレッドアトリビュートオブジェクトは, `pthread_create()` によってスレッドを作成する際に用いられます. 単一のスレッドアトリビュートオブジェクトを複数の `pthread_create()`の呼出しに使用しても構いません.

`pthread_attr_destroy`

`pthread_attr_destroy()`は, スレッドアトリビュートオブジェクトを削除するために用います. 削除されたスレッドアトリビュートオブジェクトを使用してはいけません.

`pthread_attr_setstacksize`

`pthread_attr_setstacksize()`は, *attr* のスタックサイズ属性を設定します. *stacksize* の単位は byte です.

`pthread_attr_getstacksize`

`pthread_attr_getstacksize()`は, *attr* のスタックサイズ属性を取得します. *stacksize* の単位は byte です.

`pthread_attr_setdetachstate`

`pthread_attr_setdetachstate()` は, *attr* のデタッチステート属性を設定します. *detachstate* は PTHREAD_CREATE_DETACHED か PTHREAD_CREATE_JOINABLE の値をとります. PTHREAD_CREATE_DETACHED ならば, *attr* を用いて生成された全てのスレッドはデタッチされた状態で開始されます. PTHREAD_CREATE_JOINABLE ならば, デタッチされない状態で開始されます.

`pthread_attr_getdetachstate`

`pthread_attr_getdetachstate()` は, *attr* のデタッチステート属性を取得します.

`pthread_attr_setstackprop_np`

`pthread_attr_setstackprop_np()` は, *attr* のスタックプロパティ属性を *property* に設定します. *property* は以下の定数の論理和です.

`PTHREAD_STACK_SAFE_NP`

安全なスタック (溢れ検出可能)

`PTHREAD_STACK_EXTENSIBLE_NP`

自動拡張可能なスタック

`PTHREAD_STACK_NONE_NP`

特に性質を指定しない

(Thread Stack については第 3.2.3.5 節 [スタックプロパティ], 9 ページ.).

`pthread_attr_getstackprop_np`

`pthread_attr_getstackprop_np()` は, *attr* のスタックプロパティ属性を取得します.

`pthread_attr_setsuspended_np`

`pthread_attr_setsuspended_np()` は, *attr* のサスペンドステート属性を設定します. *suspendstate* はPTHREAD_CREATE_NOT_SUSPENDED_NP か PTHREAD_CREATE_SUSPENDE の値をとります. PTHREAD_CREATE_SUSPENDED_NP ならば, *attr* を用いて生成された全てのスレッドはサスペンドされた状態で生成され, `pthread_resume_np()` が呼ばれるまでブロックします. PTHREAD_CREATE_NOT_SUSPENDED_NP ならば, サスペンドされない状態で開始されます.

`pthread_attr_getsuspended_np`

`pthread_attr_getsuspended_np()` は, *attr* のサスペンドステート属性を取得します.

4.1.2 スレッドの生成

```
#include <pthread.h>

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine)(void *arg), void *arg);
```

`pthread_create()` は, プロセス中に *attr* を属性として, 新たなスレッドを生成します. もし, *attr* が NULL ならば, デフォルトアトリビュート (第 3.2.3 節 [スレッドアトリビュートオブジェクト], 7 ページ.) が用いられます. *attr* が後に変更されたとしても, スレッドの属性には影響を与えません. スレッドの生成に成功すると, `pthread_create()` は, *thread* の指す場所に生成したスレッドの ID を返します.

生成されたスレッドは, *start_routine* から実行され, そのただ一つの引数は *arg* です. *start_routine* からリターンした場合は, 暗黙に *start_routine* からの返り値を引数として `pthread_exit()` が呼ばれます. ただし, 初期スレッドはこれと異なり, `main()` からリターンすると, 暗黙に `main()` からの返り値を引数として `exit()` が呼ばれます.

もし, `pthread_create()` が失敗した場合には, *thread* の指す内容は不定となります.

生成したスレッドのシグナルの状態に関しては 第 3.6.2 節 [シグナルの状態の継承], 21 ページ. を参照して下さい.

4.1.3 スレッドの終了の Wait

```
#include <pthread.h>

int pthread_join(pthread_t thread, void **status);
```

`pthread_join()` は, *thread* が既に終了していない場合, この関数を呼び出したスレッドの実行を *thread* が終了するまで停止させます. `pthread_join()` の呼出しが成功してリターンすると, *status* が NULL でない場合, 終了したスレッドが `pthread_exit()` に渡した値が, *status* の指す場所に返ります. デタッチされたスレッドに対して `pthread_join()` を呼び出してはいけません.

thread に自分自身のスレッドを指定した場合, EDEADLK を返します.

4.1.4 スレッドのデタッチ

```
#include <pthread.h>

int pthread_detach(pthread_t thread);
```

`pthread_detach()` は, *thread* で示されるスレッドが終了した場合, そのスレッドのための領域を再利用しても良いことを宣言します. もし, *thread* が終了していなかった場合は, *thread* が終了次第, 領域を再利用します.

4.1.5 スレッドの終了

```
#include <pthread.h>

void pthread_exit(void *status);
```

`pthread_exit()` は, この関数を呼び出したスレッドを終了させ, *status* の値を, join しているスレッドに伝えます. プッシュされている全ての Cleanup ハンドラはプッシュされた順番と逆順に実行されます. 全ての Cleanup ハンドラが実行された後, スレッドが Thread-Specific データを保持していた場合, 適当なデストラクタ関数が, 定義されない順序で呼び出されます. スレッドの終了時には, Mutex 等のリソースは自動的には一切開放されません.

初期スレッド以外のスレッドが, 作成する時に用いた start routine からリターンする時, 返り値がスレッドの終了ステータスとして用いられます. 初期スレッドが `main()` からリターンすると, その返り値を引数として暗黙に `exit()` が呼ばれます.

暗黙の, あるいは明示的な `pthread_exit()` の呼出しによる cleanup ハンドラやデストラクタ関数の実行中に `pthread_exit()` が呼ばれた場合の動作は不定です.

最後の未終了のスレッドが `pthread_exit()` を呼ぶことによってプロセスは終了します. このときのプロセスの終了ステータスは, 第 3.1.2 節 [スレッドの終了], 3 ページ. を参照して下さい.

4.1.6 スレッドのサスペンド

```
#include <pthread.h>

extern int pthread_suspend_np(pthread_t thread);
extern int pthread_resume_np(pthread_t thread);
```

`pthread_suspend_np`
: `pthread_suspend_np()` は, *thread* をサスペンドするために用います. (第 3.1.5 節 [スレッドのサスペンドの概要], 4 ページ.) 指定したスレッドが既にサスペンドされていた場合, EALREADY を返します.

`pthread_resume_np`
: `pthread_resume_np()` は, サスペンドされた *thread* を再開するために用います. 指定したスレッドがサスペンドされていなかった場合, EALREADY を返します.

スレッドは自分自身をサスペンドしても構いません. この場合, 他のスレッドから `pthread_resume_np()` を呼び出してもらうことになります.

`pthread_mutex_lock()`, `pthread_cond_wait()`, `pthread_join()`, `sleep()` 等の呼び出しによってブロックしているスレッドをサスペンドした場合, サスペンドされたスレッドは, 待っているイベントが発生し, かつ `pthread_resume_np()` が呼ばれるまで実行を再開することはありません.

サスペンドされているスレッドに, 非同期シグナルが配送されることはありません. また, 同期シグナルや `pthread_kill()` によるシグナルが配送された場合, `pthread_resume_np()` が呼ばれるまでシグナルハンドラの実行は延期されます. (第 3.6.1 節 [シグナルの配送], 20 ページ.)

4.1.7 スレッド ID の取得

```
#include <pthread.h>

pthread_t pthread_self(void);
```

`pthread_self()` は, この関数を呼び出したスレッドの ID を返します.

4.1.8 スレッド ID の比較

```
#include <pthread.h>

int pthread_equal(pthread_t t1, pthread_t t2);
```

この関数はスレッド ID *t1* と *t2* を比較し, 同一のスレッドならば非 0, そうでなければ 0 を返します.

4.1.9 パッケージの動的な初期化

```
#include <pthread.h>

pthread_once_t once_control = PTHREAD_ONCE_INIT;

int pthread_once(pthread_once_t *once_control, void (*init_routine)());
int pthread_first_np(pthread_once_t *once_control);
void pthread_first_done_np(pthread_once_t *once_control);
```

これらの関数は, ユーザの指定した関数やブロックを正確に一回だけ呼び出すために存在します. たとえ, この呼出しが複数のスレッドから同時に行なわれたり, 複数回呼ばれても, 関数やブロックは一回しか呼ばれません.

プロセスで最初の *once_control* を伴う `pthread_once()` の呼出しは, *init_routine* を引数無しで呼び出します. それ以降の `pthread_once()` の呼出しは *init_routine* を呼び出しません. `pthread_once()` からのリターンは, *init_routine* が終了していることを保証します. 2 回目以降の呼出し時にまだ `init_routine` が実行中だった場合, `pthread_once()` は `init_routine` が終了するまでブロックします. *once_control* パラメータは, 関連したどの初期化ルーチンが呼ばれたかどうかを決定するのに用いられます.

PTHREAD_ONCE_INIT の値は, ‘`<pthread.h>`’ で定義されます.

もし *once_control* のストレージクラスが auto か, あるいは初期化されていない場合の `pthread_once()` の振る舞いは未定義です.

`pthread_first_np()` は, 最初の *once_control* を伴う呼出しでは非 0 を返し, それ以降の呼出しでは 0 を返します. ユーザが関数内の一回だけ実行したい初期化ブロックは, `pthread_first_np()` が 1 を返した時に実行すれば良いことになります. この際, 初期化ブロックが終了したことを示すため, 初期化ブロックの最後で `pthread_first_done_np()` を, `pthread_first_np()` と同一の引数で実行する必要があります.

4.1.10 スレッドに対するネーミング

スレッドに名前を与えることができます. 名前は, ユニークで無くても構いません. ライブラリでは, SIGSEGV 等が発生して停止する際に, 実行中のスレッドに名前が与えられていれば, その名前を表示します.

```
#include <pthread.h>

int pthread_setname_np(pthread_t thread, const char *name);
int pthread_getname_np(pthread_t thread, char **name);
```

`pthread_setname_np`

`pthread_setname_np()` は, *thread* に名前として *name* を割り当てます. *name* は内部でヒープ領域にコピーされるため, *name* の領域は `pthread_setname_np()` の呼出しの後に開放しても構いません.

`pthread_getname_np`

`pthread_getname_np()` は, *name* に *thread* の名前へのポインタを格納します. *name* は *thread* が exit して, detach されるまで有効な名前を指しています.

4.1.11 スタックキャッシュ

```
#include <pthread.h>

int pthread_alloc_stack_cache_np(pthread_attr_t *attr, int nstack);
```

この関数は, スレッドアトリビュートオブジェクト *attr* で指定されるスタックを *nstack* 個確保し, 確保できたスタックの数を返します. *attr* 中でスタックプロパティ属性と, スタックサイズ属性のみが参照されます. 確保したスタックはスタックキャッシュに蓄えられ, 以降の `pthread_create()` によるスレッドの生成の際に利用されます.

4.2 同期のための関数

4.2.1 Mutex アトリビュートオブジェクト

```
#include <pthread.h>

int pthread_mutexattr_init(pthread_mutexattr_t *attr);
int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
int pthread_mutexattr_getpshared(const pthread_mutexattr_t *attr,
                                  int *pshared);
int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr,
                                  int pshared);
int pthread_mutexattr_getprotocol(pthread_mutexattr_t attr);
int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr,
                                   int protocol);
int pthread_mutexattr_getprioceiling(pthread_mutexattr_t attr);
int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr);
```

`pthread_mutexattr_init`

`pthread_mutexattr_init` は, Mutex アトリビュートオブジェクトを生成, 初期化します. Mutex アトリビュートオブジェクトの初期値については, 第 3.2.4 節 [Mutex アトリビュートオブジェクト], 11 ページ. を参照してください.

`pthread_mutexattr_destroy`

`pthread_mutexattr_destroy`は, Mutex アトリビュートオブジェクトを破棄します.

`pthread_mutexattr_getpshared`

PTL ではサポートしていません.

`pthread_mutexattr_setpshared`

PTL ではサポートしていません.

`pthread_mutexattr_getprotocol`

`pthread_mutexattr_getprotocol()` は *attr* の *Mutex* プロトコル (第 3.3.1.1 節 [プライオリティの逆転の回避], 14 ページ.) を取得します.

`pthread_mutexattr_setprotocol`

`pthread_mutexattr_setprotocol()` は *attr* に *Mutex* プロトコルを設定します.

`pthread_mutexattr_getprioceiling`

`pthread_mutexattr_getprioceiling()` は *attr* の Mutex のシーリングを取得します.

`pthread_mutexattr_setprioceiling`

`pthread_mutexattr_setprioceiling()` は *attr* にシーリングを設定します.

PTL では, 現在のところ PTHREAD_PRIO_PROTECT プロトコルをサポートしていませんが, シーリングの設定, 取得だけは可能です.

4.2.2 Mutex の初期化と破棄

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *mutex, pthread_mutexattr_t *attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

`pthread_mutex_init`

`pthread_mutex_init()` は, *attr* で指定された Mutex アトリビュートオブジェクトを用いて *mutex* で示される Mutex を初期化します. *attr* が NULL ならば, デフォルトの Mutex アトリビュートオブジェクト (第 3.2.4 節 [Mutex アトリビュートオブジェクト], 11 ページ.) が用いられます.

`pthread_mutex_init()` が失敗した場合, Mutex は初期化されず, *mutex* の内容は不定になります.

`pthread_mutex_destroy`

`pthread_mutex_destroy()` は, *mutex* で指定された Mutex を破棄します. ロックされた Mutex を破棄したり, 他のスレッドがブロックしている Mutex を破棄した場合の動作は未定義です.

`PTHREAD_MUTEX_INITIALIZER`

デフォルトの Mutex アトリビュートを使用する場合, PTHREAD_MUTEX_INITIALIZER マクロを使用して, 静的に確保した Mutex を初期化することができます.

4.2.3 Mutex のロックとアンロック

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

`pthread_mutex_lock`

`pthread_mutex_lock()` によって *mutex* で示される Mutex をロックすることが出来ます (第 3.3.1.3 節 [Mutex のロック], 15 ページ.) もし, Mutex が既に他のスレッドによってロックされていた場合, `pthread_mutex_lock()` を呼び出したスレッドは Mutex が再びロックできるようになるまでブロックされます.

`pthread_mutex_lock` が成功すると, *mutex* で示される Mutex を, ロックした状態でリターンします.

呼びだしスレッドが既にロックしている Mutex を再びロックしようとした場合, EDEADLK を返します.

`pthread_mutex_trylock`

`pthread_mutex_trylock()` は *mutex* によって示される Mutex がロックされていた場合に直ちに EBUSY を返すことを除いて, `pthread_mutex_lock()` と同じです.

`pthread_mutex_unlock`

`pthread_mutex_unlock()` は, *mutex* によって示される Mutex をアンロックするために用います.

`pthread_mutex_unlock()` が Mutex をロックしていないスレッドから呼ばれた場合, あるいは, Mutex がロックされていない場合は EPERM を返します.

Mutex を待ってブロックしているスレッドが複数存在した場合, もっとも高いプライオリティのスレッドが Mutex をロックします.

4.2.4 Mutex のプライオリティシーリングの変更

PTL では, 現在 PTHREAD_PRIO_PROTECT プロトコルをサポートしていませんが, シーリングの設定, 取得だけは可能です.

```
#include <pthread.h>

int pthread_mutex_getprioceiling(pthread_mutex_t mutex,
                                 int *prio_ceiling);
int pthread_mutex_setprioceiling(pthread_mutex_t mutex,
                                 int prio_ceiling);
```

pthread_mutex_getprioceiling

`pthread_mutex_getprioceiling()` は Mutex のシーリングを **prio_ceiling* に返します.

pthread_mutex_setprioceiling

`pthread_mutex_setprioceiling()` は, Mutex をロックし, Mutex のシーリングを変更し, Mutex を開放します.

もし, `pthread_mutex_setprioceiling()` が失敗した場合, Mutex のシーリングは変更されません.

4.2.5 Mutex に対するネーミング

Mutex に名前を与えることができます. 名前は, ユニークで無くても構いません. ライブラリでは, スレッドが終了する際に, Mutex がアンロックされていないと, Mutex の名前を表示して警告を発します.

```
#include <pthread.h>

int pthread_mutex_setname_np(pthread_mutex_t mutex,
                             const char *name);
int pthread_mutex_getname_np(pthread_mutex_t mutex, char **name);
```

`pthread_mutex_setname_np`

`pthread_mutex_setname_np()` は, *mutex* に名前として *name* を割り当てます. *name* は内部でヒープ領域にコピーされるため, *name* の領域は `pthread_mutex_setname_np()` の呼出しの後に開放しても構いません.

`pthread_mutex_getname_np`

`pthread_mutex_getname_np()` は, *name* に *mutex* の名前へのポインタを取得します. *name* は, *mutex* が破棄されるまで有効な名前を指しています.

4.2.6 Mutex で Wait 中のスレッドの数

```
#include <pthread.h>

int pthread_mutex_waiters_np(pthread_mutex_t mutex);
```

この関数は, *mutex* でブロックしているスレッドの数を返します. この返り値は, *mutex* がロックされている時のみ信頼できます.

4.2.7 Condition アトリビュートオブジェクトの操作

```
#include <pthread.h>

int pthread_condattr_init(pthread_condattr_t *attr);
int pthread_condattr_destroy(pthread_condattr_t *attr);
int pthread_condattr_getpshared(pthread_condattr_t *attr,
                                int *pshared);
int pthread_condattr_setpshared(pthread_condattr_t *attr,
                                int pshared);
```

`pthread_condattr_init`

`pthread_condattr_init()` は, Condition アトリビュートオブジェクト *attr* を初期化します (第 3.2.5 節 [Condition アトリビュートオブジェクト], 12 ページ.).

`pthread_condattr_init()` が失敗した場合, *attr* の内容は不定となります.

Condition アトリビュートオブジェクトを変更, 破棄しても, それは既に作られた Condition Variable に影響を与えません.

`pthread_condattr_destroy`

`pthread_condattr_destroy()` は, Condition アトリビュートオブジェクトを破棄します.

`pthread_condattr_getpshared`

この関数は, PTL ではサポートしていません.

`pthread_condattr_setpshared`

この関数は, PTL ではサポートしていません.

4.2.8 Condition の初期化と破棄

```
#include <pthread.h>

int pthread_cond_init(pthread_cond_t *cond,
                      const pthread_condattr_t *attr);
int pthread_cond_destroy(pthread_cond_t *cond);
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

`pthread_cond_init`

pthread_cond_init() は, *attr* で指定された Condition アトリビュートオブジェクトを用いて *cond* で示される Condition Variable を初期化します.

attr が NULL ならば, デフォルトの Condition アトリビュート (第 3.2.5 節 [Condition アトリビュートオブジェクト], 12 ページ.) が用いられます.

pthread_cond_init() が失敗した場合, Condition Variable は初期化されず, *cond* の内容は不定になります.

`pthread_cond_destroy`

pthread_cond_destroy()は, Condition Variable を破棄します. 他のスレッドがブロックしている Condition Variable を破棄した場合の動作は未定義です.

`PTHREAD_COND_INITIALIZER`

デフォルトの Condition アトリビュートを使用する場合, PTHREAD_COND_INITIALIZER マクロを使用して, 静的に確保した Condition Variable を初期化することができます.

4.2.9 Conditionのブロードキャストとシグナル

```
#include <pthread.h>

int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

これらの関数は, Condition Variable でブロックしているスレッドをアンブロックします.

`pthread_cond_signal`

pthread_cond_signal() は, 指定した *cond* でブロックしているスレッドを一つアンブロックします. 複数のスレッドが Condition Variable でブロックしていた場合, もっとも高いプライオリティのスレッドがアンブロックされます.

`pthread_cond_broadcast`

pthread_cond_broadcast() は, 指定した *cond* でブロックしているスレッドを全てアンブロックします.

これらの関数によって, Condition Variable でのブロックから開放されるスレッドは, pthread_cond_wait() や pthread_cond_timedwait() からリターンする前に, 関連した Mutex をロックします. (第 3.3.2.2 節 [Condition Variable での Wait], 17 ページ.)

pthread_cond_signal() と pthread_cond_broadcast() は, *cond* でブロックしているスレッドが存在しなければ何の効果もありません.

4.2.10 Condition での Wait

```
#include <pthread.h>

int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
int pthread_cond_timedwait(pthread_cond_t *cond,
                           pthread_mutex_t *mutex,
                           const struct timespec *abstime);
```

`pthread_cond_wait()` と `pthread_cond_timedwait()` は, Condition Variable でブロックするために用いられます. これらは呼びだしスレッドが `mutex` をロックした状態で呼び出さなければなりません. さもなくば未定義の動作を引き起こします. これらの関数は `mutex` をアンロックし, 呼び出しスレッドを Condition Variable `code` でブロックさせます. これらの関数からリターンした際には, Mutex は呼び出しスレッドによってロックされた状態となっています.

`pthread_cond_timedwait()` は, *cond* がシグナルされる前に *abstime* によって指定される絶対時刻が過ぎた場合にエラーでリターンする (Mutex は再びロックされます) ことを除いて, `pthread_cond_wait()` と同一です.

4.2.11 Condition に対するネーミング

Condition Variable に名前を与えることができます. 名前は, ユニークで無くても構いません.

```
#include <pthread.h>

int pthread_cond_setname_np(pthread_cond_t cond, const char *name);
int pthread_cond_getname_np(pthread_cond_t cond, char **name);
```

`pthread_cond_setname_np`
: `pthread_cond_setname_np()` は, *cond* に名前として *name* を割り当てます. *name* は内部でヒープ領域にコピーされるため, *name* の領域は `pthread_cond_setname_np()` の呼出しの後に開放しても構いません.

`pthread_cond_getname_np`
: `pthread_cond_getname_np()` は, *name* に *cond* の名前へのポインタを取得します. *name* は, *cond* が破棄されるまで有効な名前を指しています.

```
#include <pthread.h>

int pthread_cond_waiters_np(pthread_cond_t cond);
```

この関数は, *cond* でブロックしているスレッドの数を返します. 返される値には, 「Condition でのブロックからは開放されているが, 引き続く Mutex のロックでブロックしているスレッド」の数は含まれません. この値は, 関連する Mutex がロックされている場合のみ信頼できます.

4.3 Thread-Specific データのための関数

4.3.1 Thread-Specific データキーの管理

```
#include <pthread.h>

int pthread_key_create(pthread_key_t *key,
                       void (*destructor)(void *value));
int pthread_key_delete(pthread_key_t *key);
```

`pthread_key_create`

この関数は, プロセス中の全てのスレッドで共有される キー を作ります. `pthread_key_create()` によって返されるキーは, Thread-Specific データを指定するために使われます. 異なったスレッドによって同じ キー の値が使われたとしても, `pthread_setspecific()` による キー に対応した値はスレッド毎に管理され, 呼びだしスレッドが存在しなくなるまで存在します. (第 3.4 節 [Thread-Specific データの概要], 18 ページ.)

キー が作成された時, 全ての既存のスレッドのキーに対応した値は NULL で初期化されます. また, 新たにスレッドが生成された場合は, いままでに定義されたキーに対応した値は NULL に初期化されます.

複数のスレッドによる, 同一の *key* を指定した `pthread_key_create()` の並行した呼出しの結果は未定義です.

それぞれのキーに対応して, オプショナルなデストラクタ関数を結び付けることが出来ます. スレッドが終了する際, キー が NULL でないデストラクタポインタを持っていて, スレッドの キー に対応する値が NULL でない場合, その値を引数としてデストラクタ関数が呼ばれます. スレッドが終了する際に複数のデストラクタが存在した場合, デストラクタの呼ばれる順序は不定です.

`pthread_key_delete`

この関数は, キーを削除します. デストラクタは呼び出しません. 既に設定された Thread-Specific データは, アプリケーション側で開放する必要があります. この関数の呼び出し以降, `pthread_getspecific` や `pthread_setspecific` を呼び出した結果は未定義です.

4.3.2 Thread-Specific データの管理

```
#include <pthread.h>

int pthread_setspecific(pthread_key_t key, const void *value);
void *pthread_getspecific(pthread_key_t key);
```

`pthread_setspecific`

`pthread_setspecific()` は, `pthread_key_create()` を用いて得られた *key* に対して Thread-Specific データ, *value* を対応させます. 異なったスレッドが同一のキーに異なった値を結び付けることが出来ます. これらの値は典型的には, 呼びだしスレッドが用いるために動的に確保したメモリブロックへのポインタです.

`pthread_setspecific`

`pthread_getspecific()` は, 呼びだしスレッドの指定した *key* に対応する値を返します.

Thread-Specific データのデストラクタからの `pthread_setspecific()` や `pthread_getspecific()` の呼出しの結果は未定義です.

これらの関数は, マクロとして実装される可能性があります. (PTL では関数として実装しています)

4.4 スケジューリングのための関数

4.4.1 スケジューリング属性の設定

```
#include <pthread.h>

int pthread_attr_setscope(pthread_attr_t *attr, int contentionscope);
int pthread_attr_getscope(pthread_attr_t *attr, int *contentionscope);
int pthread_attr_setinheritsched(pthread_attr_t *attr, int inherit);
int pthread_attr_getinheritsched(pthread_attr_t *attr, int *inherit);
int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
int pthread_attr_getschedpolicy(pthread_attr_t *attr, int *policy));
int pthread_attr_setschedparam(pthread_attr_t *attr,
                               const struct sched_param *param);
int pthread_attr_getschedparam(pthread_attr_t *attr,
                               struct sched_param *param);
```

`pthread_attr_setscope`
`pthread_attr_getscope`

これらの関数は, *attr* の コンテンションスコープ属性 を設定, 取得します (第 3.2.3.4 節 [コンテンションスコープ], 9 ページ.). *contentionscope* が PTHREAD_SCOPE_SYSTEM ならば, グローバルなコンテンションスコープ, PTHREAD_SCOPE_PROCESS ならばローカルなコンテンションスコープです.

PTHREAD_SCOPE_SYSTEM, PTHREAD_SCOPE_PROCESS は, ‘`<pthread.h>`’で定義されます.

PTL ではコンテンションスコープは設定, 取得できますが, 何の意味も持ちません.

`pthread_attr_setinheritsched`
`pthread_attr_getinheritsched`

これらの関数は, *attr* の *inheritsched* 属性を設定, 取得します (第 3.2.3.3 節 [Inherit スケジューリング], 9 ページ.).

`pthread_attr_setschedparam`
`pthread_attr_getschedparam`

これらの関数は, *attr* の スケジューリングプライオリティ属性を設定, 取得します (第 3.2.3.2 節 [スケジューリングプライオリティ], 8 ページ.).

`pthread_create()` によってアトリビュートオブジェクトが使用される際, *inheritsched* 属性が, 生成されるスレッドのその他のスケジューリング属性を決定するために用いられます.

`PTHREAD_INHERIT_SCHED`

スケジューラや, 関連する属性が, スレッドを生成するスレッドから引き継がれます

`PTHREAD_EXPLICIT_SCHED`

スケジューラや, 関連する属性が, アトリビュートオブジェクトの対応する値から設定されます

PTHREAD_INHERIT_SCHED, PTHREAD_EXPLICIT_SCHED は, ‘`<pthread.h>`’ で定義されます.

`pthread_attr_setschedpolicy`
`pthread_attr_getschedpolicy`

`pthread_attr_setschedpolicy)`, `pthread_attr_getschedpolicy()` は, *attr* のスケジューリングポリシーを設定, 取得します.

スケジューリングポリシー属性はスレッドのスケジューリングポリシーを決定します．ポリシーとして，‘`<pthread.h>`’ で定義される `SCHED_FIFO`, `SCHED_RR`, `SCHED_OTHER` のいずれかを選ぶことができます．*policy* の値の意味に関しては，第 3.2.3.1 節 [Scheduling Policy Attribute]，8 ページ．を参照して下さい．

4.4.2 動的なスケジューリング属性の変更

```
#include <pthread.h>

int pthread_getschedparam(pthread_t thread, int *policy,
                          struct sched_param *param);
int pthread_setschedparam(pthread_t thread, int policy,
                          const struct sched_param param);
```

`pthread_getschedparam`
`pthread_setschedparam`

> `pthread_getschedparam()` は *thread* によって与えられるスレッド ID を持つスレッドのスケジューリングポリシーと，スケジューリングプライオリティを *policy*, *param* に格納します．
>
> `pthread_setschedparam()` は *thread* によって与えられるスレッド ID を持つスレッドのスケジューリングポリシーと，スケジューリングプライオリティを *policy*, *param* のポリシーとプライオリティに設定します．この関数は，既存のスレッドのスケジューリングポリシー，プライオリティを再設定する唯一の手段です．

`pthread_setschedparam()` が失敗した場合，*thread* のどのスケジューリング属性も変更されません．

4.4.3 CPU の明渡し

```
void sched_yield();
```

`sched_yield()` は，呼びだしスレッドの CPU を明け渡させます．すなわち，この関数を呼び出したスレッドの実行は再びスケジュールされるまで停止します．

この関数は，走行可能な最も高い優先度のスレッドへコンテキストスイッチを行ないます．

4.4.4 スケジューリングパラメータの範囲

```
#include <pthread.h>

int sched_get_priority_max(int policy);
int sched_get_priority_min(int policy);
int sched_rr_get_interval(pid_t pid, struct timespec *interval);
```

`sched_get_priority_max`
`sched_get_priority_min`

> これらの関数は，スレッドのスケジューリングポリシー毎のプライオリティの最大値，最小値を得るために使用します．

`sched_rr_get_interval`
この関数は, SCHED_RR でのタイムスライスを取得します. PTL では, *pid* は無視されます.

4.5 プロセスコントロールのための関数

4.5.1 プロセスの生成

```
#include <sys/types.h>

pid_t fork(void);
```

プロセスは, 常に単一のスレッドを持つように生成されます. 複数のスレッドを持つプロセスが `fork()` を呼び出すと, 新たなプロセスは呼びだしスレッドと完全なアドレス空間の複製を持ちます (もしかすると, Mutex やその他のリソースの状態をも含みます). 従ってエラーを避けるためには, 子プロセスは exec 関数の一つが呼ばれるまで, 安全な操作だけを実行すべきです. 安全な操作に関しては, POSIX.1 3.3.1.3 Signal Action 参照.

PTL では, 現在 `fork()` に関しては何も処理していません. 全てのスレッドが子プロセスに引き継がれます.

4.5.2 ファイルの実行

```
int execl(char *path, char *arg0, char *arg1, ...,
          char *argn, (char *)0);
int execv(char *path, char *argv[]);
int execle(char *path, char *arg0, char *arg1, ...,
           char *argn, (char *)0, char *envp[]);
int execlp(char *file, char *arg0, char *arg1, ...,
           char *argn, (char *)0);
int execvp(char *file, char *argv[]);
```

`exec` 関数は単一のスレッドを持つプロセスから呼び出すと, POSIX.1 で定義されたように働きます. 複数のスレッドを持つプロセスから呼び出すと, 全てのスレッドが終了し, 新たな実行ファイルが POSIX.1 のようにロードされます.

PTL では, これらの関数については何も処理していませんが, 既存の関数で問題ないと思われます.

4.5.3 プロセスの終了

```
void _exit(int status);
```

スレッドが `_exit()` を呼び出すと, プロセス全体が終了します. さらに, プロセス中の全てのスレッドも終了します. このことはどんな理由によるプロセスの終了に関しても起こります. さらに, `_exit()` の呼出し

によって終了するスレッドは Cleanup ハンドラ (第 3.7.3 節 [スレッド Cleanup], 24 ページ.) を実行しません. また, Thread-Specific データのデストラクタも実行しません. (第 3.4 節 [Thread-Specific データの概要], 18 ページ.)

PTL では, これらの関数については何も処理していませんが, 既存のシステムコールで問題ないと思われます.

4.5.4 プロセスの終了待ち

```
int wait(int *stat_loc);
int waitpid(pid_t *pid, int *stat_loc, int options);
```

`wait()` と `waitpid()` は, 呼びだしスレッドのみをブロックさせせることを除いて POSIX.1 と同様に働きます.

PTL では, これらの関数については何も処理していません. プロセス全体がブロックされます.

4.5.5 プロセスの終了ステータスの設定

```
#include <pthread.h>

int pthread_set_exit_status_np(int status);
```

この関数は, 全てのスレッドが終了した時のプロセスの終了ステータスを設定します. この関数が呼ばれなかった場合の終了ステータスは 0 です.

この関数によって設定された終了ステータスは, `exit()` の呼び出しによる終了ステータスには影響しません.

返り値は, 以前にこの関数によって設定されていたプロセスの終了ステータスの値です. 初めての呼び出しのときには 0 が返ります.

4.6 シグナルのための関数

4.6.1 非同期シグナルの Wait

```
#include <signal.h>

int sigwait(sigset_t *set, int *sig);
```

この関数は, *set* からペンディングされたシグナルを選び, そのシグナルを不可分にペンディングシグナル集合からクリアし, そのシグナルの番号を *sig* に返します. もし, 呼びだし時に *set* 中のシグナルがペンディン

グされていなかった場合, スレッドはシグナルの少なくとも一つが配送されるまでブロックされます. *set* 中のシグナルは, `sigwait()` の呼びだし時にブロックされます.

複数のスレッドが同一のシグナルを Wait するために `sigwait()` を用いている時にシグナルが到着すると, ただ一つのスレッドが `sigwait()` からリターンします. どのスレッドが `sigwait()` からリターンするかどうかは不定です.

`sigwait()` は Cancellation Point です. (第 3.7.2 節 [キャンセルポイント], 24 ページ.)

シグナルを `sigwait()` で待ちたい場合, シグナルアクションを SIG_SIGWAIT_NP に設定することをお勧めします. (第 3.6.1 節 [シグナルの配送], 20 ページ.) これはシグナルによって, `sigwait()` がリターンし, スレッドが処理をしている間に, 次のシグナルが配送された場合の混乱を避けることが出来ます.

```
SIGINT と SIGQUIT を待つコード

int sig;
sigset_t set;
sigemptyset(&set);
sigaddset(&set, SIGINT);
sigaddset(&set, SIGQUIT);
signal(SIGINT, SIG_SIGWAIT_NP);
signal(SIGQUIT, SIG_SIGWAIT_NP);
while (1) {
  sigwait(&set, &sig);
  switch (sig) {
    case SIGINT:
      /* Process SIGINT */
      break;
    case SIGQUIT:
      /* Process SIGQUIT */
      break;
  }
}
```

4.6.2 シグナルの状態の取得と変更

```
#include <signal.h>

int sigaction(int sig, struct sigaction *act,
              struct sigaction *oact);
int pthread_sigmask(int how, sigset_t *set, sigset_t *oset);
int sigsuspend(sigset_t *mask);
int sigpending(sigset_t *set);
int pause(void);
```

`sigaction`

`sigaction()` は, シグナルを受け取った際のアクションを設定するために用いられます. `sigaction()` によって指定されるシグナルハンドラ, 追加のシグナルマスク, フラグはプロセス中の全てのスレッドで共有されます.

	同一のシグナルに対して `sigaction()` と `sigwait()` を並行に用いる場合, `sigwait()` が優先します.
`pthread_sigmask`	`pthread_sigmask()` は呼びだしスレッドのシグナルマスクを設定, 取得します.
`sigsuspend`	`sigsuspend()` は *set* 中のシグナルの一つがスレッドへ非同期に配送されるまで, 呼びだしスレッドだけをブロックすることを除いて, POSIX.1 の定義と同一です.
`sigpending`	`sigpending()` は, スレッドでペンディングされているシグナルと, プロセスでペンディングされているシグナルとの和を返すことを除いて, POSIX.1 の定義と同一です.
`pause`	`pause()` は, シグナルが配送され, シグナル捕捉関数が実行されるまで, 呼びだしスレッドをブロックさせます.

4.6.3 シグナルマスクの操作

```
#include <signal.h>

int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set , int sig);
int sigdelset(sigset_t *set, int sig);
int sigismember(sigset_t *set, int sig);
```

これらの関数は, シグナルマスクを操作します. これらの関数は `sigset_t` 型の変数を操作するだけで, スレッドのシグナルマスクには影響を与えません.

`sigemptyset`	`sigemptyset()` は *set* で指定されたシグナルマスク中の全てのシグナルを 0 に初期化します.
`sigfillset`	`sigfillset()` は *set* で指定されたシグナルマスク中の全てのシグナルを 1 に初期化します.
`sigaddset`	`sigaddset()` は *set* で指定されたシグナルマスクに `sig` で指定したシグナルを追加します.
`sigdelset`	`sigdelset()` は *set* で指定されたシグナルマスクから `sig` で指定したシグナルを削除します.
`sigismember`	`sigismember()` は *set* で指定されたシグナルマスク中に `sig` で指定したシグナルがメンバに存在するかを返します.

4.6.4 大域ジャンプ

```
#include <setjmp.h>

void longjmp(jmp_buf env,int val);
void siglongjmp(sigjmp_buf env, int val);
void setjmp(jmp_buf env);
```

```
void sigsetjmp(sigjmp_buf env, int savemask);
```

`longjmp()`, `siglongjmp()` はそれぞれ `setjmp()`, `sigsetjmp()` によって初期化された環境へとジャンプします. `jmp_buf`, `sigjmp_buf` が呼びだしスレッドによって初期化されなかった場合の動作は未定義です.

スレッド間での `longjmp()`, `siglongjmp()` はできません (`longjmperror()` が呼ばれます)

`setjmp()` および *savemask* が非 0 の `sigsetjmp` は, 呼びだし時にスレッドのシグナルマスクを保存し, 対応する `longjmp()`, `siglongjmp()` によってシグナルマスクを復帰させます.

`setjmp()` および `siglongjmp()` は, `setjmp()` あるいは `sigsetjmp()` を発行してから後にプッシュされた Cleanup ハンドラを全て実行します.

4.6.5 アラーム

```
unsigned int pthread_alarm_np(unsigned int second);
```

`pthread_alarm_np()` を呼び出すことによって, SIGALRM シグナルが, `pthread_alarm_np()` を呼び出したスレッドに非同期に配送されます.

POSIX では, alarm() はスレッドにではなく, プロセスに SIGALRM を送ることになっていますが, PTL ではまだ実装していません.

4.6.6 スレッドの実行の遅延

```
unsigned int sleep(unsigned int second);
int nanosleep(const struct timespec *sleep, struct timespec *remain);
void usleep(unsigned int usecond);
```

`sleep`
: `sleep()` は *second* で指定した実時間での秒数が経過するか, 呼びだしスレッドにシグナルが配送されるまで, 呼びだしスレッドをブロックさせます. 返り値は *second* の残り時間で, シグナルで割り込まれない限りは 0 です.

`nanosleep`
: `nanosleep()` は, ナノセコンド単位で時間を指定できることを除いて, `sleep()` と同一です (ただし, 精度は実装に依存します.). *sleep* で指定した実時間での時間が経過するか, 呼びだしスレッドにシグナルが配送されるまで, 呼びだしスレッドをブロックさせます. *remain* が NULL でなければ, *remain* の指す場所には *sleep* の残り時間が格納されます.

`usleep`
: `usleep()` は, 互換性のために用意されています. 引数で指定した時間 (単位はマイクロ秒), 呼びだしスレッドはブロックします. ブロック中にシグナルが配送された場合でも, スリープは継続されます.

: usleep は POSIX 規格ではありません.

4.6.7 シグナルの送信

`int raise(int` *sig*`);`

`raise()` はプログラムから明示的にシグナルを生成するために用います．この関数の効果は, `pthread_kill(pthread_self(), sig)` と同一です．

4.6.8 スレッドへのシグナルの送信

`#include <signal.h>`

`int pthread_kill(pthread_t` *thread*`, int` *sig*`);`

`pthread_kill()` は, 指定したスレッドへ非同期にシグナルを送るために用います．

`kill()` と同じように, エラーチェックのために *sig* として 0 を与えることが出来ます．この場合シグナルは実際には送られません．

4.7 キャンセルのための関数

4.7.1 スレッドのキャンセル

`#include <pthread.h>`

`int pthread_cancel(pthread_t` *thread*`);`

`pthread_cancel()` は, *thread* のキャンセルを要求します．

4.7.2 Cancelability State の設定

`#include <pthread.h>`

`int pthread_setcancelstate(int` *state*`, int *`*oldstate*`);`
`int pthread_setcanceltype(int` *type*`, int *`*oldtype*`);`

`pthread_setcancelstate`

`pthread_setcancelstate()` は呼びだしスレッドのキャンセル許可フラグを *state* に設定し, 以前のキャンセル許可フラグを *oldstate* に返します．

state は PTHREAD_CANCEL_ENABLE, PTHREAD_CANCEL_DISABLE のいずれかです． PTHREAD_CANCEL_ENABLE の時は *state* を設定した後, `pthread_setcancelstate()` はキャンセルポイントになります．

`pthread_setcanceltype`
: `pthread_setcanceltype()` は呼びだしスレッドの キャンセルタイプを *type* に設定し, 以前のキャンセルタイプを *oldtype* に返します. *type* は `PTHREAD_CANCEL_DEFERRED`, `PTHREAD_CANCEL_ASYNCHRONOUS` のいずれかです.

4.7.3 キャンセルのテスト

```
#include <pthread.h>

void pthread_testcancel(void);
```

`pthread_testcancel`
: `pthread_testcancel()` は呼びだしスレッドにキャンセルポイントを作ります. スレッドがこの関数を実行する前に, キャンセル要求がなされていて, キャンセル許可フラグが `PTHREAD_CANCEL_ENABLE` の場合, キャンセルが実行されます. `pthread_testcancel()` は, キャンセル許可フラグが `PTHREAD_CANCEL_DISABLE` の場合は, なんの効果もありません.

4.7.4 Cleanup ハンドラの設定

```
#include <pthread.h>

void pthread_cleanup_push(void (*routine)(void *arg), void *arg);
void pthread_cleanup_pop(int execute);
void pthread_cleanup_push_f_np(void (*routine)(void *arg), void *arg);
void pthread_cleanup_pop_f_np(int execute);
```

`pthread_cleanup_push`
: `pthread_cleanup_push()` は *routine* を呼び出しスレッドの cleanup スタックに push します. Cleanup ルーチンは以下の場合にスタックから pop され, 実行されます.

- スレッドが exit する (`pthread_exit()` の呼びだし)
- スレッドがキャンセルされる.
- スレッドが `pthread_cleanup_pop()` を 0 でない引数で呼び出す

`pthread_cleanup_pop`
: `pthread_cleanup_pop()` は, 呼びだしスレッドの cleanup スタックのトップからルーチンを削除します. *execute* が 0 でない場合は, 削除するルーチンを実行します.

`pthread_cleanup_push_f_np`
: `pthread_cleanup_push()` の関数版です.

`pthread_cleanup_pop_f_np`
: `pthread_cleanup_pop()` の関数版です.

`pthread_cleanup_push()`, `pthread_cleanup_pop()` は高速化のためマクロとして実装されています. このため, これらの関数は同一の lexical scope でペアとなって現れなければなりません (つまり, `pthread_cleanup_push()` は最初の文字が '{' で始まる文字列を導き, `pthread_cleanup_pop()` は最後の文字が '}' で終わる文字列を導くということです). この制限がネックとなる場合には, 関数版の `pthread_cleanup_push_f_np()`, `pthread_cleanup_pop_f_np()` を利用して下さい.

`pthread_cleanup_push()` でプッシュした Cleanup ハンドラを `pthread_cleanlup_pop_f_np()` でポップすることは構いません. (この場合でも, 対応する lexical scope に `pthread_cleanup_pop()` が存在しなければなりません.)

マッチしない `pthread_cleanup_push()` (あるいは`pthread_cleanup_push_f_np()`) と `pthread_cleanup_pop()` (あるいは `pthread_cleanup_pop_f_np()`) がマッチしていない時の `longjmp()`, `siglongjmp()` の呼出しの結果は, 未定義です. Cleanup ハンドラからの `longjmp()`, `siglongjmp()` の呼出しの結果も未定義です.

4.8 ログのための関数

```
#include <pthread.h>

extern int pthread_log_np(const char *format, ...);
```

(この機能は現在メンテナンスされていません)

この関数を用いて, ログファイル (第 3.8 節 [ログ機能について], 25 ページ.) に, 任意の文字列を出力することが可能です. *format* は, printf 形式のフォーマット文字列で, 後に任意個数の引数が続きます.

4.9 リエントラント関数

この章では, リエントラント化された関数について述べます.

4.9.1 時間・時刻関数

```
#include <time.h>

struct tm *localtime_r(const time_t *clock, struct tm *result);
struct tm *gmtime_r(const time_t *clock, struct tm *result);
char *asctime_r(const struct tm *tm, char *buf);
char *ctime_r(const time_t *clock, char *buf);
```

これらの関数は, `localtime()`, `gmtime()`, `asctime()`, `ctime()` のスレッド対応版です. 引数のポインタの指す領域に返るように変更されています. `asctime_r`, `ctime_r` は, 最低 26 文字のバッファを確保する必要があります.

4.9.2 文字列関数

```
#include <string.h>

char *strtok_r(char *s, const char *sep, char **lasts);
```

`strtok()` のスレッド対応版です.

最初の呼び出しでは, *s* にヌル文字で終了する文字列, *sep* にヌル文字で終了するセパレータ文字の文字列, *lasts* に char へのポインタのアドレスを渡します. `strtok_r()` は文字列 *s* 中の最初のトークンへのポインタを返します. その際, トークンの直後にヌル文字を置き, *lasts* の指すポインタをそのヌル文字の直後を指すように更新します.

以降の呼び出しでは, *s* にはヌルポインタを渡し, *lasts* は変更されていてはなりません. `strtok_r()` は, 呼び出すごとに次のトークンを返します. トークンが無くなるとヌルポインタが返ります.

4.9.3 ヒープメモリ操作関数

PTL では, 以下のヒープメモリ操作関数は, 内部で逐次的に実行されるようになっているため, スレッドから安全に呼ぶことが出来ます.

```
extern void *malloc(size_t size);
extern void *realloc(void *ptr, size_t size);
extern void *calloc(size_t nelem, size_t elmsize);
extern void free(void *ptr);
```

4.9.4 標準入出力ライブラリ (stdio)

PTL では, 以下の標準入出力ライブラリの関数群をスレッド対応にしています. これらの関数を実行中は, 指定したストリーム (`FILE*`) に対応するデスクリプタが Mutex を用いてロックされます.

```
#include <stdio.h>

fclose fdopen fflush fgetc fgets fopen fprintf fputc fputs
fread freopen fscanf fseek ftell fwrite getc getchar gets
getw printf putc putchar puts putw rewind scanf setbuf
setbuffer setlinebuf setvbuf sprintf sscanf ungetc vfprintf
vprintf
```

さらに, 以下の関数が追加されています.

```
#include <stdio.h>

void flockfile(FILE *file);
void funlockfile(FILE *file);
int getc_unlocked(FILE *file);
int getchar_unlocked();
int putc_unlocked(char c, FILE *file);
int putchar_unlocked(char c);
```

`flockfile()` は, ファイル構造体 *file* をロックし, 呼び出しスレッドが独占的に使えるようにします. `funlockfile()` はロックを解除します. このロックは, ネストしていても構いません. つまり, 複数回 `flockfile()` を呼び出し, 同じ数だけ `funlockfile()` を呼び出した場合, 最後の `funlockfile()` の呼び出しの時にロックが解除されます.

`getc_unlocked()`, `getchar_unlocked()`, `putc_unlocked()`, `putchar_unlocked()` は, ロックされたファイル構造体に対して働きます. これらは, `_unlocked` の付かないバージョンに比べて, 高速に動作します. (マクロとして実装されています).

4.9.5 テンポラリファイルの作成

```
#include <stdio.h>

char *mktemp(char *template);
int mkstemp(char *template);
char *tmpnam(char *s);
char *tempnam(const char *dir, const char *prefix);
FILE *tmpfile();
```

これらはテンポラリファイルを扱うための関数です. 複数のスレッドがこれらの関数を並行に呼び出しても, 比較的安全にテンポラリファイルを得ることが出来ます.

詳細は, UNIX のオンラインマニュアルを参照してください.

これらの関数のうち, ファイル名を得るもの (`mktemp()`, `tmpnam()`, `tempnam()`) は安全ではありません. これらの関数は同一プロセス内ではユニークなファイル名を生成しますが, ファイル名の生成と, そのファイルの create の間にタイムラグがあるため, 他のプロセスによってそのファイル名が使用される可能性があります. これらの関数は, 自分自身のプロセス ID をファイル名に埋め込むので, このような可能性はほとんど無いと思われますが, 新しいアプリケーションでは, `mkstemp()`, `tmpfile()` の使用をお勧めします.

4.10 入出力のための関数

PTL では, 特定のスレッドの I/O 操作によってプロセス全体がブロックしないように, 代替の I/O 関数を用意しています. これらの関数では, I/O 操作が直ちに終わらない場合, 呼びだしスレッドのみがブロックします.

これは, PTL 内部で UNIX の非ブロック IO を用いることによって実現されています. 現在のところ, ユーザが `fcntl()` を用いて非ブロックフラグ (`FNDELAY`) を設定しても, 無視されます.

```
#include <sys/types.h>
#include <sys/socket.h>

int accept(int fd, struct sockaddr *addr, int *addrlen);
int connect(int fd, struct sockaddr *name, int *namelen);
ssize_t read(int fd, char *buf, int nbytes);
ssize_t readv(int fd, struct iovec* iov, int iovcnt);
ssize_t write(int fd, char *buf, int nbytes);
ssize_t writev(int fd, struct iovec* iov, int iovcnt);
ssize_t send(int fd, char *msg, int len, int flags);
ssize_t recv(int fd, char *buf, int len, int flags);
ssize_t recvfrom(int fd, char *buf, int len, int flags,
                 struct sockaddr *from, int *fromlen);
```

```
ssize_t recvmsg(int fd, struct msghdr *msg, int flags);
```

`write()`, `writev()`, `read()`, `readv()` は, 不可分に実行される保証はありません. すなわち, 複数のスレッドが同一のファイル記述子で平行して I/O を行った場合, 入出力が 「混ざる」可能性があります.

```
#include <sys/types.h>
#include <sys/time.h>

int select(int nfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);
```

複数の並行に動作するスレッドが同一のファイルデスクリプタに対して `select()` した場合, デスクリプタが有効になると, `select()` 中の全てのスレッドが `select()` からリターンします. 複数のスレッドが同一のデスクリプタに対して `select()` することは勧められません.

#include <sys/types.h> #include <sys/ipc.h> #include <sys/msg.h>

int msgsnd(int *msqid*, void *msgp, int *msgsz*, int *msgflg*); int msgrcv(int *msqid*, void *msgp, int *msgsz*, long *msgtyp*, int *msgflg*);

OS が SYSV スタイルのメッセージキューをサポートしている場合, PTL でも使用可能です.

4.11 その他の関数

4.11.1 時刻の取得

```
#include <time.h>

int clock_gettime(clockid_t clock_id, struct timespec *tp);
```

clock_gettime は, *clock_id* で指定される現在のタイマーの値を *tp* に返します.

PTL では, *clock_id* は, CLOCK_REALTIME (システムクロック) しかサポートしていません.

4.11.2 セマフォ

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop(int semid, struct sembuf *sops, int nsops);
```

OS が SYSV スタイルのセマフォをサポートしている場合, PTL でも使用可能です.

Function Index

Concept Index

目 次