Vol.2020-OS-150 No.1
2020/7/30

# Ultra-Low Latency SSDを活用するための ユーザ空間ライブラリ

田所 秀和[1,a)]

概要：従来のSSDよりも低レイテンシを実現したUltra-Low Latency SSD (ULL SSD) と呼ばれるデバイスが出現しつつある．このような低レイテンシデバイスを活用する上で，カーネルを経由するオーバーヘッドや割り込みを使うことによる性能低下が問題になっている．そこで，ユーザ空間スレッドとユーザ空間ファイルシステムを組み合わせることで，高い性能を出すことができるライブラリを開発した．このライブラリは，POSIX互換で作られているため，既存のアプリケーションをそのまま動作させることができる．我々の知る限り，カーネルの関与と割り込みの利用を完全に排除しつつ，既存アプリケーションを動作させる技術は，このライブラリが初めてである．市販のULL SSDを用いた実験において，既存ファイルシステムではIO性能を使い切れなかったが，提案ライブラリ適用によりファイルシステムを経由してもカタログスペック値と同等のファイルIO性能を実現できることを確認した．また，SQLiteのテストを利用し，99.97%を越えるテストケースをパスする高い互換性を確認した．

## 1. 背景

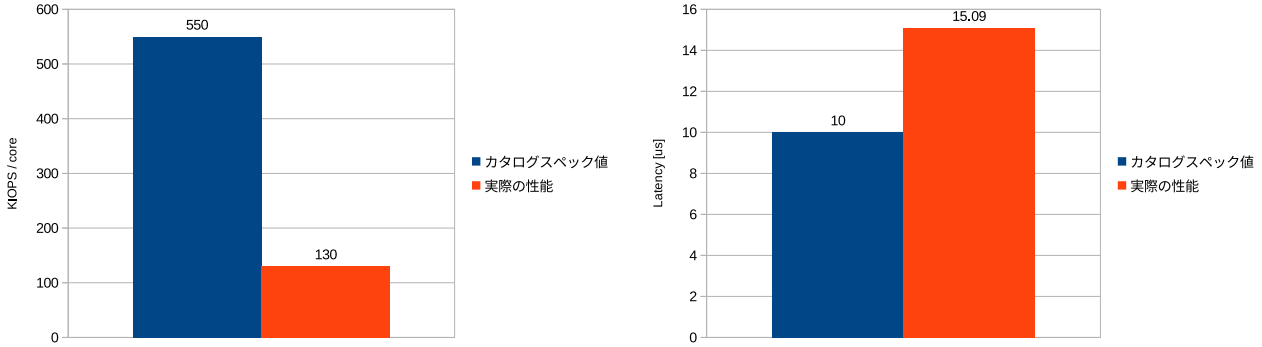
XL-FLASH™ [1], Z-NAND™ [2], Intel® 3D XPoint™ [3]など，低レイテンシを実現した新しいデバイスが開発されている．これらのデバイスを用いたSolid State Drive (SSD)は，従来のNAND型フラッシュメモリを用いたSSDよりも高性能・低レイテンシを実現しており，Ultra-Low Latency SSD (ULL SSD) と呼ばれている．ULL SSDのレイテンシは，約10-15$\mu$s [2,4,5]と報告されており，従来のSSDと比べて約1桁小さい値を達成している．

ULL SSDの特徴は，既存のSSDのインターフェイスを維持しつつ低レイテンシ化を実現したことにより，既存のアプリケーションをそのまま高速化可能という点である．既に存在するファイルシステムをULL SSDに移すことにより，ソフトウェアを変更することなくシステムを高速化可能である．SSDはHDDの互換品として利用されており，Linuxカーネルでは HDDとSSDを大きく区別することなく同じファイルシステムで利用することができる．ULL SSDも同様に，既存のシステムを変更することなく利用可能である．

既存のソフトウェアの多くは，POSIX [6]で記述されている．POSIXとは，IEEEとThe Open Groupが策定したUNIX系OSで利用可能な共通のインターフェイス仕様のことである．世界で最も使われるデータベースであるSQLite [7]は，POSIXで規定されているreadやwriteを利用してファイルIOを実行している．多様なストレージシステムのエンジンとして使われるRocksDB [8]では，IOをC++のクラスで抽象化している．デフォルトでは，POSIX環境を想定した`posix_env`クラスで動作し，IOの並列度はスレッドを用いて表現されている．現代のOSで使われているアプリケーションを調査した研究 [9]によれば，ストレージIOはSQLiteのようなライブラリを通して抽象化されている．アプリケーションが直接POSIX APIを発行することは少ないものの，多くのアプリケーションでは最終的にストレージIOやスレッド操作をPOSIXに依存している．

このように，POSIXは多くのアプリケーションで使われる一方で，POSIXを用いる欠点としてULL SSDのポテンシャル性能を出せないという問題がある．実際，ULL SSDの一種であるIntel® Optane™ SSDを用いてファイルシステム経由で性能を測定したところ，カタログスペック値を大きく下回った．図1(a)が1コアあたりのIOPS性能であり，カタログスペック値の約2割程度の性能しか出なかった．図1(b)はリードレイテンシの結果であり，カタログスペック値の約1.5倍になってしまう．この原因に関しては多くの研究がなされており，原因は大きく2つに分けることができる．1つは割り込みによる性能低下であり [10–12]，もう1つはシステムコールを用いることによ

[1] キオクシア株式会社 メモリ技術研究所
[a)] hidekazu.tadokoro@kioxia.com




(a) 1 コアあたりの IOPS　　(b) レイテンシ

図 1　Optane™ SSD におけるカタログスペック値と実際に測定した性能の比較

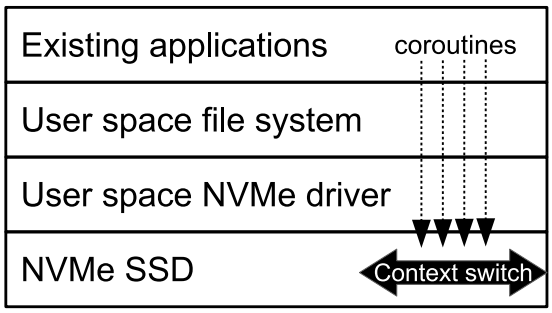



図 2　提案ライブラリの全体像

るオーバーヘッド [11, 13–17] である．

これらの問題を解決する手法も多数提案されている．しかし，既存のアプリケーションと互換性を保ちつつ割り込みとカーネルの介入を完全に排したシステムは，我々の知る限り存在しない．SPDK [11] は，ポーリングを用いて IO の完了を待つことで割り込みを排除し，また，ユーザ空間からストレージにアクセスすることでシステムコールオーバーヘッドを削減している．一方，特殊なスレッドモデルを採用しているため，既存アプリケーションをそのまま動作させることができない．他にも，特殊なハードウェアを利用するもの [13–15] や，カーネルとユーザ空間のハイブリッド方式 [16, 17]，ブロッキング IO の実現にカーネルの介入が必要なもの [18, 19] などがある．

## 2. 提案ライブラリ

これらの問題を解決し，既存のアプリケーションが動作しつつ，ULL SSD のポテンシャル性能を出すことができるライブラリを提案する．図 2 に示すように，このライブラリは IO がユーザ空間で完結することで，カーネルを経由するオーバーヘッドを削減している．ブロッキング IO であってもポーリング動作をすることにより，割り込みを完全に排除している．POSIX インターフェイスを提供することで，既存のアプリケーションがコードの変更なしに動作する．これらを実現するために，スレッド機構とファイルシステムをユーザ空間で提供している．また，ユーザ空間スレッドとユーザ空間ファイルシステムが協調して動作することで，効率的な IO を実現している．

IO の並列性は，アプリケーションの記述を利用している．pthread で書かれたアプリケーションならば，並行に動作する．pthread は，POSIX で規定されたスレッドインターフェイスである．複数スレッドを実行している状態で各スレッドで IO を発行すると，IO 待ちでコンテキストスイッチが発生する．このように，IO 待ち時間に別のスレッドで処理を行うことができるので，CPU 時間を有効に活用できる．

ファイルシステムは Linux でデファクトスタンダードとして使われる ext4 を採用した．このため，カーネルでマウントしている既存の ext4 ファイルシステムを，変更することなく提案ライブラリから利用可能である．ファイルシステムのマイグレーションが不要であり，カーネルファイルシステムでマウントしなおして元のシステムに戻すことも可能である．

### 2.1 ユーザ空間スレッド

提案ライブラリはユーザ空間でスレッド機構を提供しているので，カーネルの介在なしに複数のスレッドを切り替えられる．図 3 がユーザ空間スレッドライブラリの構成である．libcoro [20] と呼ばれるコルーチンライブラリをバックエンドで使うことで，レジスタの退避とスタックポンタの切り替えを行い，軽量なコンテキストスイッチを実現している．

プログラムインターフェイスとして，pthread と互換性を持たせている．これにより，pthread で書かれたアプリケーションを，変更なしに提案ライブラリで動作させることができる．例えばプログラムが `pthread_create` を呼びだすと，ユーザ空間スレッドが生成される．スレッド操作だけでなく，signal や mutex もユーザ空間スレッド専用のものを用意することで，効率的なスレッド間通信を実現している．

このユーザ空間スレッドは，一つのカーネルスレッドを

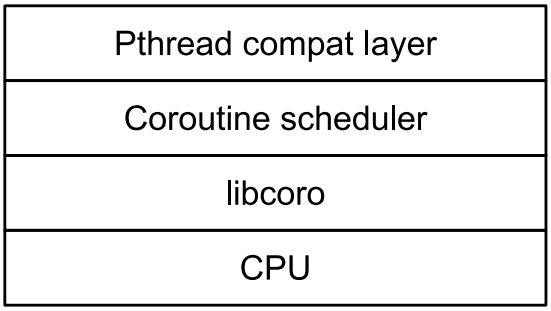



図 3　ユーザ空間スレッドライブラリの構成

共有して動作する．ある瞬間に動作しているユーザ空間スレッドは常に一つである．スレッドスケジューリングは協調的に行われる．明示的に CPU を明け渡すことで，コンテキストスイッチが発生する．スレッドスケジューラは，FIFO による単一キューを実装しており，一度 CPU を明け渡すと，他の実行可能スレッドがすべて実行されることが保証される．

Mutex は，協調的スケジューリングであることを利用し，効率的な実装になっている．動作しているユーザ空間スレッドは常に 1 つのみのため，現在どのスレッドがロックを取得しているかを管理すれば十分である．これは，明示的に CPU を明け渡さないかぎりコンテキストスイッチが発生しないため，compare and swap などの特殊な命令を使わずに実装可能である．ロック待ち行列への追加も，途中でコンテキストスイッチが発生しないので，単純なリスト操作で実現できる．

タイマもユーザ空間で実装しており，タイマスレッドが一括で管理している．タイマスレッドは，スケジュールされるとソートされたタイマをチェックし，発火する必要があるタイマを待っているスレッドを実行可能にする．このため，タイマの数が増えてもスケジューリングオーバーヘッドが増加しない．時間の計測は，起動時に時間を測定するシステムコールを発行し，それ以降はタイムスタンプカウンタを用いて経過時間を見ている．そのため，アプリケーション実行中はシステムコールを発行することなくタイマを実現できている．

### 2.2　ユーザ空間ファイルシステム

提案ライブラリは，ユーザ空間でファイルシステムを提供することで，カーネルの介在なしにファイル IO を実現している．図 4 がこのファイルシステムの全体像である．このファイルシステムは，ブロックデバイスドライバを経由し，ブロックデバイスへ IO を行う．このブロックデバイスドライバとして，ユーザ空間で動作する NVMe™ ドライバを利用することで，ユーザ空間のみで IO を実現している．また，バッファキャッシュを実装しており，読み出しデータのキャシュとデータ書き込みの遅延により，性能を向上させている．

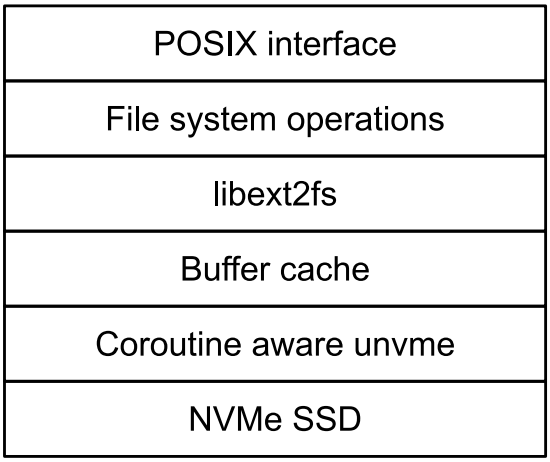



図 4　ユーザ空間ファイルシステムの構成

ファイルシステムとして ext4 を採用しており，その利点として mkfs や fsck などの既存のツールが使える．典型的な運用では，fsck による定期的なファイルシステムの一貫性チェックが行われている．そのため，完成度の高い既存のツールが流用できることは，運用上重要である．

ext4 を実現する方法として，libext2fs [21] を利用している．libext2fs は，mkfs や fsck などのツールを実現するためのライブラリであり，inode 操作など ext4 の基本的な操作を提供している．そのため，POSIX インターフェイスである open や read，ファイルディスクリプタなどを実装した．システムコールレベルだけでなく，stdio や opendir などのライブラリ関数も提供している．

libext2fs は，ブロックデバイスへの IO を発行するインターフェイスが拡張可能になっており，ブロックデバイスへのアクセス方法を切り替えられる．提案ライブラリでは，ここにユーザ空間 NVMe™ ドライバである unvme [22] を使うことで，IO スタックをすべてユーザ空間で実現している．unvme 以外にも，ブロックデバイスドライバとして，カーネルを経由する liburing [23] や libaio [24] を使うことができる．

#### 2.2.1　効率的な IO の発行と完了

提案ライブラリでは，ユーザ空間スレッドとユーザ空間ファイルシステムが連携して動作する．各スレッドが単独で IO を発行したとしても，IO をできる限りまとめて発行することで，効率的に動作する．これは，IO はまとめて発行するほうが効率が良いという前提に基いている．例えば NVMe™ プロトコルでは，複数のリクエストをキューに書いてから MMIO を発行することで，1 回の MMIO で複数のリクエストを送信することができる．また，liburing でも，複数のリクエストをキューに書いてからシステムコールを 1 回発行することで，カーネルに IO 発行を知らせることができる．libaio では，複数のリクエストを一回のシステムコールで発行可能である．

具体例として，NVMe™ プロトコルを使い説明する．

**Code 1** IO 発行時の疑似コード

```
1  void read(struct request* req) {
2      queue.append(req);
3      yield();
4      if (queue.has_pending_request()) {
5          queue.mmio();
6      }
7      while (!queue.is_completed(req)) {
8          yield();
9      }
10 }
```

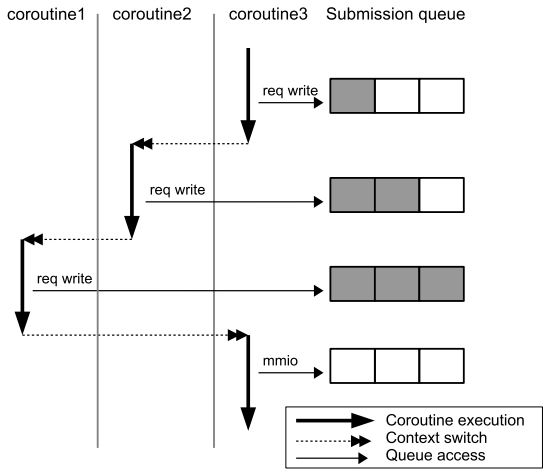



図 **5** 効率的な IO 発行

Code 1 が疑似コードである．IO を発行する手順として，キューに書いてからそのまま MMIO するのではなく，一度 CPU を手放す (2-3 行目)．その後実行が戻ってきたところで，もしキューが空でなければ MMIO を発行する (4-5 行目)．他のスレッドにコンテキストスイッチすることで，そのスレッドが別のリクエストをキューに追加できる (2 行目)．他のスレッドがリクエストを発行しようとしても，既に MMIO されてキューが空になっていれば，なにもする必要がない (4 行目)．io_uring では，MMIO の部分がシステムコールに対応する．libaio ではライブラリで IO リクエストをバッファしておき，複数の IO を一括して発行する．図 5 が，これを模擬的に説明した図になっており，3 つの NVMe™ リクエストを 1 回の MMIO で発行している．

IO 完了時は，効率的にポーリングを行っている．素朴なポーリングは CPU を占有し続けてしまうため，他のスレッドが動作しなくなってしまう．そこで，軽量なコンテキストスイッチを利用して，IO の完了のチェック後に CPU を手放すようにしている (7-8 行目)．軽量なコンテキストスイッチを利用しているため，高い頻度で CPU を手放しても問題がない．他にスレッドが存在しないときは，IO 完了チェックのコードに戻るため，ポーリングによる低レイテンシを実現できる．複数のスレッドが存在する場合，他のスレッドにスイッチするため，他のスレッドの実行を継続できる．図 6 が，3 つのスレッドが同時にポーリングをする例である．

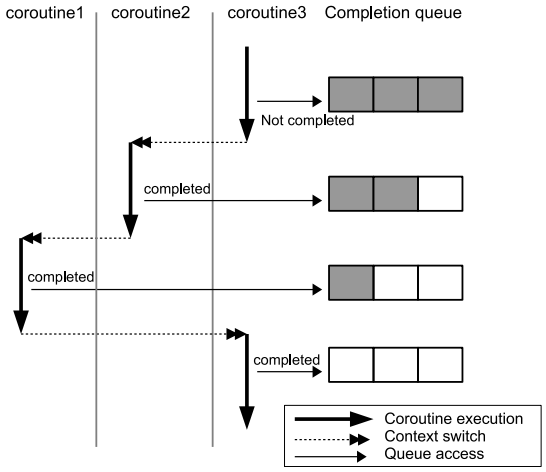



図 **6** 協調的ポーリング

### 2.2.2 バッファキャッシュ

ユーザ空間スレッドと連携することで効率的なバッファキャッシュを実装している．バッファキャッシュの役割は，ブロックデバイスからの読み出し結果をキャシュすることと，ブロックデバイスへの書き出しを遅延させることである．これらの効果により，システムとしての性能を向上させることができる．キャッシュアルゴリズムとして Adaptive Replacement Cache (ARC) [25] をベースに実装した．ARC は LRU と比較して，直近のアクセスだけでなく，長期の傾向も認識することができる．また，LRU を 2 つ組合せているだけなので LRU と同等の実装複雑さである．

常に実行しているスレッドが最大で 1 つであることを利用し，ロックを使わずにバッファキャッシュを実現している．各スレッドはロックを取得することなくバッファキャッシュを操作できる．例えば，読み出しでミスした場合には，新たなページをキャッシュに挿入する必要がある．このとき，保持しているキャッシュのサイズが設定を越えたらば，ARC のアルゴリズムに従って一番古いページを削除する必要がある．リストの継ぎ換えや検索のためのインデックスの更新が必要である．さらに，このページがダーティである場合には，ブロックデバイスへ書き出す必要がある．また，LRU を実現しているリストの継ぎ換えや検索のためのインデックスの更新も必要である．これらの操作を，ロックを使わずに実現している．

バッファキャッシュの全ての操作において，IO を最後に行うことでロックレスを実現している．提案ユーザ空間スレッドの性質として，他に実行中のスレッドは存在しないことが保証されることと，協調的スケジューリングがある．これらの性質により，明示的にコンテキストスイッチしなければ，操作中のデータ構造は他のスレッドから観測されない．そして，明示的なコンテキストスイッチは IO でしか発生せず，IO が発生するのはダーティページの書き出しのみである．また，1 回のページキャッシュの操作

において，ダーティページの書き出しは最大 1 回であり，この IO を最後に実行することが可能である．これらの性質により，必要なデータ構造を操作した後に IO を行うことで，全てのスレッドが一貫したバッファキャッシュへアクセスすることを保証できる．

## 3. 実験

3 つの実験を行った．1 つ目はコンテキストスイッチの実行時間を測定した．提案ライブラリは，効率的な IO を実現するために軽量コンテキストスイッチを利用している．そのため，本当にコンテキストスイッチが軽量であるかを確認する実験を行った．次に，実際にファイルシステムを通して性能が出るかを確認する実験を行った．提案ライブラリは，ファイルシステムを利用しても，性能が出ることを目標としている．最後に，互換性を確認するために SQLite のテストを実行した．これらの 3 つの実験では，POSIX で動作するプログラムを変更することなく，提案ライブラリを用いてユーザ空間で実行した．実験環境は，ULL SSD として Intel® Optane™ SSD 905P を使い，CPU が Xeon® E7-4890 v2，メモリが 512GB のサーバで実行した．

### 3.1 コンテキストスイッチの時間

提案ユーザ空間スレッドライブラリを使い，コンテキストスイッチの時間を測定した．スレッドを 1024 個作り，それぞれ 512K 回コンテキストスイッチを行う時間を測定し，1 回あたりのコンテキストスイッチ時間を算出した．比較のため，pthread を直接使う場合と，提案ライブラリでユーザ空間スレッドに置き換えて実行する場合の 2 つを比較した．実行したコードは Code 2 である．ユーザ空間スレッドライブラリは，pthread の API を置き換えるため，同じコードでも正しく動作する．また，pthread では CPU コアを 1 つに制限して全体の実行時間を測定することで，1 回のコンテキストスイッチあたりの時間を算出した．

図 7 がコンテキストスイッチの時間である．カーネルを利用した場合には，コンテキストスイッチに約 1$\mu$s かかっている．しかし，ユーザ空間で実行した場合には，数 ns でコンテキストスイッチできていることがわかる．ULL SSD のレイテンシが約 10-15$\mu$s であることを考えれば，提案ライブラリのコンテキストスイッチは十分短かく，Section 2.2.1 で説明したようにコンテキストスイッチを多用しても問題ない．一方，カーネルを利用した場合には，このようなコンテキストスイッチを多用した技術はオーバーヘッドが大きいことがわかる．`pthread_yield` は，カーネルのスケジューリング機構を利用するため，システムコールのオーバーヘッドが発生してしまう．

### 3.2 ランダムリード性能

fio [26] を使い，ファイルシステムを経由した場合の 4KB ランダムリード性能を測定した．4GB のファイルを作成し，カーネルファイルシステムを経由した場合の性能と，提案ライブラリを利用した場合の性能を比較した．提案ライブラリは，fio で使われる POSIX インターフェイスを利用するため，実行したコマンドは 2 つの場合で共通[*1]である．並列度を増やしていき，1 コアあたりの IOPS 性能を測定した．カーネルを経由する場合では，1 コアあたりの性能を測定するため，1 コアに制限して実行した．

図 8(a) が IOPS の結果である．並列度が低い場合でも，提案ライブラリのほうが 2 倍近い性能が出ていることがわかる．また，並列度を上げていくと，カーネル実装では性能が頭打ちになってしまうが，提案ライブラリではカタログスペック値の 550KIOPS を越える性能を確認できた．

図 8(b) がレイテンシの結果である．低並列度では，カーネル実装と比較して約半分のレイテンシであり，カタログスペック値の 10$\mu$s よりも良いことがわる．また，並列度を上げていくとカーネル実装では大きくレイテンシが伸びてしまうが，提案ライブラリでは低く抑えられることがわ

**Code 2** コンテキストスイッチをするコード

```c
#include <pthread.h>

#define TASK_N (1024)
#define LOOP_N (512*1024)

static void* f(void* arg) {
  for (int i = 0; i < LOOP_N; i++) {
    pthread_yield();
  }
  return NULL;
}

int main(void) {
  pthread_t t[TASK_N];
  for (int i = 0; i < TASK_N; i++) {
    pthread_create(&t[i], NULL, f, NULL);
  }
  for (int i = 0; i < TASK_N; i++) {
    pthread_join(t[i], NULL);
  }
  return 0;
}
```

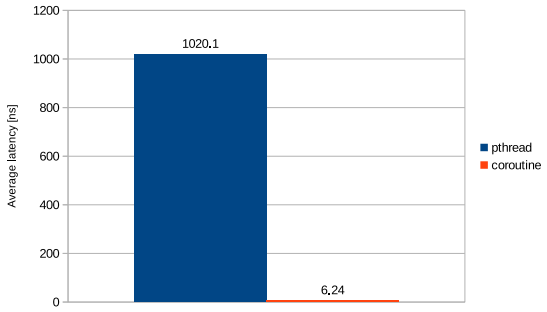
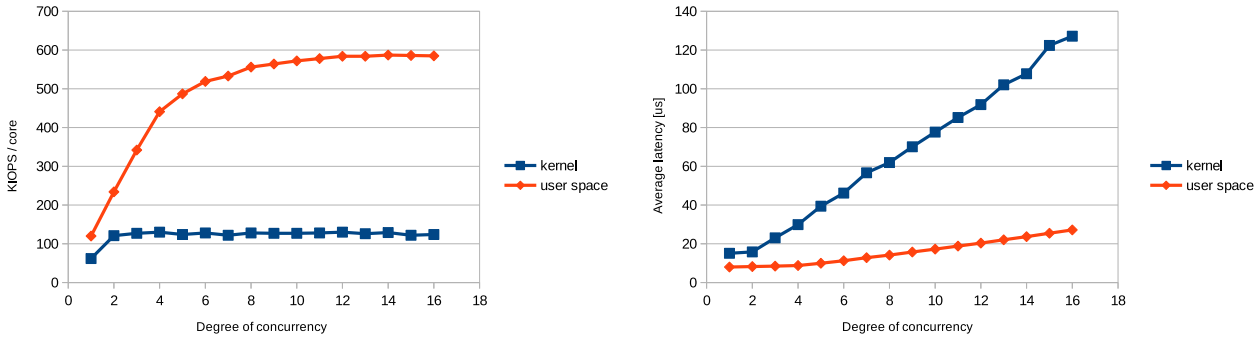



図 7 コンテキストスイッチにかかる時間

[*1] `./fio --ioengine=psync --direct=1 --bs=4k --numjobs=$N --rw=randread --iodepth=1 --filename=/var/tmp/file --name=x --runtime=30 --time_based --size=4GB --thread --group_reporting --norandommap`




(a) 1 コアあたりの IOPS　　(b) レイテンシ

図 8　提案ライブラリとカーネル実装の性能比較

かった．

### 3.3 SQLite 動作テスト

SQLite はとても多くのテストが存在するソフトウェア [27] である．これら SQLite のテストを使い，提案ライブラリがきちんと動作するかを確認する実験をおこなった．テストとして SQLite 付属の `testfixture` の `full.test` を実行した．提案ライブラリはライブラリ OS の一種であり，fork はサポートしていない．そのため，まずは fork が原因で実行不能なテストファイルを取り除いていった．36 個のテストファイルがテストの途中で実行不能になったため，取り除いた．これは fork が必ず成功することを前提にプログラムが書かれていたためである．取り除いたテストファイルは，全部で 1052 個あるテストファイルのうちの約 3.4%にあたる．

これらのテストを実行した結果，75 万以上あるテストケースのうち，176 のみが失敗し，残り 99.97%以上が通ることがわかった．図 9 がその結果の出力である．失敗した 176 のテストケースは，すべて fork できないことによるテストの失敗である．事前に取り除いたテストファイルを考慮しても，多くのテストが通ることがわかった．

## 4. 関連研究

### 4.1 ユーザ空間スレッド

多くのユーザ空間スレッドが提案されている．The GNU portable threads [28] とその後継である the new GNU portable threads [29] は，提案ライブラリと同様に一つのカーネルスレッドを複数のユーザスレッドで共有する．停止する可能性のあるシステムコールを専用のラッパー関数に置き換えることで，全体で実行が止まらないことを保証している．Windows は fiber と呼ばれるユーザ空間スレッドライブラリを提供している [30]．MassiveThreads [31]，Qthreads [32]，Argobots [33] は，HPC 向けのスレッドライブラリである．大量のスレッドを扱うことを目的に，ユーザ空間スレッドを提供している．停止する可能性のあるシステムコールを epoll などイベント駆動型の API で non-blocking に実行することで，ユーザ空間でのスケジューリングを実現している．これらはすべて，ファイルシステムに関して特別な実装を行っておらず，カーネルのものを使うことを前提にしている．

Linux では，setjmp/longjmp [34] や ucontext [35] といった，コンテキストスイッチをユーザ空間で実現する API が提供されている．提案ライブラリで使った libcoro でも，これらの API を利用した実装を選ぶことができる．libcoro は標準では，直接スタックポインタを書き換える実装になっている．

一方で，コルーチンと呼ばれるユーザ空間スレッドは，現在は汎用的には使われない傾向がある．C++にコルーチンを導入しないことを主張する提案書 [36] によれば，コルーチンは様々な OS や言語で失敗に終わり，使われない．実際，Linux や Solaris ではユーザ空間スレッドの提供を止めたこと，主要な言語では Go 言語のみがサポートしてることを指摘している．その Go 言語でも，コルーチンである goroutine は問題が大きいと述べている．また，Facebook の事例では，コルーチンの難しさとして，スタックオーバーフローが多発し不可解なバグに繋がること，同期ライブラリを置き換える必要性などを指摘している．提案ライブラリは，この提案書で否定されているコルーチンを使っている．その理由は，ユーザ空間ファイルシステムの実装で利用している libext2fs のコールスタックごとコンテキストスイッチするためである．

### 4.2 ライブラリ OS

OSv [37] はクラウド環境で効率的にアプリケーションを動かすためのライブラリ OS である．クラウドでは単一のアプリケーションを大量の仮想マシンで動かすため，従来とは違った OS が必要とされている．OSv は，提案システムと同様に，ファイルシステムをライブラリで提供している．しかし，ブロッキング IO を前提に作られており，ULL SSD のような低レイテンシデバイスを活用することは目的

```
176 errors out of 759617 tests on  Linux 64-bit little-endian
!Failures on these tests: expert1-3.1 expert1-3.2 expert1-3.3 expert1-3.4 ...
```

図 **9** SQLite のテストの出力結果

にしていない．Picoprosess [38] は，メモリやスレッド管理など少数の API のみを利用してライブラリ OS を実現することで，既存 OS 上にセキュアな環境を実現するシステムである．セキュリティを目的としたシテムであり，また，ライブラリ側でファイルシステムを提供していない．

### 4.3 ユーザ空間ファイルシステム

FUSE [39] は，ユーザ空間でファイルシステムを実装するためのフレームワークである．FUSE を用いたファイルシステムは，ユーザ空間のプロセスとして実行される．しかし，FUSE はカーネルの VFS レイヤを利用しているため，カーネルを通るオーバーヘッドは削減できない [40]．Direct-FUSE [18] は，FUSE の枠組みを使いつつカーネル空間をバイパスすることで性能を向上させている．FUSE の枠組みを改善しているので，FUSE 用に書かれたすべてのファイルシステムで性能向上を期待できる．一方で，個別のファイルシステムの実装は改善していないため，性能は個別に利用する実装に依存する．例えば，ext4 においては ext4-fuse を利用しているので，ext4-fuse の性質に制約されてしまう．ext4-fuse はグローバルロックを用いた実装のため，スケーラビリティが低いという問題点がある．提案システムは，ULL SSD をターゲットとして，並列性が出るようユーザ空間スレッドと連携したファイルシステムを実現している．

EvFS [19] は，提案システムと同様にユーザ空間でファイルシステムを提供することで，レイテンシを削減している．このファイルシステムは SPDK 上の BlobFS を用いて実装されており，ユーザ空間 NVMe™ ドライバを用いることでユーザ空間で実装している．提案システムと同様に POSIX 互換レイヤを実装することで，既存アプリケーションがそのまま動作する．しかし，同期 IO はセマフォを用いてアプリケーションスレッドを停止・再開させる実装になっている．セマフォ操作はシステムコールであり，カーネル経由のオーバーヘッドは削減できていないと考えられる．提案システムでは，同期 IO をユーザ空間のポーリングで実装しているため，カーネル由来のオーバーヘッドが存在しない．また，ext4 とブロックレベルでの互換性があるので，既存のファイルシステムを利用しやすいという点が異なる．

## 5. まとめと今後の課題

ULL SSD を活用するためのオーバーヘッドの少ないユーザ空間ライブラリを提案した．このライブラリは，スレッド機構とファイルシステムをユーザ空間で提供することで，カーネルの関与と割り込みの利用を完全に排除している．POSIX との互換性により，既存のアプリケーションを変更なしに実行可能である．ファイルシステムは ext4 を実装しているため，既存のファイルシステムを流用し，そのまま動作させることができる．実験により，ファイルシステムを通したランダムリード性能でも，市販 ULL SSD のカタログスペック性能を出すことができた．また，SQLite の多くのテストをパスし，高い互換性を確認した．

今後の課題は，使い易さを考えた外部ライブラリ化である．現在の実装はマクロによって関数呼び出しを置き換える手法であるため，C++などでは動作しない場合がある．書き込み性能のチューニングも課題である．多くのアプリケーションでは，バッファキャッシュに書き込んだ後に非同期に書き込み処理が動く．このため，単に書き込みパスを最適化するだけでなく，バッファキャッシュの実装と合せた最適化が必要である．また，実用的なアプリケーションでの性能評価も課題である．実用性を評価する上で，データベースなどの複雑な IO を行うアプリケーションでの評価が重要である．

参考文献

[1] T. Kouchi, et al.: A 128Gb 1b/Cell 96-Word-Line-Layer 3D Flash Memory to Improve Random Read Latency with tPROG=75$\mu$s and tR=4$\mu$s, *2020 IEEE International Solid-State Circuits Conference*, ISSCC'20.

[2] W. Cheong, et al.: A Flash Memory Controller for 15$\mu$s Ultra-Low-Latency SSD using High-speed 3D NAND Flash with 3$\mu$s Read Time, *2018 IEEE International Solid-State Circuits Conference*, ISSCC'18.

[3] Intel Corporation: 3D XPoint™: A Breakthrough in Non-Volatile Memory Technology, https://www.intel.com/content/www/us/en/architecture-and-technology/intel-micron-3d-xpoint-webcast.html.

[4] Zining Wu: Next Gen Data Storage - Faster, Smater (Flash Memory Summit 2019 Keynote speech), https://www.flashmemorysummit.com/Proceedings2019/08-08-Thursday/20190808_Keynote13_InnoGrit_Wu.pdf.

[5] Intel Coproration: Intel® Optane™ SSD 900P Series, https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/consumer-ssds/optane-ssd-9-series/optane-ssd-900p-series/900p-480gb-aic-20nm.html.

[6] IEEE and The Open Group: POSIX.1-2017, https://pubs.opengroup.org/onlinepubs/9699919799/.

[7] : Most Widely Deployed and Used Database Engine, https://www.sqlite.org/mostdeployed.html.

[8] Facebook, Inc.: A persistent key-value store for fast stor-

age environments, https://rocksdb.org/.
[9] Vaggelis Atlidakis, Jeremy Andrus, Roxana Geambasu, Dimitris Mitropoulos, and Jason Nieh: POSIX Abstractions in Modern Operating Systems: The Old, the New, and the Missing, *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16.
[10] Jisoo Yang, Dave B. Minturn, and Frank Hady: When Poll Is Better than Interrupt, *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12.
[11] Ziye Yang, James R. Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma and Luse E. Paul: SPDK: A development kit to build high performance storage applications, *Proceedings of the 9th IEEE International Conference on Cloud Computing Technology and Science*, CloudCom'17.
[12] Gyusun Lee, Seokha Shin, and Wonsuk Song, Sungkyunkwan University; Tae Jun Ham and Jae W. Lee, Seoul National University; Jinkyu Jeong, Sungkyunkwan: Asynchronous I/O Stack: A Low-latency Kernel I/O Stack for Ultra-Low Latency SSDs, *Proceedings of the 2019 USENIX Annual Technical Conference*, USENIX ATC'19.
[13] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, and Thomas Anderson: Arrakis: The Operating System is the Control Plane, *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'14.
[14] Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau, University of Wisconsin—Madison; Yuangang Wang, Jun Xu, and Gopinath Palani：Designing a True Direct-Access File System with DevFS, *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, FAST'18.
[15] Adrian M. Caulfield, Todor I. Mollov, Louis Alex Eisner, Arup De, Joel Coburn, Steven Swanson: Providing Safe, User Space Access to Fast, Solid State Disks, *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII.
[16] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson: Strata: A Cross Media File System, *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP'17.
[17] Srivatsa S. Bhat, Rasha Eqbal, Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich: Scaling a file system to many coresusing an operation log, *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP'17.
[18] Yue Zhu, Teng Wang, Kathryn Mohror, Adam Moody, Kento Sato, Muhib Khan, Weikuan Yu: Direct-FUSE: Removing the Middleman for High-Performance FUSE File System Support, *Proceedings of the 8th International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS'18.
[19] Takeshi Yoshimura, Tatsuhiro Chiba, and Hiroshi Horii: EvFS: User-level, Event-Driven File System for Non-Volatile Memory, *Proceedings of the 11th USENIX Workshop on Hot Topics in Storage and File Systems*, HotStroage '19.
[20] Marc Alexander Lehmann: libcoro, http://software.schmorp.de/pkg/libcoro.html.
[21] Theodore Ts'o: E2fsprogs: Ext2/3/4 Filesystem Utilities, http://e2fsprogs.sourceforge.net/.
[22] Micron Technology: UNVMe - A User Space NVMe Driver, https://github.com/MicronSSD/unvme.
[23] Jens Axboe: liburing, https://github.com/axboe/liburing.
[24] : Kernel Asynchronous I/O (AIO) Support for Linux, http://lse.sourceforge.net/io/aio.html.
[25] Nimrod Megiddo abd Dharmendra S. Modha: ARC: A Self-Tuning, Low Overhead Replacement Cache, *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, FAST '03.
[26] Jens Axboe: Flexible I/O Tester, https://github.com/axboe/fio.
[27] : How SQLite Is Tested, https://www.sqlite.org/testing.html.
[28] the GNU project: The GNU Portable Threads, https://www.gnu.org/software/pth/.
[29] The GnuPG Project: The New GNU Portable Threads Library, https://gnupg.org/software/npth/.
[30] Microsoft Corporation: Fibers, https://docs.microsoft.com/en-us/windows/win32/procthread/fibers.
[31] J. Nakashima and K. Taura: MassiveThreads: A Thread Library for High Productivity Languages, *Concurrent Objects and Beyond*, pp. 222–238 (2014).
[32] Kyle B. Wheeler, Richard C. Murphy, Douglas Thain: Qthreads: An API for Programming with Millions of Lightweight Threads, *2008 IEEE International Symposium on Parallel and Distributed Processing* (2008).
[33] Sangmin Seo, Abdelhalim Amer, Pavan Balaji, et al.: Argobots: A Lightweight Low-Level Threading and Tasking Framework, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 29, No. 3, pp. 512–526 (2018).
[34] the Linux man-pages project: SETJMP(3), http://man7.org/linux/man-pages/man3/setjmp.3.html.
[35] the Linux man-pages project: GETCONTEXT(3), http://man7.org/linux/man-pages/man3/getcontext.3.html.
[36] Gor Nishanov: Fibers under the magnifying glass, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1364r0.pdf.
[37] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov: OSv—Optimizing the Operating System for Virtual Machines, *2014 USENIX Annual Technical Conference (USENIX ATC 14)*.
[38] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A. Kalodner, Vrushali Kulkarni, Daniela Oliveira, Donald E. Porter: Cooperation and Security Isolation of Library OSes for Multi-Process Applications, *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14.
[39] Miklos Szeredi, et al.: The reference implementation of the Linux FUSE (Filesystem in Userspace) interface, https://github.com/libfuse/libfuse.
[40] Bharath Kumar Reddy Vangoor, Vasily Tarasov, Erez Zadok: To FUSE or Not to FUSE: Performance of User-space File Systems, *Proceedings of the 15th USENIX Conference on File and Storage Technologies*, FAST'17.