修士論文

POSIX 準拠 OS のスケジューラを対象とした
網羅的テスト手法に関する研究

長谷川 央

主指導教員 青木 利晃

北陸先端科学技術大学院大学
先端科学技術研究科
（情報科学）

令和 4 年 9 月

Abstract

Nowadays, in the field of embedded systems, IoT and AI features are in high demand. As a result, POSIX-compliant operating systems (POSIX-compliant OSs), such as Linux, are used as the OSs. Since typical POSIX-compliant OSs and middleware applications are provided as open source software and readily available, there is a significant advantage that developers can make a prototype easily and quickly. Moreover, typically, POSIX-compliant OSs have much more flexibility to add additional functions than traditional real-time OSs.

However, since POSIX is originally a specification for general-purpose OSs, there are parts where strict behavior is not specified to improve the versatility. Therefore, even if OSs comply with POSIX, their behavior can be completely different. For instance, POSIX does not specify how to allocate threads to each CPU core, and it depends on the implementation of each OS. In addition, since the behavior of OS itself is more complicated than traditional OSs, testing for them is challenging.

Unlike desktop and server applications computers, embedded systems are often used when failures can enormously impact human life. For instance, recently, an application for self-driving cars has also been considered, and Adaptive AUTOSAR, an extension of POSIX, has already been created. In such a mission-critical system, it is highly possible that the introduction of bugs will severely impact society and human life, so it is crucial to ensure the quality through testing or verification.

Especially, scheduling functions are essential for mission-critical systems, and they must not have bugs. Scheduling is the process of managing the execution order of each program (threads/processes). If a bug is included here, it will easily affect the entire system, leading to malfunctions or hangs.

Scheduling-related APIs are stateful; that is, the execution results change depending on the calling order of the APIs. In this case, model-based testing (MBT) is used. To apply MBT, first, we need to create a model that represents the desired behavior using the specification such as POSIX. And secondly, create test cases using the model and execute them.

Through the exhaustive testing for the scheduling-related APIs, we can obtain confidence that the scheduling functions work as we think. However, since POSIX has a problem that did not occur in testing for traditional real-time OSs, such as OSEK/VDX-compliant OSs, the traditional MBT method cannot be directly applied to POSIX-compliant OSs. For instance, consider a situation where the limitation of parallel running threads is two and two threads with low priority, TID 1 and TID 2, are already running, and a new thread TID 3, which has

higher priority, is now created. In this case, two behaviors satisfy POSIX: (1) since TID 1 has lower priority than TID 3, TID 1 is suspended, and TID 3 starts to run instead of TID 1, and (2) since TID 2 has lower priority than TID 3, TID 2 is suspended and TID 3 starts to run instead of TID 2. In this research, we call this property that the execution results are not uniquely determined from the specification as the indeterminacy of the execution result.

Traditional real-time OSs such as OSEK/VDX-compliant OSs do not have this property, and the specification determines only one correct behavior. However, since POSIX-compliant OSs have the property, the traditional MBT is not reasonable for POSIX-compliant OSs. In this study, we have proposed an MBT method that supports the indeterminacy of execution results and achieved exhaustive testing for the scheduling-related APIs of POSIX.

目次

図目次

表目次

第 1 章

導入

1.1 背景

近年，組込み機器分野において，IoT 化や AI 搭載などの高機能化が進んでいる．それに伴い，組込み機器の OS として，従来のリアルタイム OS の替わりに，Linux を始めとした POSIX [1] 準拠 OS が使用されるようになった．POSIX 準拠 OS は基本的にオープンソースであり，多種多様なミドルウェア・アプリケーションが公開されているため，プロトタイプの開発が容易に素早く行えるという利点がある．また，汎用 OS であるため，従来のリアルタイム OS に比べ，柔軟な機能追加が可能である．しかし，元々，POSIX [1] は汎用 OS 用の仕様であるため，汎用性を重視して厳密な動作が定められていない部分がある．そのため，同じ POSIX 準拠 OS であっても，動作が異なる部分がある．例えば，各 CPU コアへのスレッドの割り当て方は POSIX には明記されておらず，各 OS の実装に依存している．更に，従来のリアルタイム OS に比べて OS 自体の動作が複雑であるため，従来のリアルタイム OS に比べ，テスト作成の難易度が難化している．

組込み機器はデスクトップ・サーバ用途の計算機とは異なり，不具合が人命などに大きな影響を及ぼす場面で使用されることが多い．近年では，自動運転車での使用も検討されており，POSIX [1] を拡張した Adaptive AUTOSAR [2] が既に策定されている．このような，ミッションクリティカルシステムでは，バグの混入が社会や人命に深刻な影響を与える可能性が高いため，テストや検証による品質の確保が重要となる．

ミッションクリティカルシステムにおいて，特にバグがあってはならない機能としてスケジューリングに関する機能が挙げられる．スケジューリングとは，各プログラム（スレッド・プロセス）の実行順序を管理するための処理であり，ここにバグが含まれていると容易にシステム全体へ影響が及び，誤動作やシステム全体のハングを招いてしまう．ま

た，バグが含まれていない場合であっても，アプリケーション開発者が仕様を誤解していた場合，認識の齟齬によりバグが発生する可能性がある．

スケジューリングなどの現状態によって動作が変化する処理の検証を行う手法として，モデルベーステスト (MBT) が用いられる．従来のリアルタイム OS である OSEK/VDX 準拠 OS に対して MBT を用いた研究 [3] では，スケジューリングに関係する API の仕様をモデル検査ツール SPIN の仕様記述言語 Promela を用いて記述し，Promela の記述を基に Spin 上の状態を網羅するテストケースを自動生成することで検証を行っている（図 1.1）．スケジューリング関連の API は，ステートフルである（状態を持つ）ため，API の呼出し順によって実行結果が変化する．そのため，様々な順序で API を呼び出し，テストを行う必要がある．このとき，テストケースは API の引数や各 API の呼び出し順の組み合わせの数だけ必要であるため，テストケース数は膨大な数となる．そのため，テストケースは自動で生成できなければならない．

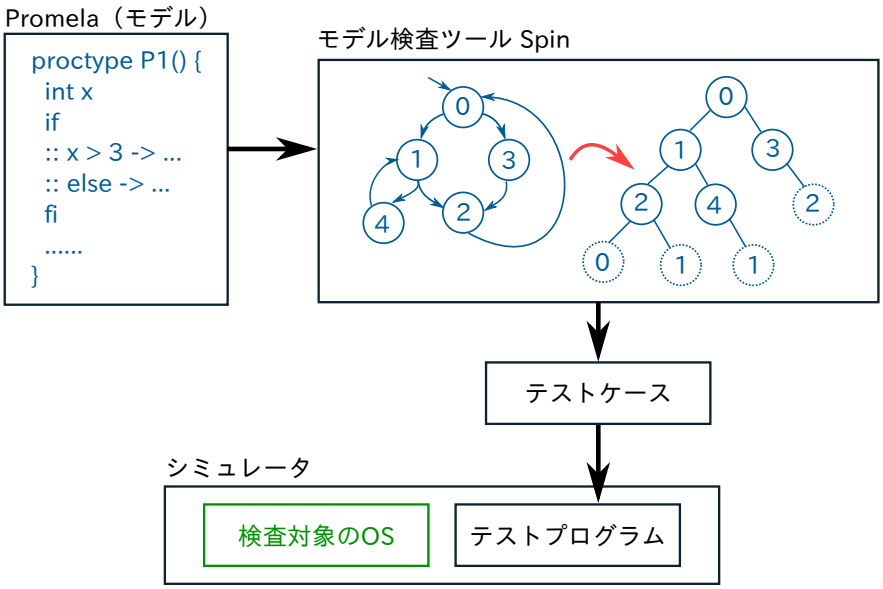


図 1.1　OSEK/VDX 準拠 OS に対するモデルベーステスト [3]

POSIX 準拠 OS のスケジューリング関連の API を対象として，呼び出し順序を網羅するようなテストを生成・実行することで，これらの機能が正しく動作するという確証を得られると期待できる．しかし，POSIX 準拠 OS では，OSEK/VDX 準拠 OS で生じなかった問題が発生するため，[3] の手法をそのまま転用することはできない．例えば，最大 2 つのスレッドが並列に実行できる環境（CPU のコア数が 2 つなど）において，低優先度

のスレッド 2 つ (TID 1, 2) が既に実行中である状況を考える（図 1.2の左側前半）．このとき，高優先度のスレッド (TID 3) を作成する場合，図 1.2の 1 と 2 の 2 つの動作が仕様から考えられる．1 では，低優先度の TID 1 の実行が一時中断され，高優先度の TID 3 が代わりに実行されている．一方で，2 では，TID 2 の実行が一時中断されている．このような，仕様から実行結果が一意に定まらない POSIX の性質を本研究では，実行結果の不確定性と呼ぶ.

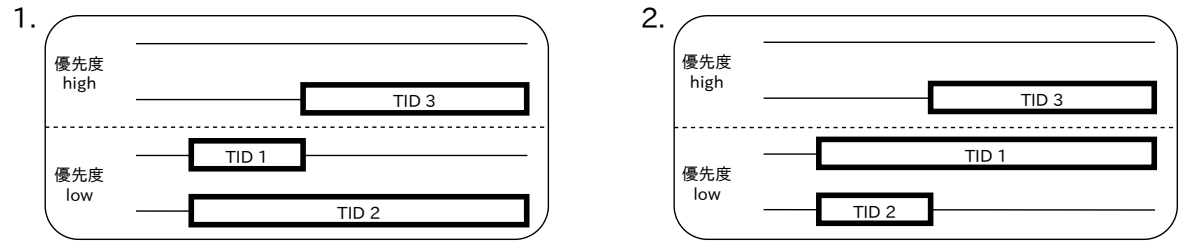


図 1.2　POSIX（仕様）から動作が複数考えられる例

OSEK/VDX 準拠 OS などの従来のリアルタイム OS では，実行結果の不確定性は生じず，1 つの API 列に対し，1 つの正しい動作が定まっていた．しかし，POSIX 準拠 OS では，実行結果の不確定性があるため，従来のテスト手法を利用することが困難である．本稿では，POSIX 準拠 OS のスケジューラに対する自動テストを実現するために，実行結果の不確定性に対応した MBT の手法を提案する.

1.2　本稿の構成

一般的なモデルベーステストと同様に，振る舞い仕様のモデル化を行い，そのモデルからテストスイートを自動生成する．第 2 章にて，モデル化に必要な事柄について導入を行う．第 3 章では，POSIX [1] に規定されている内容を基に，POSIX のスケジューリングに関する仕様の形式化を行う．第 4 章では，形式化した仕様を用いてテストスイートを自動生成する手法の提案を行う．第 5 章では，実装に関して説明を行う．第 6 章で，実験の設定を説明し，第 7 章で実験結果の考察を行っている.

第 2 章

準備

2.1 ユーザモードとカーネルモード

主記憶装置や CPU などの計算資源を管理や，スケジューリング，ユーザ定義のプログラム（アプリケーション）を実行開始する機能をなど，OS に必須な機能を持つプログラム（OS 自体のプログラム）のことをカーネルと呼ぶ．一方で，カーネル以外のプログラムのことをユーザアプリケーションと呼ぶ．ユーザアプリケーションが自由にカーネルに干渉できてしまうと，カーネルに対してユーザアプリケーションが悪影響を与えられるようになってしまう．カーネルが正しく処理を行えるようにするために，POSIX 準拠 OS を始めとした汎用 OS では，user mode と kernel mode (supervisor mode, privileged mode) の 2 つのモードに分けて処理が行われている．例えば，文字列を画面に表示するためには，`printf` 関数などを使用するが，ディスプレイなどの入出力デバイスはカーネルが管理しているため，表示処理はカーネルが行っている．図 2.1のように，ユーザモードで動作しているライブラリ内で `write` システムコールを呼び出すと，カーネルモードに移行し，ディスプレイに文字が表示される．

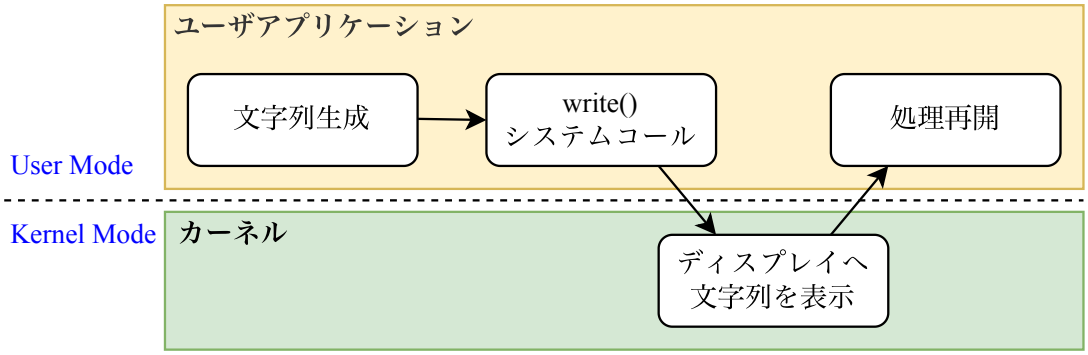


図 2.1　ユーザモード・カーネルモードの例

POSIX に記載されている API は全てユーザアプリケーション上で使用する API であり，その後のカーネルの動作は OS の種類によって異なる．

2.2　プロセスとスレッド

OS 上で実行されるユーザーアプリケーションを管理する単位をプロセスと言う．POSIX 準拠 OS などの汎用 OS では，OS がプロセスごとにメモリを割り当て，プロセス中では仮想アドレスを用いて物理メモリへのアクセスを行う（図 2.2）．そのため，実際のアドレス（物理アドレス）について，プログラマが意識せずとも，プログラムの作成・実行が可能となっている．また，効率的かつ柔軟なメモリの管理・確保が行えるような仕組みとなっている．ただし，その弊害で，プロセス間のデータ共有には POSIX メッセージキューなどを使ったプロセス間通信を行う必要がある．これらの通信には，カーネルによる処理が必要となるため，オーバーヘッドが生じるというデメリットが有る．

プロセスは 1 つ以上のスレッドから構成される．スレッド間ではメモリ空間を共有しているため，データの共有はユーザモードの処理のみで完結する（図 2.2）．また，作成時にメモリの確保などを行う必要がないため，プロセスに比べ，スレッドの作成は高速に行える．そのため，並行処理に適している．しかし，プロセス間通信とは異なり，スレッド間で共有されるデータの整合性は保証されないため，プログラマが Mutex などの排他制御機構を使用して手動で保証を行う必要がある．

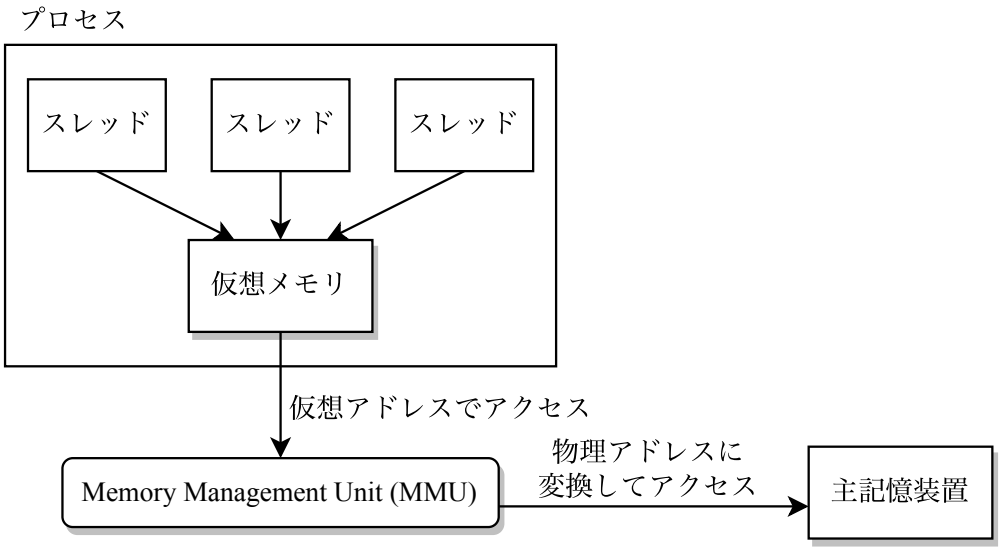


図 2.2　プロセス・スレッドとメモリの関係性

2.3　マルチコアプロセッサ

1 つの集積回路上に複数のプロセッサが搭載されたマイクロプロセッサをマルチコアプロセッサと呼び，マルチコアプロセッサ上のプロセッサをコア（core）と言う．

また，図 2.3のように，複数のコアで主記憶装置を共有する方式を共有記憶型マルチプロセッサ（shared memory multiprocessor; SMP）と言う．SMP では 1 つの OS が複数のコアを管理・使用する．これにより，OS は，複数のスレッドを同時並行的に処理することができる．近年の計算機では，SMP が一般的であるため，本稿においても SMP を前提として扱う．

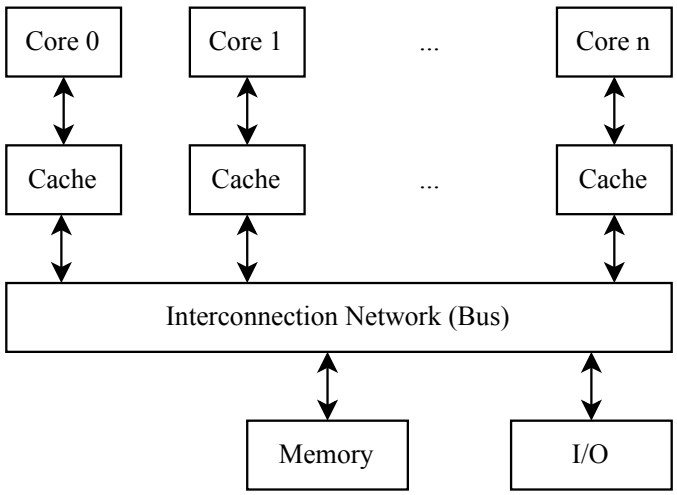


図 2.3　共有記憶型マルチプロセッサ [4]

2.4　スケジューリングポリシー

スケジューリングの方式をスケジューリングポリシーと呼ぶ．スレッドごとにスケジューリングポリシーが設定される．また，POSIX 準拠 OS には，SCHED_FIFO, SCHED_RR，SCHED_SPORADIC，SCHED_OTHER の 4 つのポリシーが実装されていなければならない．組込み機器では主に，SCHED_FIFO と SCHED_RR が用いられる．

■SCHED_FIFO [1]　POSIX では，実行前のスレッド（Ready 状態のスレッド）を並べたキューをスレッドリストと呼んでいる．ここでは，用語を統一するために，Ready Queue と呼ぶ．Ready Queue は優先度付きキューである．

SCHED_FIFO ポリシー下での Ready Queue 上のスレッドの位置に関する処理は以下の通りである [1].

- 実行中のスレッドが Preempt されたとき，そのスレッドは対応する優先度のキューの先頭に並べられる．
- Waiting 状態のスレッドが Ready 状態になったとき，対応する優先度のキューの末尾に追加される
- 実行中のスレッドが `sched_setscheduler()` を呼び出したとき，スレッドは指定されたスケジューリングポリシーと優先度に変更される．
- 実行中のスレッドが `sched_setparam()` を呼び出したとき，呼び出し元のスレッドの優先度が引数で指定した値に変更される．

- 実行中のスレッドが pthread_setschedparam() を実行したとき，スレッドは指定されたスケジューリングポリシーと優先度に変更される．
- 実行中のスレッドが pthread_setschedprio() を実行したとき，呼び出し元のスレッドの優先度が引数で指定した値に変更される．
- pthread_setschedprio() 以外によってスレッドの優先度が変更された場合，その優先度に対応するキューの末尾に追加される．
- pthread_setschedprio() によってスレッドの優先度が変更された場合，その優先度の変更方向によってキューの追加場所が異なる．
  - 優先度が上がった場合，キューの末尾に追加される．
  - 優先度が変更されなかった場合，キュー上の位置は変更されない．
  - 優先度が下がった場合，キューの先頭に追加される．
- 実行中のスレッドが sched_yield() を実行した場合，その呼び出し元スレッドは自身の優先度のキューの末尾に追加される．
- 上述の場合以外に，キュー上のスレッドの位置が変更されることはない．

有効な優先度は，sched_get_priority_max() と sched_get_priority_min() の間であり，最低でも 32 個の優先度が提供されている [1].

(スケジューリングの例)

SCHED_FIFO 下でのスレッドのスケジューリングの例を図 2.4に示す．図は左から時系列順に実行中のスレッドを表している．スレッドは T_1, T_2, T_3 の 3 つが存在し，各優先度とスレッドの終了に必要な処理時間は次を想定する．

- T_1: 優先度 低，処理時間 3
- T_2: 優先度 高，処理時間 2
- T_3: 優先度 高，処理時間 3

最初は低優先度の T_1 が実行されているが，$T = 1$ のときに高優先度のスレッドが投入されたことにより，T_1 の代わりに T_2 が実行され始める．T_2 が実行を終了した後には，同じく高優先度の T_3 が実行され，T_3 の終了後に T_1 が実行を再開する．

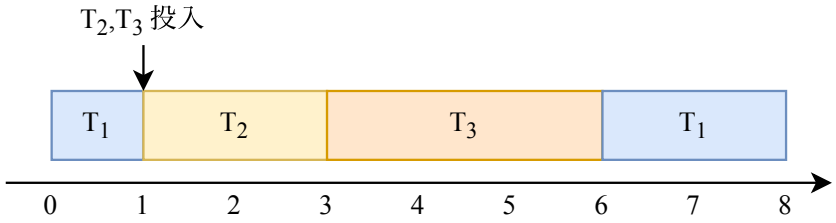


図 2.4　SCHED_FIFO のスケジューリングの例

■SCHED_RR　基本的には SCHED_FIFO と同様の性質・動作である．Running 状態になってから，`sched_rr_get_interval()` で取得可能な時間（タイムスライス）を過ぎると実行中のスレッドを対応する優先度のキューの末尾に追加し，キューの先頭のスレッドを Running 状態に変更する．

(スケジューリングの例)

SCHED_RR 下でのスレッドのスケジューリングの例を**??**に示す．図は左から時系列順に実行中のスレッドを表している．スレッドの設定は SCHED_FIFO の例と同様である．また，タイムスライスは $T = 1$ とする．

最初は低優先度の T_1 が実行されているが，$T = 1$ のときに高優先度のスレッドが投入されたことにより，T_1 の代わりに T_2 が実行され始める．SCHED_FIFO とは異なり，タイムスライスにより，$T = 2$ の段階で T_2 が中断され，T_3 が実行開始する．以降．各スレッドが処理終了するまで，$T = 1$ で交代をしながら実行を継続し，高優先度スレッド T_2, T_3 の実行終了後に T_1 の実行が再開する．

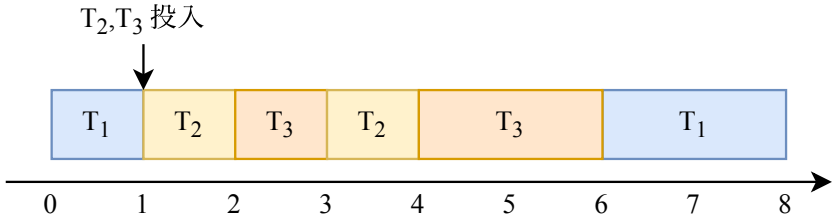


図 2.5　SCHED_RR のスケジューリングの例

2.5　探索アルゴリズム

図 2.6のような状態遷移モデルが与えられたとする．このとき，各状態を走査する際に用いられるアルゴリズムとして深さ優先探索と，その派生アルゴリズムである Stateless

Search がある．以下に，各アルゴリズムの擬似コードと図 2.6に対して適用した際の探索木を示す．

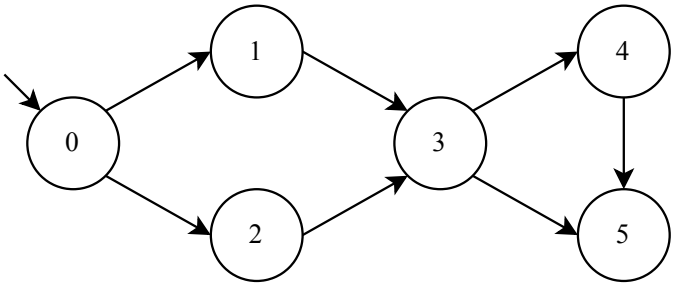


図 2.6　素朴な状態遷移モデルの例

2.5.1　Depth-First Search (DFS)

一般的な深さ優先探索である．擬似コードアルゴリズム 1は [5] から引用している．アルゴリズム 1の $Statespace$ は探索全体で訪れた状態を格納しており，$Stack$ には，遷移とその遷移先の状態を格納している．$Add\_statespace$ は，V に状態を追加する操作である．また，$Push\_Stack$ は Stack D に遷移と状態の組から構成される要素を追加する操作である．$Top\_Stack(D)$ は，D に直近に追加された要素を得る操作である．D 自体に変更は加えない．7 行目は，$Top\_Stack(D)$ で得られた組のうち，右側の値を s に代入する操作を表す．$In\_Statespace$ は，Statespace V 上に s' が含まれるときに $True$ となる関数である．$Pop\_Stack(D)$ は，D に直近に追加された要素を削除する操作である．

アルゴリズム 1 Depth-First Search Based Algorithm

```
1:  Initialize: Stack D ← ∅, Statespace V ← ∅
2:  procedure Start
3:      Add_Statespace(V, A.s_0)
4:      Push_Stack(D, A.s_0)
5:      Search()
6:  procedure Search
7:      s ← Top_Stack(D)
8:      for all (s, l, s') ∈ A.T do
9:          if In_Statespace(v, s') = false then
10:             Add_Statespace(V, s')
11:             Push_Stack(D, s')
12:             Search()
13:     Pop_Stack(D)
```

■例　図 2.6に対して DFS を行う場合，探索木は以下の通りである．

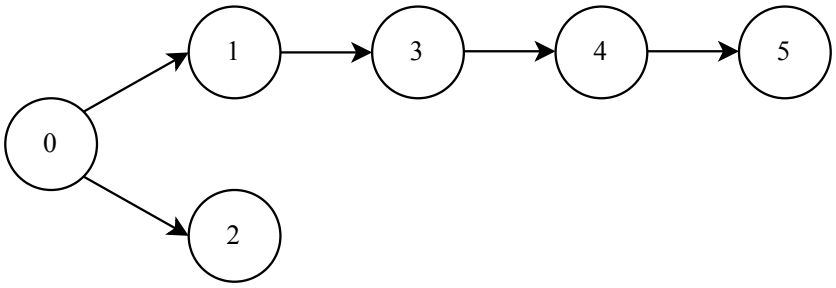


図 2.7　図 2.6に対する深さ優先探索の探索木

2.5.2　Stateless Depth-First Search

Stateless Depth-First Search は，探索全体で訪問済み状態の情報を共有せずに行う深さ優先探索である．擬似コードアルゴリズム 2は [5] から引用している．

DFS とは異なり，探索全体で訪れた状態を格納する変数 *Statespace* を持たない．また，8 行目で，*In_Statespace* の代わりに，*In_Stack* を実行している．*In_Stack* は，Stack D 上に要素 $(l,(l,s'))$ が含まれるときに *True* となる関数である．

Stack には，遷移とその遷移先の状態を格納している．*Add_statespace* は，V に状態を追加する操作である．また，*Push_Stack* は Stack D に遷移と状態の組から構成される要素を追加する操作である．$Top\_Stack(D)$ は，D に直近に追加された要素を得る操作である．D 自体に変更は加えない．一方で，$Pop\_Stack(D)$ は，D に直近に追加された要素を削除する操作である．

アルゴリズム 2 Stateless Search Based Algorithm

```
1:  Initialize: Stack D ← ∅
2:  procedure Start
3:      Push_Stack(D, A.s_0)
4:      Search()
5:  procedure Search
6:      s ← Top_Stack(D)
7:      for all (s, l, s') ∈ A.T do
8:          if In_Stack(D, s') = false then
9:              Push_Stack(D, s')
10:             Search()
11:     Pop_Stack(D)
```

■**例**　図 2.6に対して Stateless Search を行う場合，探索木は以下の通りである．

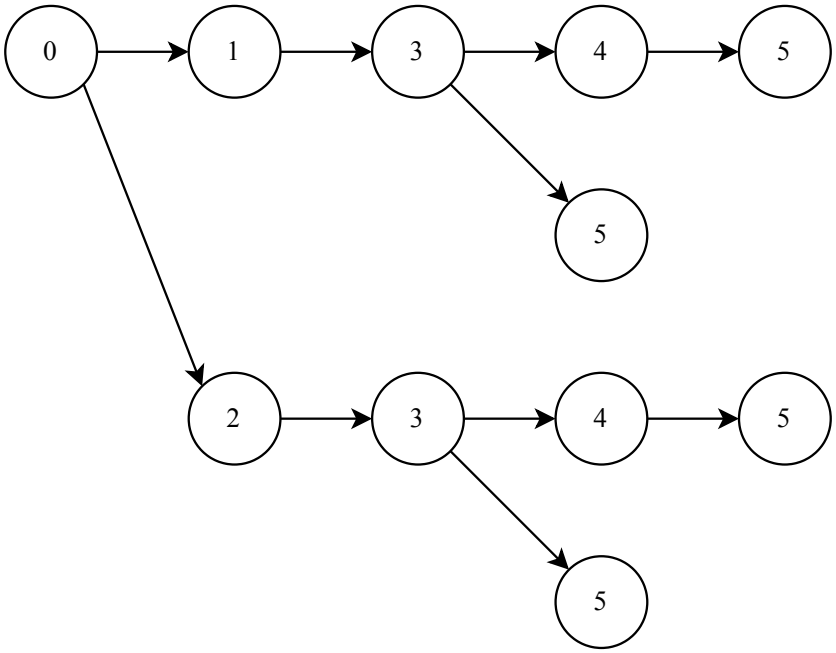


図 2.8　Stateless Search の探索木の例

図 2.9　図 2.6に対する Stateless Search の探索木

2.6　テスト

2.6.1　定義

テストは対象とするソフトウェアやシステムが意図した通りに動作する確証を得るために，テストプログラムなどを実行することである．

テストの際に使用する入力と期待値（望ましい出力）の組をテストケースと言う．テスト対象のシステムに入力を与えた結果，得られる出力が期待値と一致している場合，テストケースが Pass したと言う．また，一致しなかった場合，テストケースが Fail したと言う．

複数のテストケースをまとめたものをテストスイートと言う．テストを行う際に複数のテストケースをまとめて使用する必要がある際にこの用語を用いる．

2.6.2　API

API（Application Programming Interface）とは，アプリケーション間の連携を行うためのプロトコルやコマンドなどのインターフェイスである．POSIX では，図 2.10のようにユーザアプリケーションから OS へ要求を行うために API が定義されている．

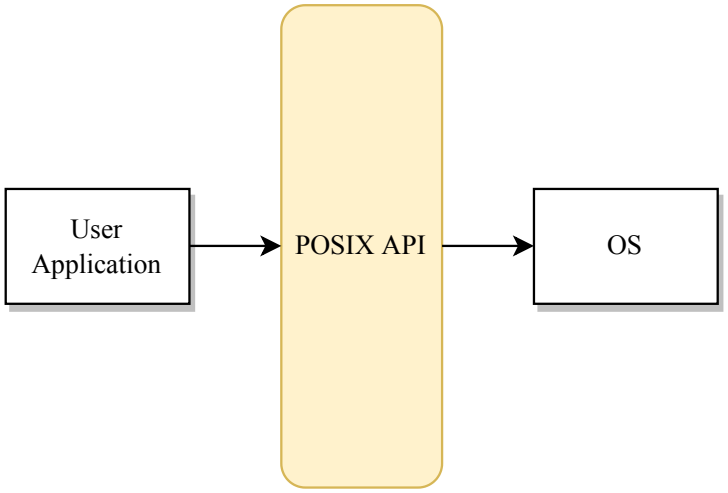


図 2.10　POSIX の API の役割

API は，ステートレスとステートフルの 2 つに大別することができる．ステートレスな API は状態を持たず，どのような場合に実行したとしても，同じ結果が得られる API である．例えば，POSIX の `gethostname` はステートレスな API である．この関数は，どのような状況で呼び出されたとしても，その計算機のホスト名を出力し返すため，状態を持たない．

一方で，ステートフルな API は状態を持ち，その状態に応じて実行結果が変化する．例えば，POSIX の `pthread_mutex_lock` と `pthread_mutex_unlock`（排他制御のための関数）はステートフルな API である．`pthread_mutex_lock` は，最初の呼び出し時には何も起こらないが，2 度目の呼び出し時には，1 度目に実行したスレッドが `pthread_mutex_unlock` を実行するまで待機する（実行完了しない）．このように，これまでの API 呼び出しが実行結果に影響を与える API はステートフルであると言う．ステートフルな API をテストする場合，API の実行順により実行結果が変化するため，様々な順序で API を呼び出すようなテストを実施する必要がある．そのような場合，テストケースは単一の入力と期待値の組ではなく，入力と期待値の列とすることがある．

2.6.3　モデルベーステスト

図 2.11のような状態遷移モデルを用いて仕様に規定されている振る舞いを記述し，そのモデルからテストケース・テストスイートを生成してテストを行う手法をモデルベーステスト（Model-based testing; MBT）と言う．ステートフルな API のテストを行う場合，状態によって動作が変化するため，MBT が利用される．図 2.11の状態遷移を API の実行だと捉えると，初期状態の S_0 にて，API の A と D を実行すると最終的に状態 S_2 に到達する．

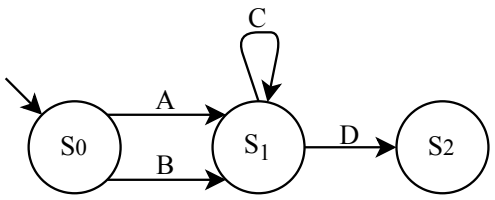


図 2.11　状態遷移モデルの例

図 2.11の状態を網羅するようにテストケースを作成する場合，深さ優先探索やその派生の探索アルゴリズムが使用される．探索アルゴリズムについては， 2.5 節で扱う．図 2.11に対して深さ優先探索を行うと，$(␣, S_0); (A, S_1); (D, S_2)$ というテストケースが 1 つ生成される．これは，入力がない状態では初期状態の S_0 に，A を入力すると S_1 へ遷移し，続けて D を入力すると S_2 となるという意味である．各入力を与えた直後に期待値と現状態が一致しているかを確認し，全て一致していればテスト結果は Pass，一つでも一致していなければテスト結果は Fail となる．

ここで，OSEK/VDX 準拠 OS のスケジューラに対するモデルベーステストを行った研究 [3] を事例として紹介する．[3] では，スケジューラの振る舞い仕様をモデル検査ツール SPIN [6] の仕様記述言語である Promela で記述し，MBT を行っている．Spin の機能を利用して Promela の記述をオートマトンに変換し，深さ優先探索アルゴリズムで状態網羅を行っている．Spin には，状態探索時の探索ログを表示する機能があるため，これを利用して探索木を作成し，そこから状態網羅を行うテストケースを作成している．そして，作成されたテストケースをテストプログラムに変換し，シミュレータ上で動作させることでテストを行っている．

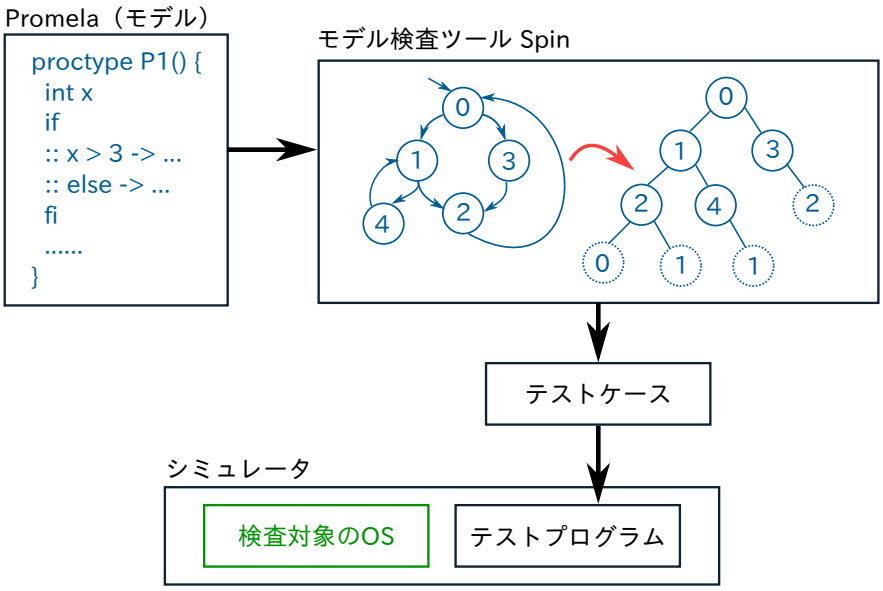


図 2.12　OSEK/VDX 準拠 OS に対する MBT の例

2.7　排他制御

並列プログラミングを行う際に，大域変数を扱う場合は排他制御を行う必要がある．次のコードのうち，関数 thread1 と関数 thread2 を並列動作させることを考える．

```
1  int ct = 0;
2
3  void thread1() {
4    ct += 30;
5  }
6
7  void thread2() {
8    ct += 50;
9  }
```

この 2 つの関数を逐次実行した場合は，変数 ct の値は 80 となる．しかし，並列動作時に次の順序で処理が行われると，thread1 が行った ct への加算が反映されず，最終的に ct の値は 50 となってしまう．ただし，矢印は代入を示し，t0 や t1 は計算に使用しているレジスタを表す．

1. thread1: t0 ← ct (load)
2. thread2: t1 ← ct (load)

3. thread1: ct ← t0 + 30 (store)
4. thread2: ct ← t1 + 50 (store)

このような値の破壊を防ぐために Mutex などの排他制御機構を使用する．ここでは，Mutex を取得する関数を lock，開放する関数を unlock とする．同じ Mutex（下の例では mtx）に対する lock は同時に 1 つのスレッドのみが実行可能であり，既に他のスレッドが lock を実行していた場合は，先に lock を実行したスレッドが unlock を実行するまで待機状態となる．そのため，次のコードでは，ct に対する加算を 2 つのスレッドが同時に行わなくなるため，上述の順での実行が起こらなくなり，常に正しい計算がされるようになる．

```
1  int ct = 0;
2  mutex_t mtx;
3
4  void thread1() {
5    lock(&mtx);
6    ct += 30;
7    unlock(&mtx);
8  }
9
10 void thread2() {
11   lock(&mtx);
12   ct += 50;
13   unlock(&mtx);
14 }
```

POSIX では Mutex が提供されている．POSIX の Mutex は Mutex Type と Robustness を設定することにより動作が変化する．動作に関してまとめられた表を POSIX [1] pthread_mutex_lock より引用する（表 2.1）．

Mutex Type	Robustness	Relock	Unlock When Not Owner
NORMAL	non-robust	deadlock	undefined behavior
NORMAL	robust	deadlock	error returned
ERRORCHECK	either	error returned	error returned
RECURSIVE	either	recursive	error returned
DEFAULT	non-robust	undefined behavior	undefined behavior
DEFAULT	robust	undefined behavior	error returned

表 2.1　Mutex の種類と動作 [1]

実装では，NORMAL と DEFAULT を区別していないことが多い．これは，リスト 2.1を実際に動作させることで確かめられる．

```
1  #include <pthread.h>
2  #include <stdio.h>
```

```
3
4  int main(void) {
5    printf("PTHREAD_MUTEX_NORMAL: %d\n",
6      PTHREAD_MUTEX_NORMAL);
7    printf("PTHREAD_MUTEX_ERRORCHECK: %d\n",
8      PTHREAD_MUTEX_ERRORCHECK);
9    printf("PTHREAD_MUTEX_RECURSIVE: %d\n",
10     PTHREAD_MUTEX_RECURSIVE);
11   printf("PTHREAD_MUTEX_DEFAULT: %d\n",
12     PTHREAD_MUTEX_DEFAULT);
13 }
```

コード 2.1　Mutex の実装の調査用コード

本研究で Mutex を扱う際は，最も一般的だと予想される設定として，Mutex Type を NORMAL に，Robustness を non-robust と設定してあると仮定している．

2.8　擬似コード

擬似コードは [7] で定義されている Informal Compiler Algorithm Notation (ICAN) を使用する．ICAN は列（sequences）やレコード（records）などの基本的なデータ構造や for や while などの文に関する記法を持っている．また，集合などの数学的な記法にも対応している．[7] にてよく説明されており，通常の和演算 (+) と列の連結 (⊕) を別記号で表しているなど，分かりやすい記法となっている．

オリジナルの ICAN では，例えば integer の列を sequence of integer と表記する．しかし，本稿では紙面の都合で seq of Int と省略して表記する．また，オリジナルでは関数の引数の型に in (call by value)，out (call by result)，inout (call by value-result) の palameter declarations が明示されている．しかし，本稿ではこれを省略し，全ての変数を値渡し (call by value) として扱う．

[7] のうち，本稿で使用する型のコンストラクタを表 2.2に示す．

表 2.2　Type Constructors [7]

Constructor	Name	Example Declaration
enum	列挙型	enum left, right
set of	集合	set of $\mathbb{N}$
seq of	列	seq of Bool
×	組	$\mathbb{N} \times \mathbb{N}$
record	レコード型	record x: Int, y: Int
→	関数	Int → Bool

各型の値の例を表 2.3に示す.

表 2.3　Constants of constructed types [7]

Type	Example Constant
seq of Int	[2,3,5,7,9]
$\mathbb{N}$×Bool	<3,true>
set of (Int×Bool)	<3,true>,<1,false>
record x: Int, y: Int	<x:3,y:8>
$\mathbb{N} \times \mathbb{N}$ →Bool	<1,1,true>,<2,5,false>

レコード型の変数 rec に対して値を代入するとき，以下のように表す．:=が代入演算子であり，後ろのコロン以降が型である．

```
rec := <x: 1,y: 3>: record {x: Int, y: Int}
```

また，二項演算子については，以下の 2 文を同じ意味として扱う．

```
i := i + 8
i += 8
```

本稿で使用する複合文を表 2.4に示す.

表 2.4　Compound statements [7]

Beginning	Internal	Ending
if	elif,else	fi
for	do	od
while	do	od

■組　組は前述の通り，× を使用して $\mathbb{N} \times \mathbb{N}$ のように記述する．また，値は<1,3>のように<>を使用して表記する．n 番目の値を指す際は@を使用して，例えば<1,3,5>@2 = 3 というように記述する．

■レコード型　レコード型は record { ... }と表記し，例えば，名前と年齢と身長を表す型は，record {name: String, age: $\mathbb{N}$, height: $\mathbb{R}$}と書く．また，その値は<>を使用して，<name:"John",age:31,height:175.0>と表記する．このとき，要素は順不同とし，<name:"John",height:175.0,age: 31> = <name:"John",age:31,height:175.0>である．

■列　空列は [] と表す．また，列の連結演算子は ⊕ を使用し，[1,3]⊕[5,7]=[1,3,5,7] と表記する．列から n 番目の要素を削除する演算子は ⊖ を使用し，[1,3,5]⊖2=[1,5] とする．列の n 番目の要素を得る演算子として ↓ を使用し，[1,3,5]↓2=3 と表記する．n として負の数値を与えた場合は，後ろから n 番目の要素を指すこととし，[1,3,5]↓-1=5 と使用する．オリジナルの ICAN では配列のための記法として定義されているが，本稿では可読性を考慮して，列に対して以下のように [] を使用した表記を導入する．

```
a := [2,3,5,6,9]: seq of ℕ
a↓3 = 5 = a[3]
a↓-2 = 6 = a[-2]
```

■集合　空集合は ∅ と表す．また，集合の内包表記は以下のように書く．

```
s := {n where ∀n ∈ ℕ (1 ≥ n ∧ n ≥ 10 ∧ n%2 = 0)} = {2,4,6,8,10} : set of ℕ
```

集合に含まれているかを判定する演算子は ∈ を使用し，その否定は ∉ を使用する．

また，集合から要素をランダムに一つ選ぶ演算子として ♦ を導入し，♦{1, 3, 5} とした場合は，1,3,5 のどれかの値を指す．この演算子は集合から要素を取り出して操作を行う場合に使用する．例えば，以下の例では A の集合の要素を全て 2 倍した集合 B を生成している．

```
A := {1,3,5}: set of ℕ
B := ∅: set of ℕ
Tmp := A: set of ℕ
while Tmp ≠ ∅ do
  a := ♦Tmp
  B ∪= {2*a}
  Tmp -= {a}
od
/* B = {2,6,10} */
```

便宜上，上のコードと同じ操作を予約語 each を使用して以下に示すように記述することもある．この場合，for each a∈A では，A のランダムな要素を a として取り出すことを表す．

```
A := {1,3,5}: set of ℕ
B := ∅: set of ℕ
for each a∈A do
  B ∪= {2*a}
od
/* B = {2,6,10} */
```

■サイズ演算子　集合，列などの constructed types 対して演算子| |を使用可能とする．以下に演算の例を示す．

```
|{1,3,5}| = 3
|[[1,3,5,7],[3],[1,5,7]]| = 3
|<1,5,8>| = 3
|<x:3,y:8>| = 2
|{x where ∀x ∈ ℕ (0 < x ∧ x < 3)}| = |{1,2}| = 2
```

■関数　関数は，以下のように記述する．関数の引数の型は procedure と begin の間に記す．また，返り値の型は returns の後に書く．また，値を返す場合は予約語 return を使用する．

```
procedure sum(xs) returns Int
  xs: seq of Int
begin
  s := 0: Int
  for i := 1 to |xs| do
    s += xs[i]
  od
  return s
end
```

■for 文　for 文は基本的に以下の記法で表し，変数 i を $1 \geq n \geq 10$ の範囲で 10 から順に変化させる．by -1 は 1 ずつ値を減らすことを意味し，省略した場合は by 1 と見做す．

```
for i := 10 by -1 to 1 do
  for_body
od
```

第 3 章

POSIX の形式化

3.1 概要

POSIX は自然言語で記述されている．そのため，POSIX からテストケースを自動生成するためには，形式化を行う必要がある．POSIX には実装に用いるデータ構造については指定されていないため，[8] を参考にデータ構造を作成し，それらを用いて POSIX を満たす動作を形式化した．本稿では，API として，使用される頻度の高い，スレッド生成・終了に関する API (pthread_create, pthread_exit) と排他制御の関する API（pthread_mutex_lock, pthread_mutex_trylock, pthread_mutex_unlock) を扱う．本章では，POSIX のスケジューラの動作と各 API の形式化について導入を行う．

3.2 優先度付きキューの形式化

POSIX のスケジューラの動作を形式化する際に優先度付きキューを多用する．そのため，本節で優先度付きキューを形式化し，導入する．本稿では優先度付きキューを各優先度に対応する複数のキューから構成されるキューとして形式化する（図 3.1）．優先度は数値が大きいほど優先度が高いとする．

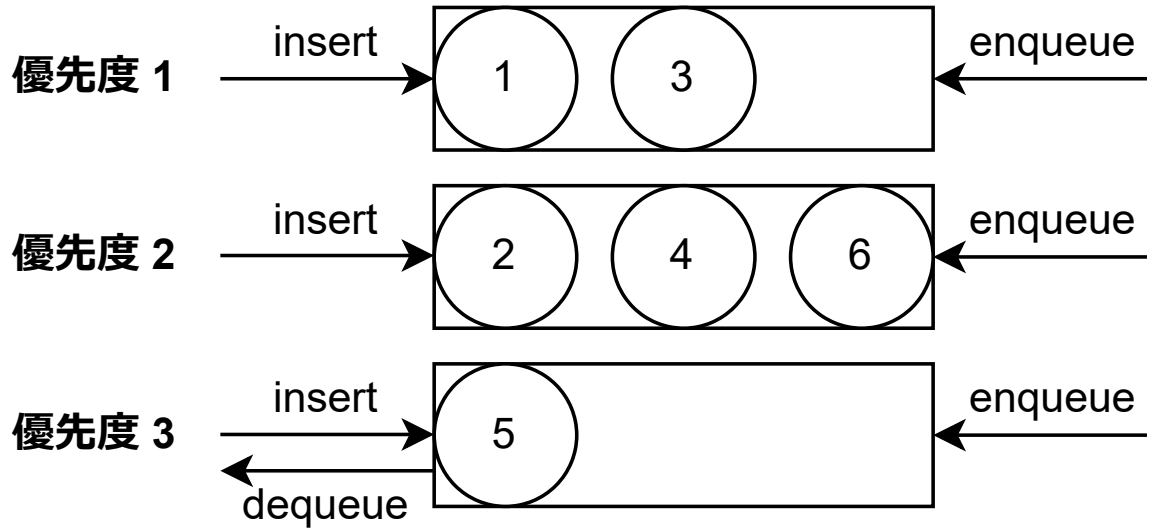


図 3.1　優先度付きキュー

本稿では，優先度付きキューの要素は自然数のみとし，次式のように自然数の列の列として捉える．

```
PriorityQueue = seq of seq of ℕ
```

PriorityQueue に対して以下に示す関数を使用する．

■head　優先度付きキューの中で最高優先度の要素を返す関数．優先度付きキューが空の場合は – を返す．図 3.1では，要素 5 が最高優先度の優先度 3 のキューの先頭にあるため，head で 5 が返る．

```
procedure head(prio_queue) returns ℕ∪{-}
  prio_queue: PriorityQueue
begin
  for i := |prio_queue| by -1 to 1 do
    que := prio_queue[i]: seq of ℕ
    if que ≠ [] then
      return que[1]
    fi
  od
  /* should not reach here */
  return -
end
```

■is_empty　優先度付きキューが空であるかを調べる関数．空の場合は True を返す．図 3.1では，キュー内に要素があるため False が返る．

```
procedure is_empty(prio_queue) returns Bool
  prio_queue: PriorityQueue
begin
```

```
4     for i := 1 to |prio_queue| do
5       if prio_queue[i] ≠ [] then
6         return false
7       fi
8     od
9     return true
10  end
```

■insert　引数で与えられた優先度 prio に対応するキューの先頭に要素 n を追加し，追加後の優先度付きキューを返す．一般的にはキューの先頭に要素を追加することはないが，スケジューリングでは高優先度のスレッドが実行開始した際に実行中の低優先度のスレッドが対応する優先度のキューの先頭に戻されるため，insert が必要となる．図 3.1で優先度 3 の要素 6 を追加すると，最高優先度（優先度 3）のキューの先頭が 6 になる．

```
1   procedure insert(prio_queue, n, prio) returns PriorityQueue
2     prio_queue: PriorityQueue
3     n: ℕ
4     prio: ℕ
5   begin
6     prio_queue[prio] := [n]⊕prio_queue[prio]
7     return prio_queue
8   end
```

■enqueue　引数で与えられた優先度 prio に対応するキューの末尾に要素 n を追加し，追加後の優先度付きキューを返す．図 3.1で優先度 3 の要素 6 を追加すると，最高優先度（優先度 3）のキューの先頭は 5 のままであるが，末尾が 6 になる．

```
1   procedure enqueue(prio_queue, n, prio) returns PriorityQueue
2     prio_queue: PriorityQueue
3     n: ℕ
4     prio: ℕ
5   begin
6     prio_queue[prio] := prio_queue[prio]⊕[n]
7     return prio_queue
8   end
```

■dequeue　引数で与えられた優先度付きキューから最高優先度の要素を取り除いた優先度付きキューを返す．ただし，同優先度の要素が複数ある場合は，その優先度に対応するキューの先頭の要素を取り除く．引数として与えられた優先度付きキューが空の場合，取り除く要素がない．このような場合での dequeue の使用は想定していない．図 4.2では，要素があるキューの最高優先度が 3 であり，それに対応するキューの要素は 5 であるため，5 が取り除かれた優先度付きキューが返る．

```
1   procedure dequeue(prio_queue) returns PriorityQueue
2     prio_queue: PriorityQueue
3   begin
4     for i := |prio_queue| by -1 to 1 do
```

```
5       que := prio_queue[i]: seq of ℕ
6       if que ≠ [] then
7         prio_queue[i] = prio_queue[i]⊖1
8         return prio_queue
9       fi
10    od
11    /* should not reach here */
12    return prio_queue
13  end
```

3.3 スケジューラの形式化

3.3.1 アプローチ

本節では，POSIX スケジューラの動作の形式化を行う．本研究では，テストケースにて実行結果の不確定性を明示的に扱いたいため，それを考慮した形式化を行う．実行結果の不確定性とは，前述の通り，API を実行した際に複数の実行結果が考えられるという性質である．本稿では，非決定性有限オートマトン (NFSA) の非決定性遷移が実行結果の不確定性と同様の性質を持つことに着目し，NFSA をベースにテスト生成に必要なパラメータを追加することで，スケジューラの動作の形式化を行った．

本稿では，リアルタイム OS として POSIX 準拠 OS を使用することを想定しているため，スケジューリングポリシーとして SCHED_FIFO を仮定している．

3.3.2 NFSA を用いたスケジューラの形式化

$SchedModel = (S, \Sigma, \delta), where$

- S はスケジューラの状態の有限状態
- $\Sigma = API \times list\ of\ \mathbb{N}$ は POSIX の API 名と引数の組の集合
- $\delta : S \times \Sigma \rightarrow 2^S$ は API の実行による状態遷移

スケジューラの状態の定義

上述の API の形式化やスケジューリングに必要な情報をスケジューラの状態として定義する．それぞれ，必要なデータと本稿での呼称は以下のとおりである．

- スレッドの生成・終了に関する API
  - ThreadState: スレッドの状態 (Running, Ready, Waiting, etc.)
  - Priority: スレッドの優先度

  - TID: スレッド ID
  - Ready Queue (RQ) : Ready 状態のスレッドの優先度付き待ち行列
- 排他制御に関する API
  - MutexOwner: Mutex の所持者（解放時は所持者なし）
  - Waiting Queue (WQ) : Mutex 取得待ちのスレッドの優先度付き待ち行列

上述のデータを要素として持つ集合をスケジューラの状態として定義し，OsState と呼ぶこととする．

各データの本稿での定義を以下に示す．

■ThreadState　スレッドの状態であり，以下のいずれかの値を取る．

- Not Created: 未作成
- Ready: 実行待ち
- Running: 実行中
- Waiting: リソース取得待ち (本稿では Mutex のみ)
- Terminated: 終了済み

ThreadState は図 3.2に示す状態遷移を行うとする．これは [9] を参考に作成した．

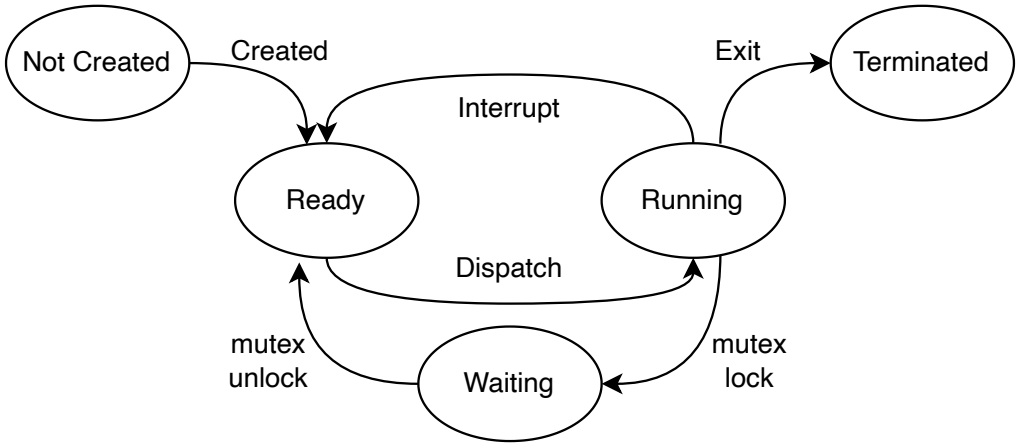


図 3.2　ThreadState の状態遷移

■Priority　スレッドの優先度を表す．SCHED_FIFO では 1 以上の整数値が優先度として使用されている．そのため，本稿の形式化においても，Priority は 1 以上の整数値であるとする．また，各スレッドの Priority は不変とする．

■TID　スレッド ID を表す 1 以上の整数である．各スレッドを区別するために使用する．本稿では，各スレッドはユニークな TID を持つと考える．

■Ready Queue (RQ)　Ready 状態のスレッドの優先度付き待ち行列である．要素は $\mathbb{N}$ であり，Ready のスレッドの TID を格納する．RQ で使用する優先度は前述した各スレッドの Priority である．

■MutexOwner　Mutex の所持者の TID である．値は $\mathbb{N} \cup \{-\}$ の要素であり，Mutex が解放されている（所持者がいない）場合は MutexOwner = -と書く．

■Waiting Queue (WQ)　Waiting 状態のスレッドの優先度付き待ち行列である．要素は $\mathbb{N}$ であり，Waiting のスレッドの TID を格納する．RQ で使用する優先度は前述した各スレッドの Priority である．

3.4　API の形式化

3.4.1　アプローチ

3.3 節では，NFSA としての形式化を行う方法について概要を述べた．そこでは，API は NFSA の語としてのみ導入していた．しかし，テストでは，(1) 実行結果が仕様上，定義されていない状態/引数での API 実行や (2) 入力されることがない引数での API 呼出しの 2 つのケースはテスト対象外としたい．この 2 つを自動生成されるテストケース群から除外するために，以下の 3 つの要素を持つレコード型として API を形式化する．

```
1 API = record {
2   arguments: OsState → seq of ℕ×ℕ,
3   requirement: seq of ℕ × OsState → Bool,
4   execute: seq of ℕ × OsState → set of OsState,
5 }
```

arguments は，形式化した API の引数のアリティと各値が取り得る最小値・最大値を表す．引数が 2 つであれば，$\langle\langle 1, 10\rangle, \langle 0, 3\rangle\rangle$ のように，2 つの組から為る列となる．例えば，スレッド生成の API である pthread_create で優先度-1 のスレッドを生成しようとした場合，優先度はどのようなスケジューリングポリシーであっても，0 以上でなければならないと POSIX で規定されているため，実行してもエラーが生じる．この仕様を arguments で表し，このようなエラーをテスト対象外とする．arguments で定めた引数の数（アリティ）と範囲は execute と requirement の引数 (seq of $\mathbb{N}$) で使用する．

requirement は実行結果が仕様上，定義されていない状態での API 実行や，実行結果が

未定義となる値を引数として与えられた場合の API 実行を除外するために使用する．現状態と API への引数を受け取り，実行結果が仕様上定義されていれば True を返し，未定義であれば False を返す．このとき，引数は arguments で定めたものであるとする．例えば，POSIX 準拠 OS では排他制御の Mutex を取得する際に，pthread_mutex_lock を実行するが，デフォルトの設定では Mutex を取得したスレッドがこの API を使用することは想定されておらず，POSIX 上で未定義動作となると書かれている．実際にこの動作を Linux で試したところデッドロックを引き起こした．このような未定義動作を表現し，テスト対象外とするために requirement を使用する．

execute は NFSA の遷移を表しており，API の引数 (seq of $\mathbb{N}$) と現状態を受け取り，状態の集合を返す．このとき，引数は arguments で定めたものであるとする．

3.4.2 スレッド生成・終了

pthread_create

pthread_create は，新規スレッドの作成・実行開始を行う API である．新規スレッド作成時にスレッドの実行開始は行わず，別の API を呼び出すことで実行開始を行う OS もあるが，POSIX では 1 つの API (pthread_create) で作成と実行開始処理が行われる．

POSIX に規定されている pthread_create では，引数としてスケジューリング属性を表す構造体を受け取る．この構造体を使って優先度やスケジューリングポリシーなどを設定する．しかし，本研究の形式化では，各 API の引数の型を統一したいため，整数列の列（seq of seq of $\mathbb{N}$）を引数として取り，これにより，優先度を設定するように形式化を行っている．本稿では pthread_create の仕様は次のレコード型として形式化する．各要素の説明についてはそれ以降で順に述べる．

```
1 PthreadCreate := <arguments: pc_arg,requirement: pc_req,execute: pc_exec>
```

■arguments　形式化した pthread_create で使用する引数のアリティと値の範囲の定義である．左から順に (1)API 呼び出し元の TID，(2) 新規作成する TID，(3) 新規作成するスレッドの優先度を表している．MaxPrio と MaxTid はテスト時に決定する値であり，スレッドの最大優先度と最大スレッド数を意味している．

```
1 procedure pc_arg(os_state) returns seq of ℕ×ℕ
2   os_state: OsState
3 begin
4   return [<1,MaxTid>,<1,MaxTid>,<1,MaxPrio>]
5 end
```

■requirement　requirement では，pthread_create の実行結果が未定義となる場合に False を返し，定義される場合に True を返す．本稿では TID の再利用は考えておらず，また，POSIX が TID の重複を許していないため，新規作成するスレッドは NotCreated (未作成) であるはずである．そのため，TCB で新規作成するスレッドの状態が NotCreated となっていることを要求している．また，実行中 (Running) でないスレッドは API を呼び出せないため，呼び出し元のスレッドが Running であることを要求している．

```
procedure pc_req([tid,ttid,tprio],os_state)
  [tid,ttid,tprio]: seq of ℕ
  os_state: OsState
begin
  return os_state.tcb[tid].state = Running && os_state.tcb[ttid].state = NotCreated
end
```

■execute　pthread_create を呼び出した際の動作の形式化である．API 実行時には，引数で与えられた TID のスレッドの状態を NotCreated から Ready に変更し (図 3.2参照)，RQ に追加している．また，今回は正常動作の場合を形式化しているため，実行後の errno は 0 にしている．

```
procedure pc_exec([tid,ttid,tprio],os_state) returns set of OsState
  [tid,ttid,tprio]: seq of ℕ
  os_state: OsState
begin
  os_state.errno := 0
  os_state.tcb[ttid].state := Running
  os_state.tcb[ttid].prio := tprio
  return {os_state}
end
```

■例　以下のような状態であると仮定する．

```
os_state: OsState = <tcb: [<1,1,Running>,<2,1,Ready>,<3,0,NotCreated>], rq: [[2],[]],
  mutex_owner: -, wq: [[],[]], errno: 0>
```

このとき，引数として [1,3,1] を与えて PthreadCreate を実行すると，以下に示すように，TID 3 が NotCreated から Ready となり，優先度が 1 のキューに追加される．また，errno が 0 となる．

```
pc_exec([1,3,1],os_state) =
  {<tcb: [<1,1,Running>,<2,1,Ready>,<3,1,Ready>],rq: [[2,3],[]],mutex_owner: -, wq:
    [[],[]], errno: 0>}
```

pthread_exit

pthread_exit を呼び出したスレッドを終了させる API である．pthread_exit は，スレッドの返り値を引数として受け取ったポインタに格納するが，本研究では返り値に対するテストは行わないため，省略している．形式化した pthread_exit の仕様は以下である．

```
PthreadExit := <arguments: pe_arg,requirement: pe_req,execute: pe_exec>
```

■arguments　どのスレッドが API を呼び出したかを区別するために，呼び出し元のスレッドの TID を引数として渡す API として形式化を行っている．

```
procedure pe_arg(os_state) returns seq of ℕ×ℕ
  os_state: OsState
begin
  return [<1,MaxTid>]
end
```

■requirement　実行中 (Running) のスレッドのみが API を呼び出せるため，呼び出し元のスレッドが Running であることを要求している．

```
procedure pe_req([tid],os_state)
  [tid]: seq of ℕ
  os_state: OsState
begin
  return os_state.tcb[tid].state = Running
end
```

■execute　pthread_exit は呼び出し元のスレッドを終了させる API であるため，呼び出し元のスレッドの状態を Running から Terminated にする動作として形式化している（図 3.2参照）．

```
procedure pe_exec([tid],os_state) returns set of OsState
  [tid]: seq of ℕ
  os_state: OsState
begin
  os_state.errno := 0
  os_state.tcb[tid].state := Terminated
  return {os_state}
end
```

■例　以下のような状態であると仮定する．

```
os_state := <tcb: [<1,1,Running>,<2,1,Running>],rq: [[]],errno: 0,mutex_owner: -,wq:
  [[]],config: conf>: OsState
```

このとき，引数として TID 2 を与えて PthreadExit を実行すると，以下に示すように，TID 2 が終了し Terminated となる．

```
pe_exec([2],os_state) = {<tcb: [<1,1,Running>,<2,1,Terminated>],rq: [[]],errno: 0,
  mutex_owner: -,wq: [[]],config: conf>}
```

3.4.3 排他制御 (Mutex)

pthread_mutex_lock

pthread_mutex_lock は Mutex を取得する API である．POSIX [1] 中，B.2 General Information の Spin Locks versus Mutexes において，Mutex が取得不可能であるときは，スレッド自身の状態を Waiting にすると書いてある．また，[8] では，Waiting 状態のスレッドの管理は Wait Queue (WQ) と呼ばれるキューを用いて行われている．これらを参考にし，Mutex の取得待ち順を優先度付きキュー（WQ) で管理する方式で形式化を行っている．形式化された pthread_mutex_lock を以下に示す．

```
PthreadMutexLock := <arguments: pml_arg,requirement: pml_req,execute: pml_exec>
```

■arguments　本稿では，Mutex は簡単のため，1 つのみであると仮定しているため，対象とする Mutex を指定する引数は必要ない．そのため，引数は API を呼び出したスレッドの TID のみを引数として取る．

```
procedure pml_arg(os_state) returns Bool
  os_state: OsState
begin
  return [<1,MaxTid>]
end
```

■requirement　API を呼び出すスレッドは Running（動作中）でなければならない．また，デフォルトの設定では，Mutex を取得しているスレッドが再度 Mutex の取得を試みると未定義動作となると POSIX に記述されているため，現在の Mutex 所持者がこの API を呼び出すことを禁じている．

```
procedure pml_req([tid],os_state)
  [tid]: seq of ℕ
  os_state: OsState
begin
  return os_state.tcb[tid].state = Running && os_state.mutex_owner ≠ tid
end
```

■execute　API 呼び出し時に Mutex が解放されている場合は API を呼び出したスレッドが Mutex を取得する（mutex_owner を API を呼び出したスレッドの TID に変更する)．Mutex が他のスレッドによって取得済みである場合は，Mutex が解放されるのを待つために自身をキュー（Waiting Queue）に追加し，スレッドの状態を Waiting にする．

```
procedure pml_exec([tid],os_state) returns set of OsState
  [tid]: seq of ℕ
  os_state: OsState
begin
  if os_state.mutex_owner = - then
    os_state.mutex_owner := tid
    os_state.errno := 0
  else
    prio := os_state.tcb[tid].prio: ℕ
    os_state.wq := enqueue(os_state.wq,tid,prio)
    os_state.tcb[tid].state := Waiting
  fi
  return {os_state}
end
```

■例 1　Mutex が解放されている次の状態を考える．

```
os_state: OsState = <tcb: [<1,1,Running>,<2,1,Running>],rq: [[],[]],errno: 0,mutex_owner
  : -,wq: [[],[]] config: conf>
```

このとき，引数として TID 2 を与えて PthreadMutexLock を実行すると，以下に示すように，Mutex の所有者が TID 2 になる．これは，実際の OS 上の動作では，TID 2 のスレッドが pthread_mutex_lock を実行した状態に対応する．

```
pml_exec([2],os_state) = {<tcb: [<1,1,Running>,<2,1,Running>],rq: [[]], errno: 0,
  mutex_owner: 2, wq: [[]], config: conf>}
```

■例 2　次のように Mutex が TID 1 によって取得されている場合を考える．実 OS では，TID 1 のスレッドが pthread_mutex_lock を実行した後の状態に対応する．

```
os_state': OsState = <tcb: [<1,1,Running>,<2,1,Running>],rq: [[]],errno: 0,mutex_owner:
  1,wq: [[]] config: conf>
```

この場合，TID 2 が Mutex の取得を試みると以下のように Waiting となり，WQ に TID 2 が入る．これは，実 OS 上で，TID 2 に対応するスレッドが pthread_mutex_lock を実行した際の動作に対応する．

```
pml_exec([2],os_state') = {<tcb: [<1,1,Running>,<2,1,Waiting>],rq: [[]], errno: 0,
  mutex_owner: 1, wq: [[2]], config: conf>}
```

pthread_mutex_trylock

pthread_mutex_trylock は，Mutex が解放されている場合は，pthread_mutex_lock と同様に Mutex を取得する．Mutex が他のスレッドによって取得されている場合には errno として EBUSY を返し，Mutex が解放されるまで待機せずに，即時復帰する．以下に形式化した pthread_mutex_trylock を示す．

```
PthreadMutexTrylock := <arguments: pmt_arg,requirement: pmt_req,execute: pmt_exec>
```

■arguments　本稿では，Mutex は 1 つのみと仮定しているため，対象とする Mutex を指定する引数は必要ない．そのため，引数は API を呼び出したスレッドの TID のみである．

```
procedure pmt_arg(os_state) returns Bool
  os_state: OsState
begin
  return [<1,MaxTid>]
end
```

■requirement　pthread_mutex_lock と requirement と同様の理由により，(1)API を呼び出したスレッドが Running であり，(2)API を呼び出したスレッドが既に Mutex の取得者でないことを要求する．

```
procedure pmt_req([tid],os_state) returns Bool
  [tid]: seq of ℕ
  os_state: OsState
begin
  return os_state.tcb[tid].state = Running && os_state.mutex_owner ≠ tid
end
```

■execute　Mutex が解放されている場合は，pthread_mutex_lock と同様に Mutex を取得する．このとき，errno は正常に Mutex を取得出来ているため，0 となる．Mutex が他のスレッドによって取得されている場合は，pthread_mutex_lock とは異なり，errno を EBUSY に変更する．pthread_mutex_trylock は Mutex が取得できない場合でも待機しないため，スレッドの状態は変更しない．

```
procedure pmt_exec([tid],os_state) returns set of OsState
  [tid]: seq of ℕ
  os_state: OsState
begin
  if os_state.mutex_owner = - then
    os_state.mutex_owner := tid
    os_state.errno := 0
  else
    os_state.errno = EBUSY
  fi
  return {os_state}
end
```

■例　Mutex が解放されている状態で呼び出された場合は PthreadMutexLock と同様の動作をするため省略する．Mutex が以下のように取得済みである場合を考える（PthreadMutexLock の例で使用した状態を流用する）．

```
os_state': OsState = <tcb: [<1,1,Running>,<2,1,Running>],rq: [[]],errno: 0,mutex_owner:
  1,wq: [[]] config: conf>
```

この場合，TID 2 が Mutex を取得を試みると以下のように errno が EBUSY となる．ただし，PthreadMutexLock とは異なり，WQ に変更は加えられない．

```
pmt_exec([2],os_state') = {<tcb: [<1,1,Running>,<2,1,Running>],rq: [[]], errno: EBUSY,
  mutex_owner: 1, wq: [[]], config: conf>}
```

pthread_mutex_unlock

スレッドが所持している Mutex を解放する API である．形式化した pthread_mutex_-unlock の仕様を以下に示す．

```
PthreadMutexUnlock := <pmu_arg: arguments,pmu_req: requirement,pmu_exec: execute>
```

■arguments 本稿では，Mutex は 1 つのみであり，Mutex 自体を指定する必要がないため，API を呼び出したスレッドの TID のみを引数として取る．

```
procedure pmu_arg(os_state) returns Bool
  os_state: OsState
begin
  return [<1,MaxTid>]
end
```

■requirement デフォルト設定では API を呼び出したスレッドが所持していない Mutex の解放を試みると未定義動作が起こると POSIX に規定されているため，requirement で Mutex 所持者以外の実行を禁止している．また，他の API と同様の理由で，API を呼び出すスレッドは Running である必要がある．

```
procedure pmu_req([tid],os_state)
  [tid]: seq of ℕ
  os_state: OsState
begin
  return os_state.tcb[tid].state = Running && os_state.mutex_owner = tid
end
```

■execute Mutex を解放する（mutex_owner を-に設定する）．このとき，既に Mutex の取得待ちをしているスレッドがある場合は，そのスレッドを Mutex の所持者に設定し，そのスレッドの状態を Waiting から Ready に設定する．また，Ready 状態になったスレッドを ready queue (RQ) の末尾に追加する．

```
procedure pmu_exec([tid],os_state) returns set of OsState
  [tid]: seq of ℕ
  os_state: OsState
begin
  os_state.mutex_owner := -
  os_state.errno := 0
```

```
7    if is_empty(os_state.wq) = false then
8      tid := head(os_state.wq): ℕ
9      prio := os_state.tcb[tid].prio: ℕ
10     os_state.mutex_owner := tid
11     os_state.wq := dequeue(os_state.wq)
12     os_state.tcb[tid].state := Ready
13     os_state.rq := enqueue(os_state.rq,tid,prio)
14   fi
15   return {os_state}
16 end
```

■例　TID 1 によって Mutex が取得されている以下の状態を考える．

```
os_state: OsState = <tcb: [<1,1,Running>,<2,1,Waiting>],rq: [[]],errno: 0,mutex_owner:
  1,wq: [[2]] config: conf>
```

このとき，TID 1 が Mutex の解放を試みると以下のように mutex_owner が-になり，WQ に入っている TID 2 が Ready となる．

```
pmu_exec([2],os_state') = {<tcb: [<1,1,Running>,<2,1,Ready>],rq: [[2]], errno: 0,
  mutex_owner: -, wq: [[]], config: conf>}
```

3.4.4　スケジューラの動作

スケジューラがスレッドを CPU のコアへ割り当てる処理を本稿では，Dispatch と呼ぶ．POSIX の API ではないが，POSIX の API と同様のレコード型を用いて形式化を行った．スケジューリングポリシーは，組込み機器を想定し SCHED_FIFO であると仮定している．

Level1 と Level2 の 2 つの動作から構成されている．Level1 では，SCHED_FIFO で Preempt されたスレッドの仕様（ 2.4 節）に則り，Running の全てのスレッドを Ready Queue の先頭に戻し，一度 Ready にしている．そして，Level2 でコアに割り当てるスレッドを決定し，スレッドの状態を Running に変更する．

Level1 において，同優先度のスレッドが複数存在しているとき，Ready Queue へ戻す順番によって Level2 で選ばれるスレッドが変化することがある．Level1 ではキューの先頭に戻し，Level2 ではキューの先頭から順に Running にするため，Level1 でキューに戻されたスレッドよりも高優先度のスレッドが実行開始した場合，スケジューラ実行後の動作が複数考えられる．そのため，本稿では，Dispatch が実行結果の不確定性の発生源であると考えている．

```
Dispatch := <arguments: d_arg,requirement: d_req,execute: d_exec>

procedure d_arg(os_state) returns Bool
  os_state: OsState
```

```
begin
  return []
end

procedure d_req([],os_state) returns Bool
  []: seq of ℕ
  os_state: OsState
begin
  return true
end

procedure d_exec([],os_state) returns set of OsState
  []: seq of ℕ
  os_state: OsState
begin
  return level2(level1(os_state))
end

procedure level1(os_state) returns set of OsState
  os_state: OsState
begin
  Lvl1 := []: set of OsState
  xss:= set_to_seq({x where x ∈ os_state.tcb,x.state = Running}): seq of seq of ℕ
  for i := 1 to |xss| do
    xs := xss[i]: seq of ℕ
    os_state' := os_state: OsState
    for j := 1 to |xs| do
      tid := xs[j]: ℕ
      os_state'.tcb[tid].state := Ready
      prio := os_state'.tcb[tid].prio: ℕ
      os_state'.rq := insert(os_state'.rq,tid,prio)
    od
    Lvl1 ∪= os_state'
  od
  return Lvl1
end

procedure level2(Lvl1) returns set of OsState
  Lvl1: set of OsState
begin
  Lvl2 := []: set of OsState
  for each st∈Lvl1 do
    n_running := 0: Int
    while n_running ≠ st.config.core_num && is_empty(st.rq) = true do
      n_running += 1
      tid := head(st.rq): ℕ
      st.tcb[tid].state := Running
      st.rq := dequeue(st.rq)
    od
    Lvl2 ∪= st
  od
  return Lvl2
end
```

第 4 章

テスト

4.1　アプローチ

第 3 章の形式化を用いてテストスイートの自動生成をし，テストを行う．POSIX 準拠 OS には実行結果の不確定性があるため，図 4.1のように 1 つの API 呼び出し列を実行した際に 2 つの実行結果が考えられる．このような場合，テストの期待値は一意に定めることができない.

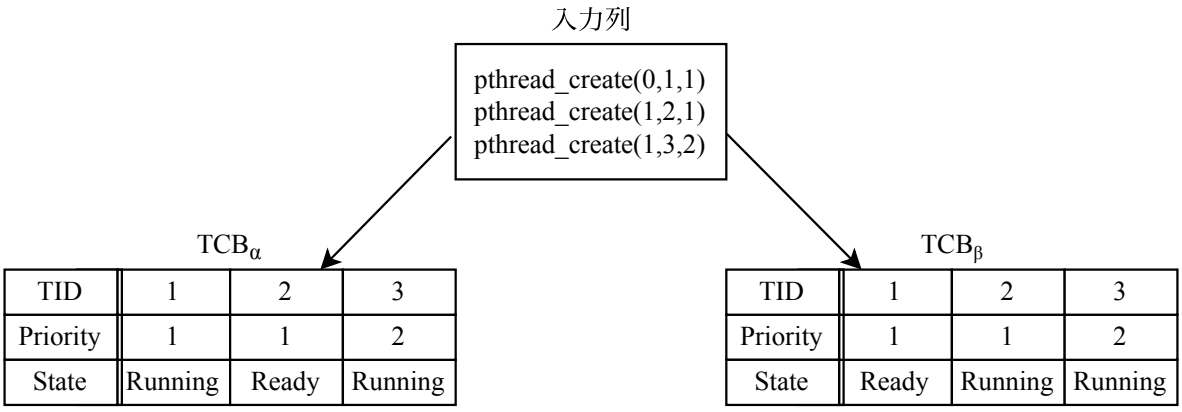


図 4.1　実行結果の不確定性の例

従来 OS のスケジューラに対する MBT [3] では入力に対する期待値は一意に定まっていた．その場合，生成された全てのテストケースを順に実行し，全てが成功したら成功であり，失敗するテストケースがあれば，その部分に何らかの問題が含まれるとしていた.

しかし，POSIX 準拠 OS の場合，実行結果の不確定性による分岐があるため，例えば，図 4.1の API 呼び出し列を入力として与えた際に，TCB$_\alpha$ と TCB$_\beta$ の 2 つの実行結果が

考えられる．実行結果がどちらであっても POSIX は満たしており両方の実行結果が得られる必要はない．また，それぞれ，期待値は異なるため，従来のテスト方式では，それぞれに対応したテストケースが生成され，合計 2 つのテストケースが生成される．このとき，TCB_α に対応するテストケースが失敗し，TCB_β に対応するテストケースが成功した場合，片方が成功しているため POSIX は満たしており問題はないといえるが，従来のテスト方式ではテストケース間の関係性がテスト時には失われているため，このような判断は行えず，片方の失敗したテストケースがフォールスポジティブとして現れてしまう．そのため，図 4.1のような実行結果の不確定性による期待値の分岐を考慮したテストを行えるようにする必要がある．

本章では，(1) 実行結果の不確定性による期待値の分岐を扱うためのテストスイートの定義，(2) 定義したテストスイートの自動生成アルゴリズムについて述べる．

4.2 実行結果の不確定性に対応したテストスイート

API α をテストしているとする．α に入力を与えたときに得られる出力を期待値の型に変換したものを β とする．このとき，期待値 γ が従来のように一意に定まる場合は，$\beta = \gamma$ であるかを確認すれば良い．しかし，POSIX 準拠 OS のスケジューラを対象としたテストでは，実行結果の不確定性により，1 つの入力に対する期待値が複数生じることがある．複数ある期待値を γ_1，γ_2 とする．これらの 2 つの期待値に対し，それぞれ別々の 2 つのテストケースを生成し，$\gamma_1 = \beta$，$\gamma_2 = \beta$ をそれぞれテストする場合，従来手法では，例えば $\gamma_1 = \beta$ となり前者のテストが成功しても，後者のテストが失敗してしまった場合，実行結果の不確定性は考慮されず，後者のテストが失敗したという結果のみが得られ，フォールスポジティブが生じてしまう．これを解決するために，期待値の確認を行う際に，$\gamma_1 = \beta \vee \gamma_2 = \beta$ としてテストを行う方法が考えられる．しかし，この方法では，テスト成功時に $\gamma_1 = \beta$ であるか，$\gamma_2 = \beta$ であるかがテスト結果から判別できない．スケジューラはステートフルであるため，どちらの期待値に一致した動作をしたかにより，今後の動作・実行すべきテストケースが変化するため，テスト時にはどちらの期待値に一致したかが分かるようにする必要がある．したがって，1 つのテストケースで複数の期待値を許容する方法は不適当である．以上の理由により，(1)1 つのテストケースには 1 つの期待値のみ含まれるようにし，かつ，(2) 実行結果の不確定性から生じる期待値同士の関係が失われないテストを行う必要がある．

この 2 つの条件を満たすために，テストケース同士の関係性を図 4.2のような木構造で保持し，この木をテストスイートとして扱う．図の木の葉がそれぞれテストケースを表し

ている．赤枠のテストケースはテストに成功したことを意味しており，青枠は失敗したことを意味している．また，AND は子が全て成功（赤枠）となったときに成功とみなし，OR は子のうち 1 つ以上が成功（赤枠）となったときに成功とみなすとする．このとき，AND は従来のテストケース間の関係性を表し，OR は実行結果の不確定性による分岐を表している．図 4.2では，上側のテストケースは失敗しているが，OR で結ばれた部分木が成功しているため，全体として成功し，テスト結果が POSIX を満たした振る舞いであったという意味になる．

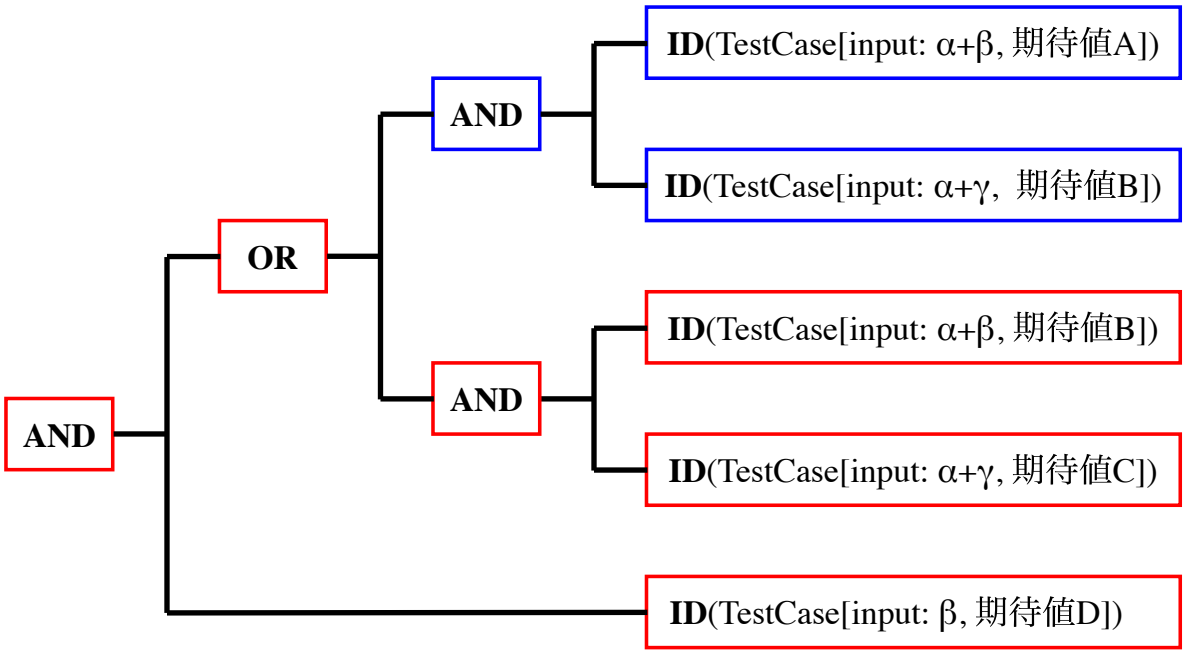


図 4.2　テストスイートの概要図

擬似コードを用いると，テストスイートは次のように記述を行う．ただし，TestCase の第二項目 seq of $\mathbb{N}$ は API の引数を表す．また，OR は実行結果の不確定性による図 4.2の分岐を表す．

```
TestSuite = enum { AND(set of TestCase), OR(set of TestCase), ID(TestCase) }
TestCase = seq of (API × (seq of ℕ) × OsState)
```

4.3　テストケースの定義

テストケースは入力と期待値の組で構成される．スケジューラはステートフルであり，API の実行順により動作が変化するため，テストケースの入力は API 列とし，様々な順

序で API を呼び出すようなテストを行う．このとき，(1) 入力として与えた API 列に含まれる全ての API を実行した後に期待値（1 つ）の確認をする方法と，(2) 入力として与えた API 列に含まれる API の各実行後にその時々の期待値を確認する方法（期待値も列とする方法）の 2 つが考えられる．しかし，(1) では，テストケースに重複が発生する．例えば，API を英アルファベット，期待値をギリシャ文字で表すとき，API A を入力として与え，その結果を期待値 α と比較するテストケースと，API A を実行後に API B を実行し期待値 β と比較するテストケースがそれぞれ生じる．この積み重ねで，(1) の方法では，膨大な数のテストケースが生成されてしまう．(2) の方法では，この問題は生じない．(2) では，API A を実行後にその期待値 α が満たされているかを確認し，続けて API B を実行して期待値 β と比較するため，1 つのテストケースで (1) の 2 つのテストケースで確かめたい内容をテストすることができる．そのため，本稿では，(2) の方法でテストケースを定義する．つまり，API と期待値の組の列をテストケースとして定義する．

テストケースの期待値は，各スレッドの状態（第 3 章の ThreadState）と Mutex 所持者（TID），errno の組とする．第 3 章で定義した OsState を期待値とすると，各 API の動作が正しいことが確認できるが，本稿では各 API 実行後に期待値の確認を行うため，各スレッドの状態を順に追うことで OsState 中の TCB や RQ などのキューの値が正しいことが分かる．そのため，明示的に各キューの値の確認を行う必要はない．

4.4 テストスイート生成アルゴリズム

本研究では，ミッションクリティカルシステムとして POSIX 準拠 OS を使用する場合を想定しているため，どのような状況でどの API を使用しても不正な動作が起こらないことを確かめたい．そのため，テストスイートに対し，(1) 全ての状態 (OsState) に到達し，(2) 各状態で全ての API と引数の組み合わせを実行するようなものであること，(3) 同じ状態から同じ API を実行する場合でも，それまでに通ってきたパスが違う場合は区別されていることを要求する．

この 3 つの要求を満たすテストスイートを生成するために stateless DFS search を基にした探索アルゴリズムを提案する．探索の停止条件（バックトラッキングが生じる条件）は，通常の stateless DFS search と同様に以下の 2 つである．バックトラッキングが生じる地点までのパスをテストケースとする．

■停止条件

- 探索中のパス上で既に訪れた状態である (5 行目で False の場合)

- 現状態からの遷移先が存在しない (8-9 行目で全て False の場合)

以下がテストスイートの生成アルゴリズムである．テストスイート中に冗長な And と Or が生成されることを防ぐために and_wrapper と or_wrapper という関数を導入している．

```
1  procedure search(init_state, apis)
2    init_state: OsState
3    apis: set of API
4  begin
5    return dfs(init_state, <␣,init_state>)
6  end
7
8  procedure dfs(apis,post_proc,current,accum,stack)
9    apis: set of API
10   post_proc: API
11   current: OsState
12 begin
13   and_list := []: seq of TestSuite
14   for each api∈apis do
15     args := cartesian_product(api.range_of_arguments(current))
16     for each arg∈args do
17       if api.requirement(current,arg) then
18         dests := {}: set of OsState
19         for each dest∈api.execute(current,arg) do
20           dests ∪= post_proc.execute(dest)
21         od
22         or_list := []: seq of OsState
23         for each dest∈dests do
24           if current∉stack then
25             new_accum ∪= ((api,arg),dest)
26             new_stack ∪= current
27             or_list ⊕= dfs(apis,post_proc,dest,new_accum,new_stack)
28           fi
29           and_list ⊕= or_wrapper(or_list)
30         od
31       fi
32     od
33   od
34   if and_list = [] then
35     return accum
36   else
37     return and_wrapper(and_list)
38   fi
39 end
40
41 procedure and_wrapper(s)
42   s: set of TestCase
43 begin
44   if |s| > 1 then
45     return AND(s)
46   else
47     return ID(s)
48   fi
49 end
50
51 procedure or_wrapper(s)
52   s: set of TestCase
53 begin
54   if |s| > 1 then
55     return OR(s)
56   else
```

```
57      return ID(s)
58    fi
59  end
```

第 5 章

実装

5.1 概要

[3] では，モデル検査ツールの SPIN を利用して形式化と状態探索を行うことで，テストケースを生成していた．また，そのテストケースを C 言語のテストプログラムに変換することでテストを実行していた．しかし，POSIX 準拠 OS には実行結果の不確定性があるため， 4.4 節で述べたアルゴリズムを使用してテストスイートの生成を行う．このとき，SPIN では探索アルゴリズムの変更はできないため，状態探索アルゴリズムを他のプログラミング言語を使用して実装する必要がある．本研究では，バグ混入率を低くするために，所有権などの概念が導入されておりメモリ関連のバグが生じづらいプログラミング言語である Rust を用いて状態探索を実装する（図 5.1）．また，形式化した POSIX の仕様に関しても，他の形式言語で表現する必要がないため，Rust 上でそのまま実装をする．

図 5.1に示すように，Rust 上で表現した POSIX とテストスイート生成アルゴリズム（ 4.4 節）からテストスイート（ 4.2 節）を生成する．そして，テストスイートを構成するテストケースをテストプログラムに自動変換し，これを実行することで，各テストケースの成否を判定する．テストプログラムでは，POSIX の API を実行するが，Rust では直接，POSIX の API を実行することが想定されておらず，また，所有権などの諸概念がテスト結果に影響を与える可能性があるため，テストプログラムは C++ を用いて実装する．Rust 側でテストを実行すると自動で C++ のテストプログラムを生成・実行し，テスト結果を表示する．

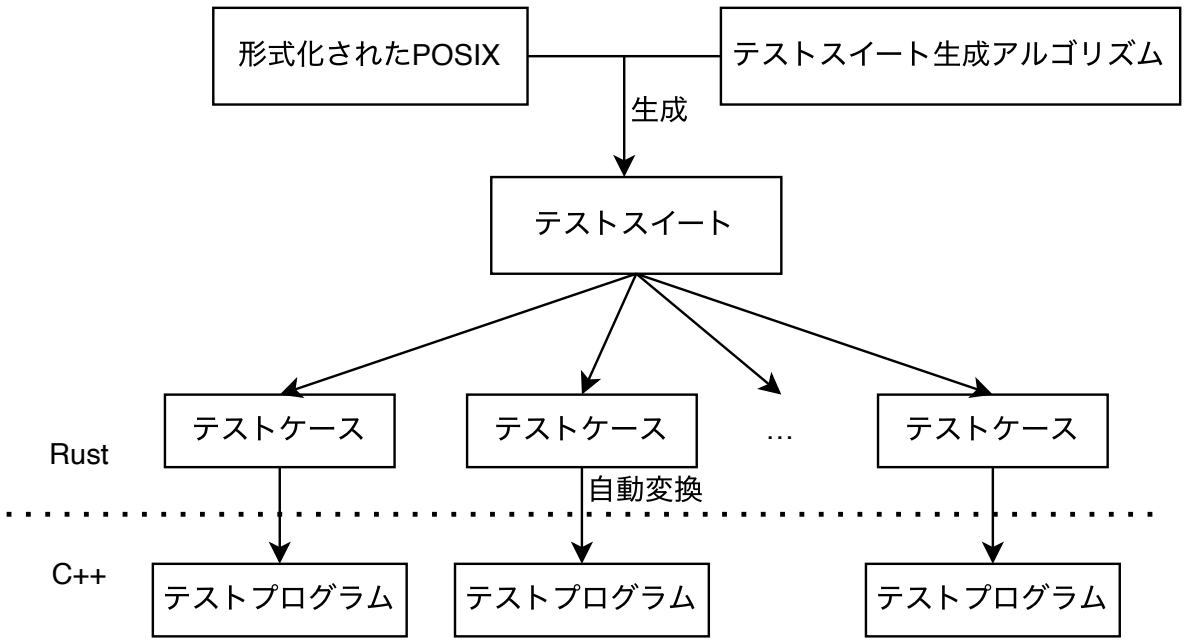


図 5.1　実装の概要図

5.2　クラス構成

前述した通り，バグの混入率を低くするために POSIX の形式化とテストスイートの生成アルゴリズムの実装は Rust で行った．Rust で実装した部分のクラス構成を図 5.2に示す．Rust には，クラスやインターフェースは存在せず，代わりに Trait や Struct，impl などが用いられるが，これらは Rust 特有の概念であるため，図 5.2では簡単のために，Trait をインターフェースとして扱っている．また，Struct と impl はクラスとそのメソッドとして扱っている．

Posix クラスでテストスイートの自動生成アルゴリズム（4.4 節）と，テストを実行するための関数を実装している．createTestSuite で C++ のソースコードを生成し，evalTestSuite で必要なプログラムのみコンパイル・実行し，テストの成否を判定する．

Posix クラスが包含する列挙型の Exp は，テストスイートの型である．Exp は木構造（多分木）であるため，再帰で実装している．Exp が持つ TestProgram クラスは C++ のテストプログラムと一対一対応しており，変数 tp にテストプログラムで実行するテストケースを文字列として保持している．Posix クラスの evalTestSuite によってテストプログラムが実行されたときの結果を TestProgram クラスの result に格納する．このとき，実行する必要のないテストプログラムも存在し，その場合はテストプログラムの成否は確定す

る必要がないため，Option<bool>型としている．

API は 3.4 節で述べたインターフェースの実装として定義している．PthreadCreate，PthreadExit，PthreadMutexLock，PthreadMutexTrylock，PthreadMutexUnlock は POSIX の API である．また，Dispatch は POSIX 準拠 OS のスケジューリングの形式化である．POSIX クラスの変数 apis としてテスト対象の POSIX の API を指定し，post_proc として Dispatch を指定する．

5.3 期待値の取得方法

4.3 節で定めた通り，各スレッドの現状態（Running など）と Mutex 所持者の TID，errno の組を期待値としている．Linux では，procfs という機能を使用して各スレッドの状態を調べることができるが，procfs では Running と Ready を区別していない．また，procfs などの機能で Running と Ready を区別する方法は見つけられていない．そのため，期待値の確認を Running であるべき全てのスレッドで行うことで，実際に Running であるべきスレッドが動作しているかを確認する．このとき，Running であるべきスレッドが Ready となってしまっていた場合，期待値の確認がされないため，テストが進行せずにデッドロックが生じる．テストプログラムが 5 秒以内に終了しない場合は，Running であるべきスレッドが Ready となっていると判断し，テストプログラムを強制終了し，Fail と判定する．

5.4 テストケースからテストプログラムへの変換

5.2 節で述べた通り，Rust 側の TestProgram クラスの変数 tp がテストケースである．しかし，テストケースは実行可能なコードではないため，テストを行うためには実行可能なコードに変換する必要がある．形式化した API と引数を実行可能なコードに変換する関数群を実装したファイルをここではテンプレートファイルと呼ぶ．

図 5.3のように，テンプレートファイルと変数 tp のテストケース（テストプログラム断片）を組み合わせてコンパイルすることにより，テストプログラムを生成する．

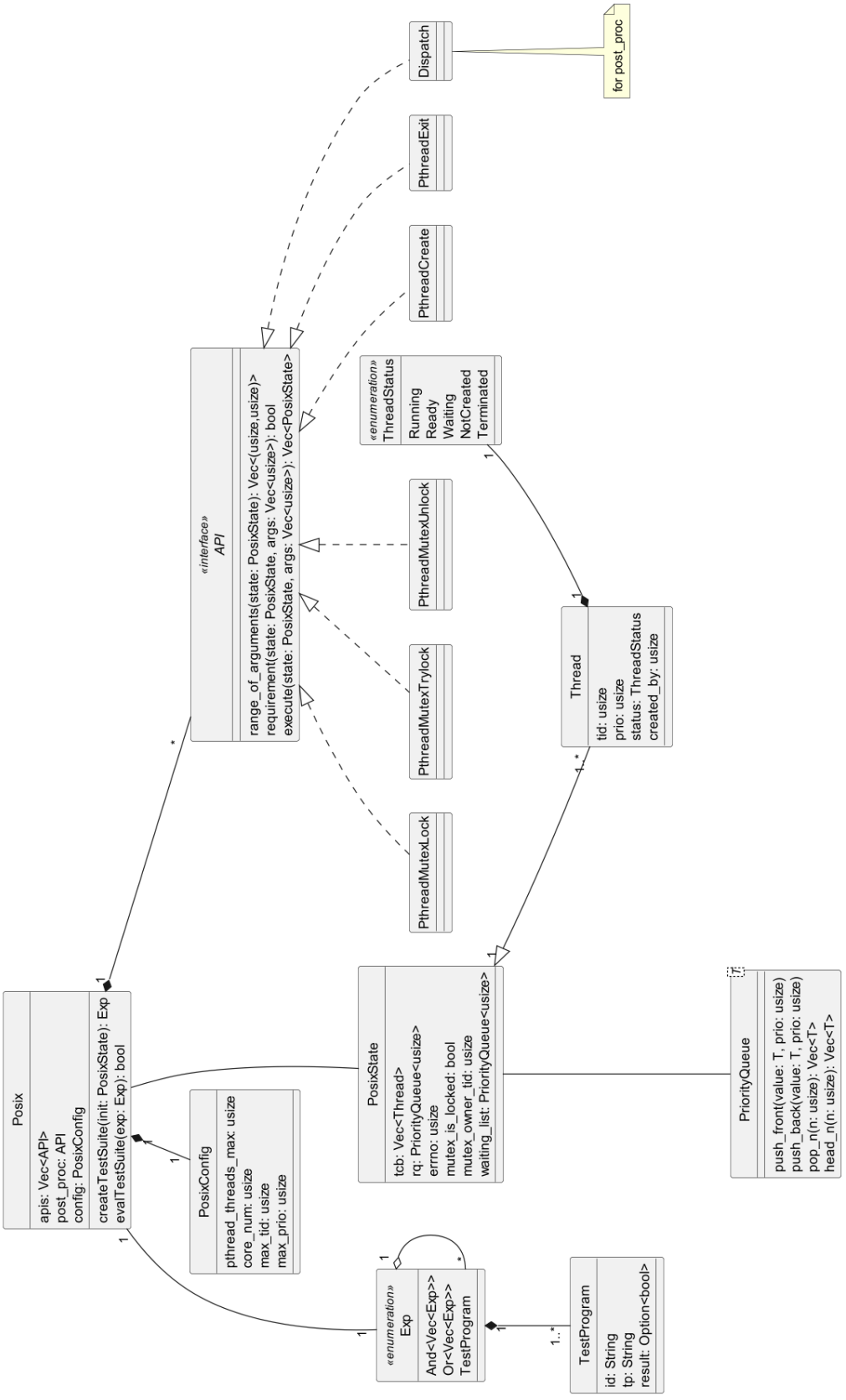

Posix
apis: Vec<API>
post_proc: API
config: PosixConfig
createTestSuite(init: PosixState): Exp
evalTestSuite(exp: Exp): bool
PosixConfig
pthread_threads_max: usize
core_num: usize
max_tid: usize
max_prio: usize
«enumeration»
Exp
And<Vec<Exp>>
Or<Vec<Exp>>
TestProgram
TestProgram
id: String
tp: String
result: Option<bool>
PosixState
tcb: Vec<Thread>
rq: PriorityQueue<usize>
errno: usize
mutex_is_locked: bool
mutex_owner_tid: usize
waiting_list: PriorityQueue<usize>
PriorityQueue
push_front(value: T, prio: usize)
push_back(value: T, prio: usize)
pop_n(n: usize): Vec<T>
head_n(n: usize): Vec<T>
Thread
tid: usize
prio: usize
status: ThreadStatus
created_by: usize
«enumeration»
ThreadStatus
Running
Ready
Waiting
NotCreated
Terminated
«interface»
API
range_of_arguments(state: PosixState): Vec<(usize,usize)>
requirement(state: PosixState, args: Vec<usize>): bool
execute(state: PosixState, args: Vec<usize>): Vec<PosixState>
PthreadMutexLock
PthreadMutexTrylock
PthreadMutexUnlock
PthreadCreate
PthreadExit
Dispatch
for post_proc

図 5.2　クラス構成

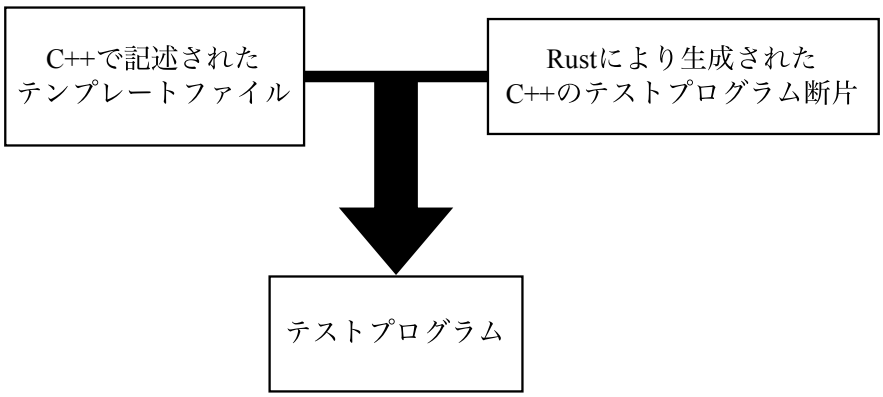


図 5.3　テストプログラムの生成方法

テストケースとテンプレートファイルを組み合わせてコンパイルするために，変数 tp が持つ文字列はテストケースを C++ のソースコードに変換したものにしてある．pthread_create と pthread_exit を対象としたテストのテストケース例（変数 tp の例）を次に示す．

```
#include <utility>
#include <vector>
#include <string>
#define READY 0
#define RUNNING 1
#define WAITING 2
#define NOTCREATED 3
#define TERMINATED 4
using ull = unsigned long long;
std::pair<std::string, std::vector<ull>> apis[] = {
{"PthreadCreate",{0,1,1}},{"<CheckRunning>",{0,0,0,1,RUNNING}},
{"PthreadCreate",{1,2,1}},{"<CheckRunning>",{1,0,0,1,RUNNING,2,RUNNING,3,NOTCREATED}},{"
  <CheckRunning>",{2,0,0,1,RUNNING,2,RUNNING,3,NOTCREATED}},
{"PthreadCreate",{1,3,1}},{"<CheckRunning>",{1,0,0,1,RUNNING,2,RUNNING,3,READY}},{"<
  CheckRunning>",{2,0,0,1,RUNNING,2,RUNNING,3,READY}},
{"PthreadExit",{1}},{"<CheckRunning>",{2,0,0,1,TERMINATED,2,RUNNING,3,RUNNING}},{"<
  CheckRunning>",{3,0,0,1,TERMINATED,2,RUNNING,3,RUNNING}},
{"PthreadExit",{2}},{"<CheckRunning>",{3,0,0,1,TERMINATED,2,TERMINATED,3,RUNNING}},
{"PthreadExit",{3}},{"<CheckRunning>",{0,0,0,1,TERMINATED,2,TERMINATED,3,TERMINATED}},
{"<completed>", {}}
};
```

変数 apis がテストケースであり，それ以外の部分はコンパイルを通すために必要な定義である．変数 apis は，API 名と引数の列であり，PthreadCreate と PthreadExit に関しては，第 3 章で形式化した API 仕様に則っている．<CheckRunning>は期待値の確認であり，引数の意味は左から，(1) 期待値の確認を行うスレッドの TID，(2)errno，(3)Mutex の所持者（解放時は 0）であり，4 つ目以降は TID とその TID を持つスレッドの現状態が交互に並んでいる．

テンプレートファイルは大域変数 g_ct_counter が定義してあり，このカウンタ変数で，

変数 apis のテストケースのうち，現在何番目の API を実行しているかを管理している．各引数の最左の値は，その API/期待値の確認を実行するスレッドの TID であり，それ以外のスレッドはビジーウェイトを実行して待機させている．これにより，各スレッド上で意図したタイミングで API を実行させている．

図 5.4は，上からそれぞれのスレッドの動作を時系列順に並べた図である．定期的に ct load (g_ct_counter の読み込み) を行いながら，ビジーウェイトを実行し，API を実行するスレッドであった場合は，API を呼び出し，g_ct_counter を加算している．例外的に，pthread_exit を実行する際は，API 呼び出し後に g_ct_counter の加算ができないため，API 呼び出し前に加算を行っている．このとき，pthread_exit を実行するスレッドが g_ct_counter を更新するタイミングが未知であるため，古い値を g_ct_counter に store する可能性が生じる．そのため，g_ct_counter の値が誤って古い値で更新されないかを判定してから値の更新を行う必要がある．古い値であるかの判定と値の更新を atomic な操作としなければ，同様の問題が生じてしまうが，OS の排他制御機構はそれ自体がテスト対象の機能であるため，テストプログラムを作るために使用することはできない．そのため，本研究では，インラインアセンブラ使って ARM64 が持つ排他制御機構 (LL/SC) を使用している．OS よりも低レイヤーの排他制御機構であるため，テストの品質に影響を与えないが，アーキテクチャごとにテストプログラムを編集する必要がある．

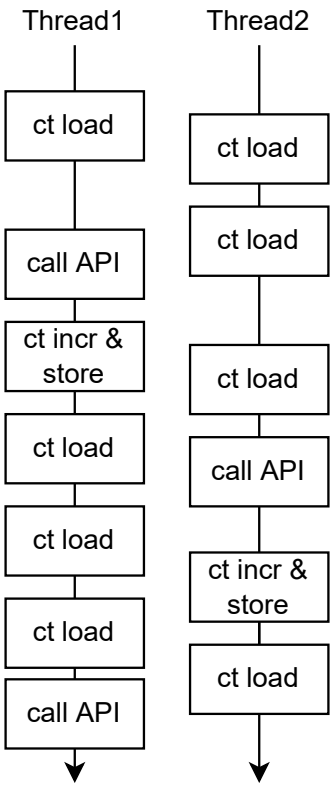


図 5.4　ビジーウェイトとインラインアセンブラによる排他制御

第 6 章

実験

6.1　目的

実験により，提案手法の有用性とパフォーマンス，形式化の妥当性を評価する．

従来手法を用いる場合，テストスイートは木構造ではなく，図 6.1の実行順の矢印に沿って全てのテストケースが実行される．図 6.1の緑色の丸で括られた部分は実行結果の不確定性による分岐している部分であり，この部分のテストケースが失敗しても，OR で結ばれた部分木のテストケースが成功すれば振る舞いは POSIX を満たしているため，問題はない．しかし，従来手法で生成されるテストケースでは，そのような判定はできないため，緑色の丸部分はフォールスポジティブとなってしまう．このようなフォールスポジティブがどの程度含まれ，削減されたかを実験から明らかにし，テストスイートを木構造として表す方法の有用性を評価する．

また，テストでは，API の引数の選び方と API の呼び出し順による組合せ爆発が起きる可能性がある．提案手法の実行に必要な時間を測定し，提案手法のパフォーマンスを評価し，テストの実行結果から提案手法の妥当性を評価する．

最後に，テストの Pass/Fail の判定から，提案手法の妥当性に関して評価を行う．

6.2　実験に使用する OS とパラメータ設定

aarch64 アーキテクチャの MacBook Pro 2022 上で UTM（QEMU のフロントエンドアプリケーション）を使用して行う．OS は使用率の高いディストリビューションである Ubuntu 22.04 LTS を使用する．また，並列に実行可能なスレッドの数（CPU のコア数）は 2 つとして設定している． 3.4.2 節のパラメータ，MaxTid（生成されるスレッド数の上

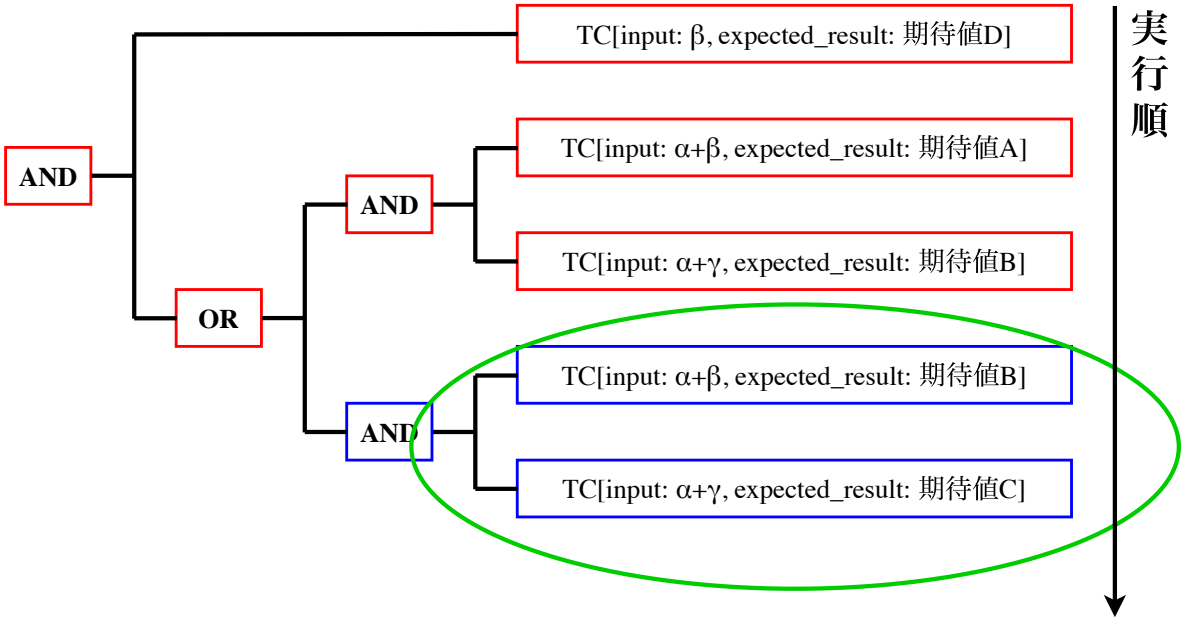


図 6.1　フォールスポジティブの例

限）と MaxPrio（生成されるスレッドの優先度上限）はそれぞれ，2 から 4 の間で変化させて実験を行う．対象 API は，(1)pthread_create,pthread_exit と (2)pthread_create,pthread_mutex_lock,pthread_mutex_unlock の 2 種である．

6.3　実験結果

6.3.1　テスト総数とフォールスポジティブの数

実行結果の不確定性で分岐したテストケースが全て成功する可能性もある．しかし，今回の実験では，実験結果の不確定性で分岐した 1 つ以外の分岐先のテストケースは全て失敗する場合（フォールスポジティブの数が最大となる場合）を想定している．提案手法により除去されたフォールスポジティブの数を表の括弧外に，生成されたテストケースの総数を括弧内に示す．また，行は MaxTid（生成されるスレッド数の上限）を表し，列は MaxPrio（生成されるスレッドの優先度上限）を表す．

スレッドの生成・終了に関する API での実験結果を表に示す．

表 6.1　pthread_create と pthread_exit を対象とした際の除去された FP 数（最良）と生成されたテストケースの総数

TID\PRIO	2	3	4
2	0 (5)	0 (7)	0 (9)
3	16 (121)	32 (261)	48 (401)
4	1824 (7117)	5520 (23035)	11088 (41113)

Mutex に関する API と pthread_create を対象とした場合の実験結果を次表に示す．生成される TID の上限を 4 とした際は，テストケースの生成が終わらなかったため，結果を省略している．

表 6.2　pthread_create と pthread_mutex_lock，pthread_mutex_unlock を対象とした際の除去された FP 数（最良）と生成されたテストケースの総数

TID\PRIO	2	3	4
2	0 (25)	0 (37)	0 (49)
3	53626 (71885)	73408 (112709)	93190 (161503)

6.3.2　テストの実行時間と実行結果

提案手法の実行に必要な時間と，テストの実行結果を示す．括弧外が実行時間であり，s は秒，m は分を表す．また，括弧内はテストの実行結果を表しており，F は Fail，P は Pass を表す．テストは結果が確定した時点で終了するため，テスト結果が失敗している部分は成功時よりも短い時間で終了している．行は MaxTid（生成されるスレッド数の上限）を表し，列は MaxPrio（生成されるスレッドの優先度上限）を表す．

スレッドの作成・終了に関する API を対象とした場合の結果を表に示す．

表 6.3　pthread_create と pthread_exit を対象とした際のテストの実行時間と実行結果

TID\PRIO	2	3	4
2	28.5 s (P)	39.8 s (P)	51.2 (P)
3	57.0 s (F)	11.5 s (F)	57.0 s (F)
4	1 m 45.1 s (F)	15.0 s (F)	45.7 s (F)

Mutex に関する API と pthread_create を対象とした場合の実行時間と実行結果を表に

示す．生成される TID の上限を 4 とした際は，テストケースの生成が終わらなかったため，結果を省略している．

表 6.4　pthread_create と pthread_mutex_lock，pthread_mutex_unlock を対象とした際のテストの実行時間と実行結果

TID\PRIO	2	3	4
2	2m 22.8 s (P)	3 m 31.1 s (P)	4 m 39.0 s (P)
3	31 m 59.2 s (F)	29 m 24.3 s (F)	7m 23.6 s (F)

第 7 章

評価

7.1　有用性

生成されるスレッドの数が 4 つを超えた時点でフォールスポジティブの数が 4 桁を超えている．例えば，スレッドの上限数が 4 つ，優先度の幅を 4 とした際に従来の手法を用いると，41113 個のテストケースを実行し，そのうちの失敗するとされる 11088 個のフォールスポジティブを手動で取り除くことになる．そのため，本研究で提案した，実行結果の不確定性を明示的に表した木構造のテストスイートは有用であると判断できる．

また，mutex と pthread_create を絡めたテストを行う際には，実験結果によれば，TID の上限値が 3 を超えた時点で既にフォールスポジティブの数が 5 桁を超えるため，手動での選別は困難となる．そのため，mutex を含めたテストを行う際には特に有用であると考えられる．しかし，スレッドの数を 4 つとした際にテストケースの生成が終わらなかったことを考えると，テストケースの総数を削減する手法が必要であると考えられる．

7.2　パフォーマンスと形式化の妥当性

パフォーマンスに関して，スレッドの作成・終了を対象とした実験（表 6.3）においては，テストが失敗していることを加味しても実行時間は短く，良好なパフォーマンスであると判断できる．しかし，Mutex を対象とした実験（表 6.4）に関しては，失敗しているにも関わらず 30 分を超えている項目もあり，また，TID の上限値を 4 とした場合は実行が終了しなかったため，問題があると考えられる．Mutex を対象としたテストでは，テストケースの総数がスレッドの作成・終了を対象としたテストに比べて，約 5 倍増加しており，その影響でテスト時間が増加している．Mutex の取得タイミング，解放タイミング，

スレッドの生成タイミングの 3 要素に関して，自由に組み合わせが取れるため，組合せ爆発が生じてテストケース数が増加していると考えられる．

妥当性に関して，API の選択に関わらず，TID の上限を 2 とした場合にテストに成功しており，3 以上にした際に失敗している．図 1.2で示した優先度に起因する実行結果の不確定性はスレッドの数が 3 以上となった場合にのみ発生するため，実行結果の不確定性が関係しない部分のテスト生成・実行には問題はないと考えられる．しかし，実行結果の不確定性が生じるのは TID の上限が 3 以上の場合であるため，テストが失敗した原因は図 1.2で示した優先度に起因する動作以外の仕様において実行結果の不確定性が存在している，または，拾い切れていない挙動が残っているためであると考えている．しかしながら，実行結果の不確定性がどの部分で生じているかについてはテスト結果を見ても発見できなかったため，この原因を解決するためにはテストケースに含まれていない動作を探索するための新たな手法が必要である．

第 8 章

関連研究

Linux を対象とした形式手法やテストに関する研究は過去にも行われている．しかし，Linux をミッションクリティカルシステム上で使用することを想定し，かつ，アプリケーション開発者が使用する POSIX に焦点を置いて検証を行った研究は行われていない．そのため，API の仕様の緩さから生じる実行結果の不確定性を扱った研究は見つけられていない．

[10] では，Linux のカーネルに対して形式手法 (runtime verification) を用いて，検証を行っている．望ましい振る舞いのモデルを作成し，そのモデルから実行可能なコードを生成している．そして，そのコードをカーネルにモジュールとして埋め込むことで runtime verification を行っている（図 8.1）．上側の最右の図でチェックマークのついている遷移は望ましい動作であり，赤の☒印の付いた遷移は望ましくない動作である．望ましくない動作をした際のログを trace として出力している．この研究では，スケジューラを扱ってはいるが，カーネルの内部に焦点を当てており，スケジューリングの処理の正しさ（スレッドの優先度が守られているかなど）に関しては検証を行っていない．加えて，マルチコア環境ではないため，実行結果の不確定性は生じていない．

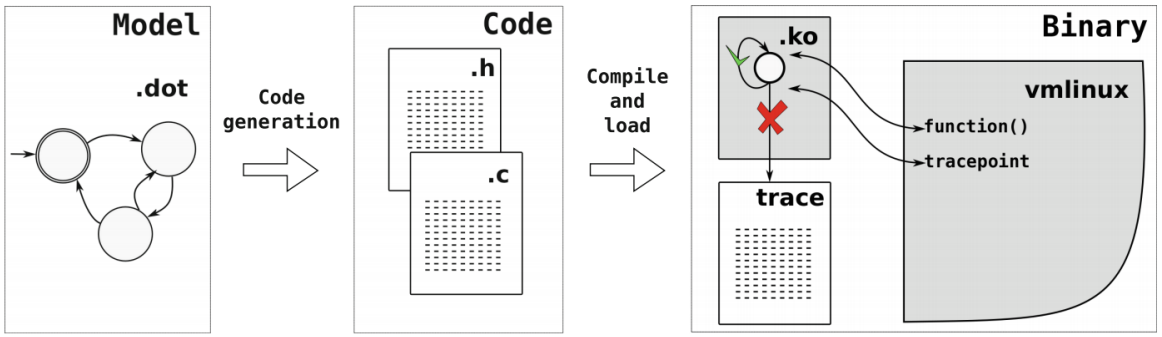


Fig. 3. Verification approach.

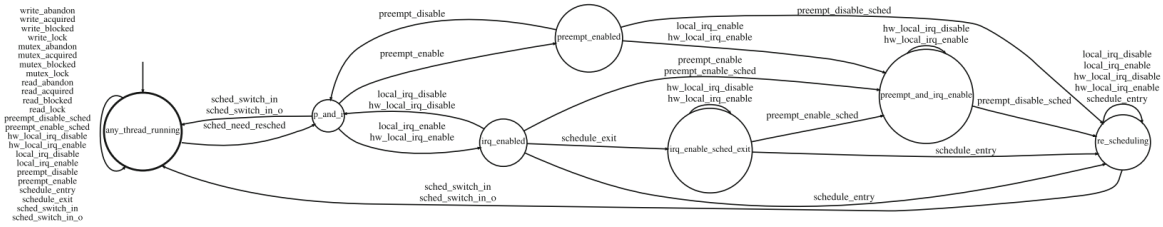
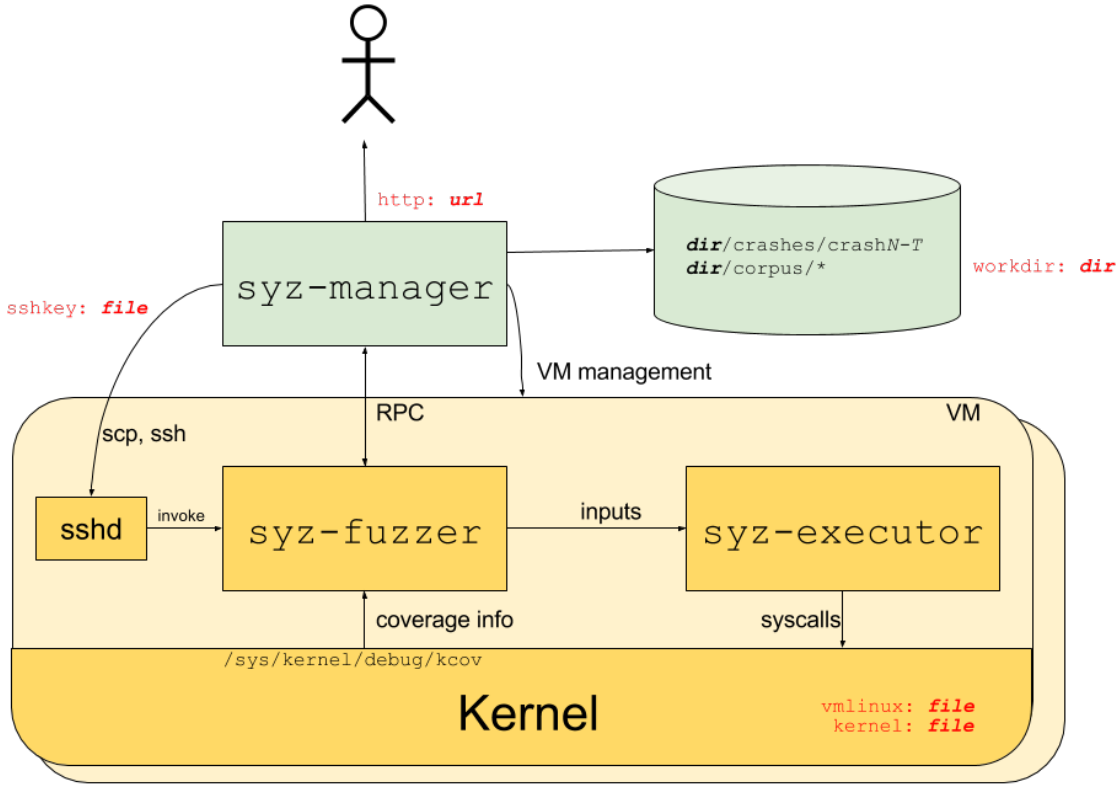


Fig. 10. Need Re-Sched forces Scheduling (*NRS* model) from [33] (see footnote 2).

図 8.1　Linux カーネルに対する runtime verification [10]

Syzkaller [11] では，Linux のカーネルを対象とした Fuzzing を行っており，多数のバグが既に発見されている（図 8.2）．Linux の品質向上に関係する有名なツールである．Syzkaller は，仮想環境上の Linux に対してランダムにシステムコールを実行し，想定外のバグが隠されていないかを探索するツールである．しかし，このツールは Fuzzing ツールであるため，仕様通りの望ましい振る舞いをすることを確かめるのではなく，ハングアップなどの望ましくない振る舞いをしないかを確かめる目的で作られており，目的が本研究とは異なる．



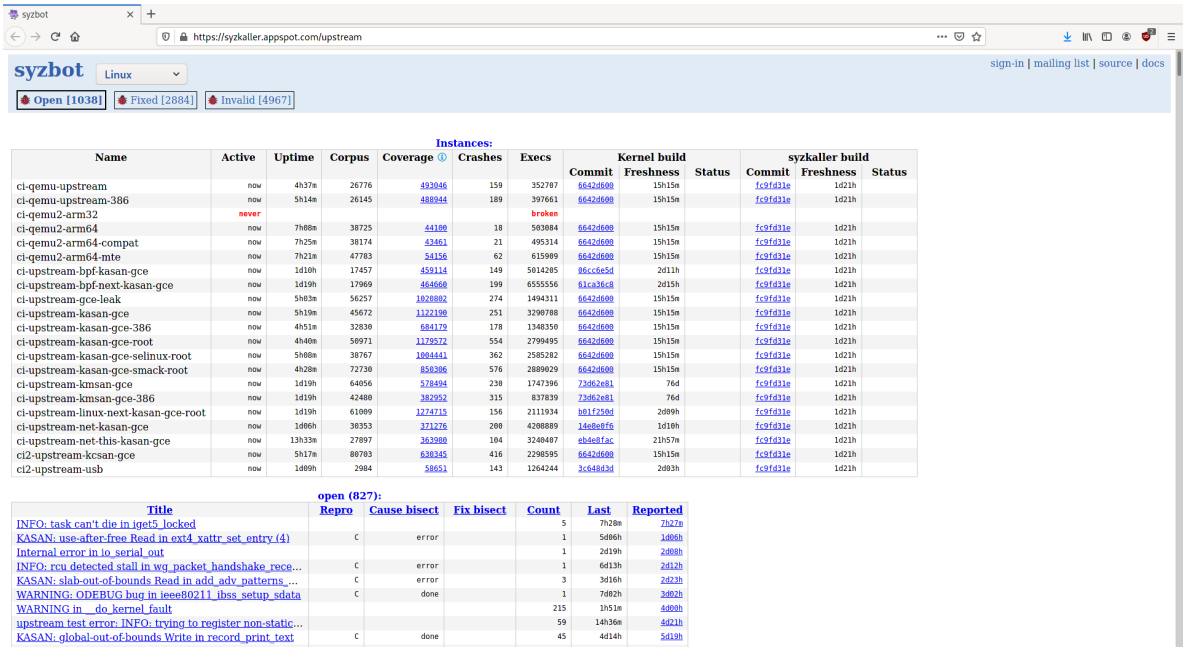


図 8.2　Linux カーネルに対する Fuzzing [11]

[12] では，リアルタイム用途の Linux に対するテストスイートを提供しており，その一環としてスケジューリングの処理に関してのテストスイートも提供している．また，POSIX を明示的に扱っており，スレッドの優先度などにも着目したテストを行っている．しかし，本研究とは異なり，Linux をミッションクリティカルシステムとして使用することを想定していないため，優先度や API の実行順の観点で網羅的なテストは行っていない．また，その影響で，実行結果の不確定性が生じるような複雑なケースは扱われていない．

第 9 章

まとめ・今後の課題

POSIX のスケジューリングに関する API に対する MBT の手法を提案した．その際，従来のリアルタイム OS では生じていなかった POSIX 特有の問題である実行結果の不確定性を MBT で扱うことを試みた．その過程で，(1) 実行結果の不確定性を明示的に扱える木構造のテストスイート，(2)API の実行順を網羅するテストスイートの自動生成アルゴリズム，(3)POSIX のスケジューリングに関する API の形式化手法の提案を行った．また，実際に Linux に対して提案手法を適用した際にテスト結果が Fail だったため，原因は不明であるが，提案手法を用いて実際の OS の動作と認識していた振る舞いに齟齬があったを明らかにした．ミッションクリティカルシステムでは，このような認識の齟齬がバグに繋がるため，齟齬の有無を確かめる用途で本提案手法は有用である．

前述のテストの失敗原因として，(1) 実装が POSIX を違反している，(2) 仕様を満たす動作のうち形式化されていないものがあるという 2 つの可能性がある．(2) であるときに今後，形式化した仕様を改善するためには，テストを失敗させる原因となった発見できていない振る舞いを探索する手法が必要である．

また，実験に提案手法を適用した際に，Mutex に関する API と pthread_create を対象としたテストを行うと，テストケース数が膨大となり，テストが実行できないことが判明した．この問題を解決するために，今後，テストケース総数を抑制する手法の提案が必要であると考えている．

第 10 章

付録

10.1　テンプレートファイル

```
#ifndef _GNU_SOURCE
#define _GNU_SOURCE
#endif
#include <algorithm>
#include <cassert>
#include <csignal>
#include <cstdio>
#include <cstring>
#include <filesystem>
#include <fstream>
#include <pthread.h>
#include <sched.h>
#include <string>
#include <unistd.h>
#include <unordered_map>
#include <vector>

#define READY 0
#define RUNNING 1
#define WAITING 2
#define NOTCREATED 3
#define TERMINATED 4

using ull = unsigned long long;
using ll = long long;

/******************/
/*  テストケース  */
/******************/
#define NUM_CORE 2
extern std::pair<std::string, std::vector<ull>> apis[];

std::unordered_map<ull, ull> tid_mapping;

/****************/
/*  カウンター  */
/****************/
bool ct_incr(volatile uint64_t *global_ct, volatile uint64_t local_ct) {
  uint64_t prev = *global_ct;
```

```
40    __asm__ volatile("1: ldaxr x0, [%0]\n"
41                     "cmp %1, x0\n"
42                     "blt 2f\n"
43                     "add x0, x0, #1\n"
44                     "stlxr w1, x0, [%0]\n"
45                     "cbnz w1, 1b\n"
46                     "2: nop"
47                     : "+r"(global_ct), "+r"(local_ct)
48                     :
49                     : "x0", "w1", "cc", "memory");
50    return prev != *global_ct;
51  }
52
53  void incr(volatile uint64_t *ct) {
54    __asm__ volatile("1: ldaxr x0, [%0]\n"
55                     "add x0, x0, #1\n"
56                     "stlxr w1, x0, [%0]\n"
57                     "cbnz w1, 1b"
58                     : "+r"(ct)
59                     :
60                     : "x0", "w1", "cc", "memory");
61  }
62
63  void dec(volatile uint64_t *ct) {
64    __asm__ volatile("1: ldaxr x0, [%0]\n"
65                     "add x0, x0, #-1\n"
66                     "stlxr w1, x0, [%0]\n"
67                     "cbnz w1, 1b"
68                     : "+r"(ct)
69                     :
70                     : "x0", "w1", "cc", "memory");
71  }
72
73  /******************/
74  /* グローバル変数 */
75  /******************/
76
77  // 次に実行するテストケース番号
78  volatile struct { uint64_t value; } g_tc_counter;
79  // volatile uint64_t g_tc_counter;
80
81  volatile uint64_t finish_counter;
82  volatile uint64_t current_active_tid = 1;
83  volatile uint64_t mutex_waiting_counter;
84  volatile bool check_mode = false;
85
86  ull api_errno;
87  pthread_mutex_t mutex;
88
89  /******************/
90  /*  テスト関数群  */
91  /******************/
92
93  void _busyWait(void) {
94    volatile int i;
95    for (i = 0; i < 1000000; i++)
96      ;
97  }
98  void *thread(void *a_tid);
99
100 ull next_check_exp(const ull a_ct) {
101   ull ct{a_ct};
102   while (apis[ct].first != "<CheckRunning>" &&
103          apis[ct].first != "<completed>") {
```

```
104       ct++;
105     }
106     return ct;
107   }
108
109   // returning -1 means the thread responding the tid has not launched yet.
110   ll tid_conv(ull tid) {
111     for (const auto &k : tid_mapping) {
112       if (k.first == tid)
113         return k.second;
114     }
115     return -1;
116   }
117
118   bool _checker(const ull caller_tid, const ull tc_counter) {
119     const std::vector<ull> api_arg{apis[tc_counter].second};
120     // Check Running
121     if (caller_tid != api_arg[0]) {
122       return false;
123     }
124     std::unordered_map<ull, ull> tid2state{};
125     for (const auto &entry :
126          std::filesystem::directory_iterator("/proc/self/task")) {
127       std::string d_path{entry.path().string()};
128       std::string d_name{entry.path().filename()};
129       std::string path = d_path + "/status";
130       std::ifstream file(path);
131       std::string s_state;
132       ll pid;
133       if (file.is_open()) {
134         std::string line;
135         while (std::getline(file, line)) {
136           pid = tid_conv(std::stoull(d_name));
137           if (pid != -1) {
138             if (line.find("State") != std::string::npos) {
139               if (line.find("sleeping") != std::string::npos) {
140                 tid2state.emplace(pid, WAITING);
141               } else if (line.find("running") != std::string::npos) {
142                 tid2state.emplace(pid, RUNNING);
143               } else if (line.find("disk sleep") != std::string::npos) {
144                 tid2state.emplace(pid, WAITING);
145               } else {
146                 printf("[ERROR] %s\n", line.c_str());
147                 exit(EXIT_FAILURE);
148               }
149             }
150           }
151         }
152         file.close();
153       }
154     }
155
156     // for (const auto&v : tid2state) {
157     //   printf("%llu (%lu): %llu, %llu\n", tc_counter, g_tc_counter, v.first,
158     //   v.second);
159
160     //}
161
162     for (int i = 3; i < api_arg.size(); i += 2) {
163       const auto tid = api_arg[i];
164       const auto th_state = api_arg[i + 1];
165       auto res = tid2state.find(tid);
166
167       if (res == tid2state.end()) {
168         if (th_state != NOTCREATED && th_state != TERMINATED)
```

```
169           return false;
170       } else {
171         if (th_state == READY) {
172           if (tid == caller_tid || res->second != RUNNING)
173             return false;
174         } else {
175           if (res->second != th_state)
176             return false;
177         }
178       }
179     }
180
181     if (api_errno != api_arg[1]) {
182       fprintf(stderr, "unexpected errno (%llu): %s\n", api_errno,
183               strerror(errno));
184       return false;
185     }
186
187     auto res_for_owner = tid_mapping.find(mutex.__data.__owner);
188     ull mutex_owner = 0;
189     if (res_for_owner == tid_mapping.end()) {
190       mutex_owner = 0;
191     } else {
192       mutex_owner = res_for_owner->second;
193     }
194
195     if (mutex_owner != api_arg[2]) {
196       fprintf(stderr, "unexpected mutex owner (%llu:%llu)\n", mutex_owner,
197               api_arg[2]);
198       return false;
199     }
200     return true;
201   }
202
203   bool check_exp(const ull caller_tid, const ull tc_counter) {
204   #ifndef __linux__
205     perror("Only Linux is supported\n");
206     exit(EXIT_FAILURE);
207   #endif
208
209     const std::string api_name{apis[tc_counter].first};
210     if (api_name != "<CheckRunning>")
211       return false;
212     if (_checker(caller_tid, tc_counter)) {
213       if (ct_incr(&g_tc_counter.value, tc_counter)) {
214         if (apis[tc_counter + 1].first != "<CheckRunning>") {
215           check_mode = false;
216         }
217       }
218       return true;
219     } else {
220       return false;
221     }
222   }
223
224   void threadContext(const ull tid) {
225     ull tc_counter = g_tc_counter.value;
226     while (check_mode && !check_exp(tid, tc_counter))
227       tc_counter = g_tc_counter.value;
228     const std::string api_name{apis[tc_counter].first};
229     const std::vector<ull> api_arg{apis[tc_counter].second};
230     if (api_name == "<completed>") {
231       exit(EXIT_SUCCESS);
232     }
233
```

```
234   if (api_name != "<CheckRunning>" && tid == api_arg[0]) {
235     {
236       check_mode = true;
237       if (ct_incr(&g_tc_counter.value, tc_counter)) {
238
239         /* BEGIN: API定義 */
240         if (api_name == "PthreadCreate") {
241           const ull new_tid{api_arg[1]};
242           const ull new_prio{api_arg[2]};
243           printf("%llu: PthreadCreate(%llu, %llu, %llu)\n", tc_counter, tid,
244                  new_tid, new_prio);
245           pthread_t tid;
246           pthread_attr_t attr;
247           pthread_attr_init(&attr);
248           pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
249           pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED);
250           pthread_attr_setschedpolicy(&attr, SCHED_FIFO);
251
252           struct sched_param param;
253           pthread_attr_getschedparam(&attr, &param);
254           param.sched_priority = new_prio + 1;
255           if ((errno = pthread_attr_setschedparam(&attr, &param))) {
256             perror("pthread_attr_setschedparam");
257             exit(EXIT_FAILURE);
258           }
259
260           incr(&current_active_tid);
261           if ((api_errno =
262                    pthread_create(&tid, &attr, thread, (void *)new_tid))) {
263             perror("pthread_create");
264             exit(EXIT_FAILURE);
265           }
266         } else if (api_name == "PthreadExit") {
267           printf("%llu: PthreadExit(%llu)\n", tc_counter, tid);
268           dec(&current_active_tid);
269           api_errno = 0;
270           pthread_exit(NULL);
271         } else if (api_name == "PthreadMutexLock") {
272           printf("%llu: PthreadMutexLock(%llu)\n", tc_counter, tid);
273           incr(&mutex_waiting_counter);
274           api_errno = pthread_mutex_lock(&mutex);
275           dec(&mutex_waiting_counter);
276         } else if (api_name == "PthreadMutexTrylock") {
277           printf("%llu: PthreadMutexTrylock(%llu)\n", tc_counter, tid);
278           api_errno = pthread_mutex_trylock(&mutex);
279         } else if (api_name == "PthreadMutexUnlock") {
280           printf("%llu: PthreadMutexUnlock(%llu)\n", tc_counter, tid);
281           api_errno = pthread_mutex_unlock(&mutex);
282         }
283       }
284       /* END: API定義 */
285     }
286   }
287 }
288
289 void *thread(void *a_tid) {
290   const ull tid = (ull)a_tid;
291   tid_mapping.emplace((ull)gettid(), tid);
292
293   while (1) {
294     threadContext(tid);
295   }
296   pthread_exit(0);
297 }
```

```
298
299 /******************/
300 /*   メイン関数   */
301 /******************/
302
303 int main() {
304 #ifdef __linux__
305   cpu_set_t cpu_set;
306   CPU_ZERO(&cpu_set);
307   CPU_SET(2, &cpu_set);
308   CPU_SET(3, &cpu_set);
309   if (sched_setaffinity(0, sizeof(cpu_set_t), &cpu_set) != 0) {
310     perror("SET AFFINITY FAILED\n");
311     exit(EXIT_FAILURE);
312   }
313 #endif
314   struct sched_param param;
315   param.sched_priority = sched_get_priority_min(SCHED_FIFO);
316   if (sched_setscheduler(0, SCHED_FIFO, &param) != 0) {
317     perror("SET POLICY FAILED\n");
318     exit(EXIT_FAILURE);
319   }
320
321   pthread_mutexattr_t mtx_attr;
322   pthread_mutexattr_init(&mtx_attr);
323   // pthread_mutexattr_setrobust(&mtx_attr, PTHREAD_MUTEX_ROBUST);
324   //  pthread_mutexattr_settype(&mtx_attr, PTHREAD_MUTEX_ERRORCHECK);
325   pthread_mutex_init(&mutex, &mtx_attr);
326   while (1) {
327     threadContext(0);
328   }
329 }
```

Bibliography

[1] *The Open Group Base Specifications Issue 7, 2018 edition*, https://pubs.opengroup.org/onlinepubs/9699919799.2018edition/ Accessed: 2022-01-02.

[2] *Adaptive Platform–AUTOSAR*, https://www.autosar.org/standards/adaptive-platform/ Accessed: 2021-12-13.

[3] T. Aoki, M. Satoh, M. Tani, K. Yatake, and T. Kishi, "Combined model checking and testing create confidence—a case on commercial automotive operating system," in *Cyber-Physical System Design from an Architecture Analysis Viewpoint: Communications of NII Shonan Meetings*. Springer, 2017, pp. 109–132.

[4] D. A. Patterson and J. L. Hennessy, **コンピュータの構成と設計 第 *5* 版 下**. 日経 BP 社, 2014, 成田光彰 訳.

[5] G. J. Holzmann, *The SPIN MODEL CHECKER: Primer and Reference Manual*. Addison-Wesley Professional, 2003.

[6] *Spin–Formal Veriification*, http://spinroot.com Accessed: 2022-01-31.

[7] S. S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[8] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating system concepts 10th edition*. John Wiley & Sons, 2018.

[9] G. C. Buttazzo, *Hard Real-Time Computing Systems Third Edition*. Springer, 2011.

[10] D. B. de Oliveira, T. Cucinotta, and R. S. de Oliveira, "Efficient formal verification for the linux kernel," in *SEFM*, Springer, 2019.

[11] *Google/syzkaller*, https://github.com/google/syzkaller Accessed: 2021-11-02.

[12] A. Claudi and A. F. Dragoni, “Testing linux-based real-time systems: Lachesis,” in *SOCA*, 2011.