平成１９年（行ケ）第１０１９６号　審決取消請求事件
平成２０年８月６日判決言渡，平成２０年７月９日口頭弁論終結

判　　決

原　告　エービー　イニティオ　ソフトウェア　コーポレーション
訴訟代理人弁理士　平木祐輔，関谷三男，渡辺敏章，大塩剛
被　告　特許庁長官　鈴木隆史
指定代理人　山崎達也，吉岡浩，山本章裕，森山啓

主　　文

原告の請求を棄却する。
訴訟費用は原告の負担とする。
この判決に対する上告及び上告受理の申立てのための付加期間を３０日と定める。

事実及び理由

第１　原告の求めた裁判

「特許庁が不服２００４－１０３１８号事件について平成１９年１月２３日にした審決を取り消す。」との判決

第２　事案の概要

本件は，特許出願の拒絶査定に対する不服審判請求を不成立とした審決の取消しを求める事案である。

１　特許庁における手続の経緯

(1)　原告は，平成９年７月１日（パリ条約による優先権主張・１９９６年（平成８年）７月２日，アメリカ合衆国），名称を「ファイルの設定状態の復元」とする発明につき，特許出願（国際出願。以下「本件出願」という。）をした（特願平

１０－５０４４７４号）。

(2)　原告は，平成１６年２月９日付けで，本件出願につき拒絶査定を受けたので，同年５月１７日，拒絶査定不服審判を請求した（不服２００４－１０３１８号事件として係属）。

(3)　特許庁は，平成１９年１月２３日，「本件審判の請求は，成り立たない。」との審決をし，同年２月６日，その謄本を原告に送達した。

２　本願発明の要旨

審決が対象としたのは，平成１６年６月１６日付け手続補正（甲６）により補正された請求項１に記載された発明（以下「本願発明」という。）であり，その要旨は次のとおりである。

「【請求項１】　ネイティブファイルシステムのネイティブファイルおよびディレクトリ動作のセットへトランザクションセマンティクスを追加するためのコンピュータプログラムライブラリを記録したコンピュータ読取り可能な記録媒体であって，前記ライブラリが１つ以上のルーチンファミリのセットを有し，このようなルーチンファミリの各々が最低１つのネイティブファイルまたはディレクトリ動作と関連し，そして，最低１つのネイティブファイルまたはディレクトリ動作の代わりに呼出されるように構成され，このようなルーチンファミリの各々が，

(a)ファミリの関連ネイティブファイルまたはディレクトリ動作の１つと機能的に同等である結果をコンピュータに提供させて，一方でこのようなネイティブファイルまたはディレクトリ動作をロールバックするために必要な情報を記憶するコンピュータ命令を含む実行ルーチンと，

(b)コンピュータに，関連実行ルーチンの結果をコミットさせるコンピュータ命令を含む終了ルーチンと，

(c)コンピュータに，関連実行ルーチンの結果をロールバックさせるコンピュータ命令を含むアンドゥルーチンと，

を有することを特徴とするコンピュータ読取り可能な記録媒体。」

３　審決の理由の要旨

審決は，本願発明は下記引用例に記載された発明（以下「引用発明」という。）に基づいて当業者が容易に発明をすることができたものであるから，特許法２９条２項の規定により特許を受けることができないとした。

引用例　１９９２年（平成４年）１月にサンフランシスコで開催された「１９９２年冬季Ｕｓｅｎｉｘ」の予稿集９頁ないし２５頁に収載されたMargo Seltzerらによる「LIBTP: Portable, Modular Transactions for UNIX」と題する論文（甲１）

審決の理由中，引用例の記載事項及び引用発明の認定，本願発明と引用発明との一致点及び相違点の認定並びに相違点についての判断に係る部分は，以下のとおりである（符号及び明らかな誤記を改めた部分，略称を本判決が指定したものに改めた部分並びに本訴における証拠番号を付記した部分がある。）。

(1)　引用例の記載事項及び引用発明の認定

引用例には，以下のように記載されている。

ア「Although these properties are most frequently discussed in the context of databases, they are useful programming paradigms for more general purpose applications. There are several different situations where transactions can be used to replace current ad-hoc mechanisms.

One situation is when multiple files or parts of files need to be updated in an atomic fashion. For example, the traditional UNIX file system uses ordering constraints to achieve recoverability in the face of crashes. When a new file is created, its inode is written to disk before the new file is added to the directory structure. This guarantees that, if the system crashes between the two I/O's, the directory does not contain are reference to an invalid inode. In actuality, the desired effect is that these two updates

have the transactional property of atomicity (either both writes are visible or neither is). Rather than building special purpose recovery mechanisms into the file system or related tools (e.g. fsck(8)), one could use general purpose transaction recovery protocols after system failure. Any application that needs to keep multiple, related files (or directories) consistent should do so using transactions. Source code control systems, such as RCS and SCCS, should use transaction semantics to allow the “checking in” of groups of related files. In this way, if the“check-in” fails, the transaction may be aborted, backing out the partial “check-in” leaving the source repository in a consistent state.

A second situation where transactions can be used to replace current ad-hoc mechanisms is in applications where concurrent updates to a shared file are desired, but there is logical consistency of the data which needs to be preserved. For example, when the password file is updated, file locking is used to disallow concurrent access. Transaction semantics on the password files would allow concurrent updates, while preserving the logical consistency of the password database. Similarly, UNIX utilities which rewrite files face a potential race condition between their rewriting a file and another process reading the file. For example, the compiler (more precisely, the assembler) may have to rewrite a file to which it has write permission in a directory to which it does not have write permission. While the “.o” file is being written, another utility such as nm(1) or ar(1) may read the file and produce invalid results since the file has not been completely written. Currently, some utilities use special purpose code to handle such cases while others ignore the problem and force users to live with the consequences.」（９～１０頁「1. Introduction」の第２ないし第４段落（９頁２３行～１０頁４行））

（仮訳）

「これらの性質はデータベースに関する文脈でしばしば議論されるが，より一般的な目的のアプリケーションのためにも有用なプログラミングパラダイムである。現に用いられているアドホックなメカニズムに換えてトランザクションを用いることが可能であるいくつかの状況が

ある。

そうした状況の一つは，複数のファイル又は複数のファイルの複数の部分が原子的に(in atomic fashion)更新される必要がある時に生ずる。例えば，伝統的な UNIX ファイルシステムはクラッシュに直面した場合の回復可能性を達成するために順番どおりに（書き込みを）行うという制約を用いている。新しいファイルが生成される時，その新しいファイルがディレクトリ構造に追加される前にそのｉノードがディスクに書き込まれる。このことによって，そのシステムがその二つの I/O の間でクラッシュした場合においても，そのディレクトリが無効な ｉノードの参照を含まないことを保証する。実際には，望まれていた効果は，これらの二つの更新がトランザクションとしての原子性(atomicity)（双方の書き込みが見える(visible)，あるいは，いずれもが見えない(invisible)，のいずれか）を備えていることである。・・・・・・・（中略）・・・・・・・・・・・・

現に用いられているアドホックなメカニズムに換えてトランザクションを用いることが可能であるような二つ目の状況は，共有ファイルに対する同時的な(concurrent)更新が望まれるが，論理的なデータの一貫性は維持されていなければならないようなアプリケーションにおいて生ずる。例えばパスワードファイルを更新するとき，同時的なアクセスを不許可にするためにファイルロッキングが用いられる。パスワードファイルにおいてトランザクションセマンティックスにより，パスワードデータベースの論理的な一貫性を維持しながら，パスワードファイルを同時に更新することが許容される。同様に，ファイルをリライトする UNIX ユーティリティは，それらのファイルのリライトと他のプロセスからの読出しとの間の潜在的な競合状態に直面している。例えば，コンパイラ（正確にはアセンブラ）は書き込み許可を有しないディレクトリ中の書き込み許可を有するファイルをリライトしなければならないかもしれない。その“.o”ファイルが書き込まれている間に nm(1)や ar(1)のような他のユーティリティがファイルを読み，ファイルが完全に書き込まれていないために無価値な結果を生成するかもしれない。現状では，いくつかのユーティリティではこうした状況を取り扱うための特別なコードを用いるが，他のユーティリティではこの問題は無視されてその帰結を受け入れる(live with the consequences)ことをユーザは強いられている。」

イ「In this paper, we present a simple library which provides transaction semantics (atomicity, consistency, isolation, and durability. The 4.4BSD database access methods have been modified to use this library optionally providing shared buffer management between applications, locking, and transaction semantics. Any UNIX program may transaction protect its data by requesting transaction protection with the db(3) library or by adding appropriate calls to the transaction manager, buffer manager, lock manager, and log manager. The library routines may be linked into the host application and called by subroutine interface, or they may reside in a separate server process. The server architecture provides for network access and better protection mechanisms.」（１０頁「1. Introduction」の第５段落（５～１１行））

（仮訳）

「この論文では，トランザクションセマンティクス（原子性，同時性，分離性，永続性）を提供する簡単なライブラリについて解説する。4.4BSD のデータベースアクセスメソッドは，選択的にアプリケーション間での共有バッファ管理，ロッキング及びトランザクションセマンティクスを提供するように，このライブラリを用いて改変された。どのような UNIX プログラムもこの db(3)ライブラリを用いたトランザクション保護を要求するか，又は，トランザクションマネージャ，バッファマネージャ，ロックマネージャ及びログマネージャに対する適切な呼出(call)を追加することによって，そのデータに対するトランザクション保護を行う(transaction protect)ことができる。これらのライブラリルーチンはホストとなるアプリケーションにリンクされてサブルーチンインターフェースにより呼び出されることができる，又は，分離したサーバープロセスとして存在することができる。このサーバーアーキテクチャによりネットワークアクセスとより良い保護メカニズムが提供される。」

ウ「3.2. Module Architecture

The preceding sections described modules for managing the transaction log, locks, and a cache of shared buffers. In addition, we need to provide functionality for transaction begin, commit, and abort processing, necessitating a transaction manager. In order to

arbitrate concurrent access to locks and buffers, we include a process management module which manages a collection of semaphores used to block and release processes. Finally, in order to provide a simple, standard interface we have modified the database access routines (db(3)). For the purposes of this paper we call the modified package the Record Manager. Figure one shows the main interfaces and architecture of LIBTP.

3.2.1. The Log Manager

The Log Manager enforces the write-ahead logging protocol. Its primitive operations are log, log_commit, log_read, log_roll and log_unroll. The log call performs a buffered write of the specified log record and returns a unique log sequence number (LSN). This LSN may then be used to retrieve a record from the log using the log_read call. The log interface knows very little about the internal format of the log records it receives. Rather, all log records are referenced by a header structure, a log record type, and a character buffer containing the data to be logged. The log record type is used to call the appropriate redo and undo routines during abort and commit processing. While we have used the Log Manager to provide before and after image logging, it may also be used for any of the logging algorithms discussed.

The log_commit operation behaves exactly like the log operation but guarantees that the log has been forced to disk before returning. A discussion of our commit strategy appears in the implementation section (section 4.2). Log_unroll reads log records from the log, following backward transaction pointers and calling the appropriate undo routines to implement transaction abort. In a similar manner, log_roll reads log records sequentially forward, calling the appropriate redo routines to recover committed transactions after a system crash.

3.2.2. The Buffer Manager

The Buffer Manager uses a Pool of shared memory to provide a least-recently-used (LRU) block cache. Although the current library provides an LRU cache, it would be simple to add

alternate replacement policies as suggested by [CHOU85] or to provide multiple buffer pools with different policies. Transactions request pages from the buffer manager and keep them pinned to ensure that they are not written to disk while they are in a logically inconsistent state. When page replacement is necessary, the Buffer Manager finds an unpinned page and then checks with the Log Manager to ensure that the write-ahead protocol is enforced.

3.2.3. The Lock Manager

The Lock Manager supports general purpose locking (single writer, multiple readers) which is currently used to provide two-phase locking and high concurrency B-tree locking, However, the general purpose nature of the lock manager provides the ability to support a variety of locking protocols. Currently, all locks are issued at the granularity of a page (the size of a buffer in the buffer pool) which is identified by two 4-byte integers (a file id and page number). This provides the necessary information to extend the Lock Manager to perform hierarchical locking [GRAY76]. The current implementation does not support locks at other granularities and does not promote locks; these are obvious future additions to the system.

If an incoming lock request cannot be granted, the requesting process is queued for the lock and descheduled. When a lock is released, the wait queue is traversed and any newly compatible locks are granted. Locks are located via a file and page hash table and are chained both by object and by transaction, facilitating rapid traversal of the lock table during transaction commit and abort.

The primary interfaces to the lock manager are lock, unlock, and lock_unlock_all. Lock obtains a new lock for a specific object. There are also two variants of the lock request, lock_upgrade and lock_downgrade, which allow the caller to atomically trade a lock of one type for a lock of another. Unlock releases a specific mode of lock on a specific object. Lock_unlock_all releases all the locks associated with a specific transaction.

### 3.2.4. The Process Manager

The Process Manager acts as a user-level scheduler to make processes wait on unavailable locks and pending buffer cache I/O. For each process, a semaphore is maintained upon which that process waits when it needs to be descheduled. When a process needs to be run, its semaphore is cleared, and the operating system reschedules it. No sophisticated scheduling algorithm is applied; if the lock for which a process was waiting becomes available, the process is made runnable. It would have been possible to change the kernel's process scheduler to interact more efficiently with the lock manager, but doing so would have compromised our commitment to a user-level package.

### 3.2.5. The Transaction Manager

The Transaction Manager provides the standard interface of txn_begin, txn_commit, and txn-abort, It keeps track of all active transactions, assigns unique transaction identifiers, and directs the abort and commit processing. When a txn_begin is issued, the Transaction Manager assigns the next available transaction identifier, allocates a per-process transaction structure in shared memory, increments the count of active transactions, and returns the new transaction identifier to the calling process. The in-memory transaction structure contains a pointer into the lock table for locks held by this transaction, the last log sequence number, a transaction state (idle, running, aborting, or committing), an error code, and a semaphore identifier.

At commit, the Transaction Manager calls log_commit to record the end of transaction and to flush the log. Then it directs the Lock Manager to release all locks associated with the given transaction. If a transaction aborts, the Transaction Manager calls on log_unroll to read the transaction's log records and undo any modifications to the database. As in the commit case, it then calls lock_unlock_all to release the transaction's locks.

### 3.2.6. The Record Manager

The Record Manager supports the abstraction of reading and writing records to a database.

We have modified the database access routines db(3) [BSD91] to call the log, lock, and buffer managers. In order to provide functionality to perform undo and redo, the Record Manager defines a collection of log record types and the associated undo and redo routines. The Log Manager performs a table lookup on the record type to call the appropriate routines. For example, the B-tree access method requires two log record types: insert and delete. A replace operation is implemented as a delete followed by an insert and is logged accordingly.」（１２頁４１行～１４頁３９行）

（仮訳）

「３．２．モジュールアーキテクチャ

前のセクションにおいて，トランザクションログ，ロック，共有バッファキャッシュの管理を行うためのモジュールが記述された。それに加えて，トランザクションの開始処理，コミット処理及びアボート処理の機能が提供されなければならず，トランザクションマネージャが不可欠である。ロックとバッファに対する競合する同時的なアクセスを調停するために，プロセスのブロックとリリースに用いられるセマフォの集合を管理するプロセス管理モジュールを含めることとする。最後に簡単で標準化されたインターフェースを提供するために，データベースアクセスルーチン(db(3))を変更した。この論文においては，この変更されたパッケージをレコードマネージャと呼ぶこととする。図１に LIBTP の主なインターフェースとアーキテクチャが示されている。

３．２．１．ログマネージャ

ログマネージャはライトアヘッドロギングプロトコルを強制する。そのプリミティブ操作は log，log_commit，log_roll，log_unroll である。log 呼出は，特別なログレコードに対するバッファされた書き込みを実行して，唯一のログ順序番号(LSN)を返す。この LSN は log_read 呼出を用いてログからレコードを獲得するために用いられる。この log インターフェースは，それが受け取るログレコードの内部形式についてはほとんど知らない。むしろ全てのログレコードは，ヘッダの構造，ログレコードの種類とログとして書き込まれるべきデータを含むキャラクタバッファによって参照される。このログレコードの種別は，アボート処理とコミット処理

の間に適切な redo と undo のルーチンを呼び出すために用いられる。ここではログマネージャを用いて事前及び事後のログを提供しているが，議論されたロギングアルゴリズムのいずれを用いることも可能であろう。

log_commit 操作は，確かに log 操作と同じように振舞うが，戻る前にディスクへの書き込みが強制される。私たちが採用するコミット戦略についての議論は実装に関する節（section 4.2）において行う。Log_unroll はログからログレコードを読み，後ろ向きのトランザクションポインタを伴って，トランザクションのアボートを実装した適切な undo ルーチンを呼び出す。同様なやり方で，Log_roll はログレコードを前向きに連続して読み，システムのクラッシュ後に既にコミット済みトランザクションの回復を行うための適切な redo ルーチンを呼び出す。

３．２．２．バッファマネージャ

バッファマネージャは，共有メモリのプールを使って least-recently-used(LRU)のブロックキャッシュを提供する。現在のライブラリはＬＲＵキャッシュを提供しているが，［CHOU85］に示唆されるように代替の置換戦略を加えたり，異なる戦略による複数のバッファプールを提供することは簡単であろう。トランザクションはバッファマネージャにページを要求し，これらを常駐化して論理的に非定常状態にある間はディスクへの書き込みが行われないことを保証する。ページの置換が必要なときは，バッファマネージャは常駐化されていないページを探してそしてログマネージャにおいてライトアヘッドプロトコルが確実に強制されるようにチェックする。

３．２．３．ロックマネージャ

ロックマネージャは，２相ロッキングと高い並列度を有する B-tree ロッキングを提供するために用いられる一般的な目的のロッキング（単一書き込み者と複数の読出し者）をサポートする。しかしながら，ロックマネージャが一般的な目的のものであるという性質によって，さまざまなロッキングプロトコルをサポートする能力が提供される。現状では，全てのロックは２つの４バイト整数（ファイル id とページ番号）によって識別されるページの粒度（バッファプールのバッファの大きさ）で発行される。これは，ロックマネージャを階層的ロッキング

を実行するように拡張するための必要な情報を提供する。現在の実装では他の粒度のロックをサポートしていないし，ロックのプロモートを行わない。・・・・・（中略）・・・・

３．２．５．トランザクションマネージャ

トランザクションマネージャは，txn_begin，txn_commit，txn_abort の標準インターフェースを提供する。それは，全てのアクティブなトランザクションの進路を追い，唯一のトランザクション識別子を割当て，そしてアボート処理とコミット処理を指揮する。txn_begin が発行されるとき，トランザクションマネージャは次に利用可能なトランザクション識別子を割当て，共有メモリ上にプロセス毎のトランザクション構造をアロケートし，アクティブトランザクション数をインクリメントし，呼出元プロセスに対して新しいトランザクション識別子を返す。メモリ内のトランザクション構造は，そのトランザクションによって保持されているロックのためのロックテーブルへのポインタ，過去最近のログ順序番号，トランザクションの状態（アイドル，実行中，アボート処理中，コミット処理中），エラーコード及びセマフォアの識別子を含む。

コミットの際，トランザクションマネージャは log_commit を呼び出してトランザクションの終了を記録し，ログをディスクへ書き出す(flush)。そして，ロックマネージャにそのトランザクションに関連して保持された全てのロックを開放するように指示する。もしトランザクションがアボートしたら，トランザクションマネージャは log_unroll をコールしてトランザクションのログレコードを読み，データベースになされた変更について undo する。それから，コミットの場合と同様に，トランザクションのロックを開放するために lock_unlock_all を呼び出す。

３．２．６．レコードマネージャ

レコードマネージャは抽象化されたデータベースへのレコードの読出しと書き込みをサポートする。データベースアクセスルーチンの db(3)がログ，ロック及びバッファの各マネージャを呼び出すように変更された。undo と redo を実行する機能を提供するために，レコードマネージャはログレコードの種別と関係する undo と redo のルーチンを定義する。ログマネージャがそのレコード種別をテーブルから探し，適切なルーチンを呼び出す。例えば B-tree アクセ

スメソッドは２つのログレコード種別，挿入と削除，を必要とする。置き換え操作は挿入が引き続くような削除として実装され，そのようにログに記録される。」

エ「3.3. Application Architectures

The structure of LIBTP allows application designers to trade off performance and protection. Since a large portion of LIBTP's functionality is provided by managing structures in shared memory, its structures are subject to corruption by applications when the library is linked directly with the application. For this reason, LIBTP is designed to allow compilation into a separate server process which may be accessed via a socket interface. In this way LIBTP's data structures are protected from application code, but communication overhead is increased. When applications are trusted, LIBTP may be compiled directly into the application providing improved performance. Figures two and three show the two alternate application architectures.

There are potentially two modes in which one might use LIBTP in a server based architecture. In the first, the server would provide the capability to respond to requests to each of the low level modules (lock, log, buffer, and transaction managers). Unfortunately, the performance of such a system is likely to be blindingly slow since modifying piece of data would require three or possibly four separate communications: one to lock the data, one to obtain the data, one to log the modification, and possibly one to transmit the modified data. Figure four shows the relative performance for retrieving a single record using the record level call versus using the lower level buffer management and locking calls. The 2:1 ratio observed in the single process case reflects the additional overhead of parsing eight commands rather than one while the 3:1 ratio observed in the client/server architecture reflects both the parsing and the communication overheard. Although there may be applications which could tolerate such performance, it seems far more feasible to support a higher level interface, such as that provided by a query language (e.g. SQL [SQL86]).

Although LIBTP does not have an SQL parser, we have built a server application using the toolkit command language (TCL) [OUST90]. The server supports a command line interface similar to the subroutine interface defined in db(3). Since it is based on TCL, it provides control structures as well.」（１４頁４０行～１５頁１１行）

（仮訳）

「３．３．アプリケーションアーキテクチャ

LIBTP の構造によって，アプリケーションプログラム設計者が性能と保護のトレードオフを行えるようになる。LIBTP の機能の大部分は共有メモリの構造を管理することによって与えられるものであり，そうした構造はこのライブラリがアプリケーションにリンクされたときには，そのアプリケーションにおける『なまり』(corruption)に支配される。このため，LIBTP はソケットインターフェースを通じてアクセスできるような分離したサーバプロセスとしてのコンパイレーションを許容するように設計されている。このようにすると，LIBTP のデータ構造はアプリケーションコードから保護されるが，通信のオーバーヘッドは増加する。アプリケーションが信頼できるときには，性能を向上させるために LIBTP をアプリケーション中に直接組み込むようにコンパイルすることができる。図２と図３にはその二つの代替的なアプリケーションアーキテクチャが示されている。

・・・・・・（中略）・・・・・・・・

そのような性能を許容できるようなアプリケーションがあるのか否かはともかく，あたかも質問言語において提供されているかのような高いレベルのインターフェースをサポートすることはとても実現性が高い。LIBTP は SQL パーザを備えていないが，私たちはツールキットコマンド言語(TCL)を用いてサーバアプリケーションを作成した。そのサーバは db(3)において定義されたサブルーチンインターフェースに類似したコマンドラインのインターフェースをサポートする。

・・・・・・・・・・（以下略）・・・・・・・」

オ「4.4. Transaction Protected Access Methods

The B-tree and length recno (record number) access methods have been modified to provide

transaction protection. Whereas the previously published interface to the access routines had separate open calls for each of the access methods, we now have an integrated open call with the following calling conventions:

```
DB *dbopen (const char *file, int flags, int mode, DBTYPE type,
            int dbflags, const void *openinfo)
```

where file is the name of the file being opened, flags and mode are the standard arguments to open(2), type is one of the access method types, dbflags indicates the mode of the buffer pool and transaction protection, and openinfo is the access method specific information. Currently, the possible values for dbflags are DB_SHARED and DB_TP indicating that buffers should be kept in a shared buffer pool and that the file should be transaction protected.

The modifications required to add transaction protection to an access method are quite simple and localized.

1. Replace file open with buf_open.
2. Replace file read and write calls with buffer manager calls (buf_get, buf_unpin).
3. Precede buffer manager calls with an appropriate (read or write) lock call.
4. Before updates, issue a logging operation.
5. After data have been accessed, release the buffer manager pin.
6. Provide undo/redo code for each type of log record defined.

The following code fragments show how to transaction protect several updates to a B-tree. In the unprotected case, an open call is followed by a read call to obtain the meta-data for the B-tree. Instead, we issue an open to the buffer manager to obtain a file id and a buffer request to obtain the meta-date as shown below.

```
char *path;
int fid, flags, len, mode;
/* Obtain a file id with which to access the buffer pool */
```

```
fid = buf_open(path, flags, mode);
/* Read the meta data (page 0) for the B-tree */
if (tp_lock(fid, 0, READ_LOCK))
        return error;
meta_data_ptr = buf_get(fid, 0, BF_PIN, &len);
```

The BF_PIN argument to buf_get indicates that we wish to leave this page pinned in memory so that it is not swapped out while we are accessing it. The last argument to buf_get returns the number of bytes on the page that were valid so that the access method may initialize the page if necessary.

Next, consider inserting a record on a particular page of a B-tree. In the unprotected case, we read the page, call_bt_insertat, and write the page. Instead, we lock the page, request the buffer, log the change, modify the page, and release the buffer.

```
int fid, len, pageno; /* Identifies the buffer */
int index; /* Location at which to insert the new pair */
DBT *keyp, *datap; /* Key/Data pair to be inserted */
DATUM *d; /* Key/data structure to insert */
/* Lock and request the buffer */
if (tp_lock(fid, pageno, WRITE_LOCK))
        return error;
buffer_ptr = buf_get(fid, pageno, BF_PIN, &len);
/* Log and perform the update */
log_insdel(BTREE_INSERT, fid, pageno, keyp, datap);
_bt_insertat(buffer_ptr, d, index);
buf_unpin(buffer_ptr);
```

Succinctly, the algorithm for turning unprotected code into protected code is to replace read operations with lock and buf-get operations and write operations with log and

buf_unpin operations.」（１７頁２６行～１８頁末行）

（仮訳）

「４．４．トランザクションによる保護を受けるアクセスメソッド

B-Tree と固定長レコードナンバーへのアクセスメソッドがトランザクションによる保護を提供するように変更された。これらのルーチンに対する以前に公開されたインターフェースにおいて，それぞれのアクセスメソッドに対する分離したオープン呼出しを備えていたが，その代替となる以下の呼出しによる統合したオープンコールを用いることとした。

DB *dbopen（以下，省略）

ここで，file はこれからオープンされようとしているファイル名，flags と mode は open(2)における標準の引数，type はアクセスメソッドのタイプの一つ，dbflags はバッファプールとトランザクション保護のモードを指示，openinfo はそのアクセスメソッドに特有の情報である。現時点では，dbflags の値として可能な値は DB_SHARED と DB_TP であり，これらの値はそれぞれバッファを共有バッファプールに保持すべきであることと，そのファイルに対してトランザクション保護がされるべきであるということを指示するものである。

アクセスメソッドに対してトランザクション保護を追加するために要求される変更は，たいへん簡単で局所的である。

１．ファイルの open コールを buf_open で置き換える。

２．ファイルの read と write コールをそれぞれ対応するバッファマネージャに対する呼出(buf_get,buf_unpin)で置き換える。

３．バッファマネージャに対する呼出の前に適当な（読出しあるいは書き込み用の）ロック呼出を置く。

４．更新処理の前にログ処理を発行する。

５．データがアクセスされた後に，バッファマネージャによる常駐をリリースする。

６．ログレコードによって定義される各々の種類に応じて undo/redo を提供する。

次のコードの断片は B-tree に対する複数の更新についてどのようにしてトランザクションによる保護を行うかを示すものである。保護がない場合であれば open 呼出に続いて B-tree の

メタデータを獲得するための read 呼出がなされるが，ここではそうせずに，次に示すように，ファイル id を獲得するためにバッファマネージャに対するオープン呼出と，そのメタデータを獲得するためのバッファリクエストを発行する。

char *path;（以下，省略）

buf_get 呼出の引数である BF_PIN は，アクセスを行っている間にこのページをスワップアウトしないように，このページをメモリに常駐化したままとすることを望んでいることを指示する。buf_get の最後の引数は，必要に応じてアクセスメソッドにおいて初期化することができるように有効とされたページのバイト数を返す。

次に B-tree の特定のページにレコードを挿入してみよう。保護がない場合であれば，ページを読み，_bt_insertat を呼出し，ページを書き出すが，ここではそうせずに，ページをロックし，バッファを要求し，ログに変更を記録し，ページを変更し，そしてバッファをリリースする。

int fid, len, pageno;（以下，省略）

簡単にいえば，保護されていないコードを保護されたコードに改良する手順は，read 操作を lock と buf_get 操作に置き換え，write 操作を log と buf_unpin 操作で置き換えることである。」と，記載されている。

すなわち，引用例には，

「伝統的な UNIX ファイルシステムにおける新しいファイルの生成等のように複数のファイルの複数の部分が原子的に更新される必要がある場合や共有ファイルに対する同時的な更新が望まれる一方で論理的なデータの一貫性を維持する必要が生ずる場合に，ライブラリに属するルーチンに対する適切な呼出を追加することによって，トランザクションセマンティクスを提供するための該ライブラリ（LIBTP）であって，

前記ライブラリがログマネージャ，バッファマネージャ，ロックマネージャ，トランザクションマネージャの各々を実現するルーチンを含むものであって，これらのルーチンにおいては，ログマネージャの実行によりコミット処理及びアボート処理の際参照される事前及び事後のログ取得処理を行い，トランザクションマネージャ等を実現する各ルーチンの実行により

txn_commit プリミティブによって起動されるコミット処理を行い，トランザクションマネージャ等を実現する各ルーチンの実行により txn_abort プリミティブによって起動されるアボート処理を行うライブラリを含むことを特徴とするファイルの更新処理方法」に関する発明が開示されている（引用発明）。

(2)　本願発明と引用発明との一致点及び相違点の認定

本願発明と，引用発明とを対比すると，

両者は共に，コンピュータプログラムのトランザクション処理に関するものであって，

引用発明の「伝統的な UNIX ファイルシステム」が本願発明の「ネイティブファイルシステム」に相当し，

引用発明における「新しいファイルの生成等のように複数のファイルの複数の部分が原子的に更新される必要がある場合や共有ファイルに対する同時的な更新が望まれる一方で論理的なデータの一貫性を維持する必要が生ずる場合」に実行されているファイル生成等の具体的なファイル操作は，トランザクションの保護の対象となるファイル操作を含む実行ルーチンであるので，これは，本願発明の「ネイティブファイルシステムにおけるファイル及びディレクトリの操作」を含むルーチンに対応するとともに「関連実行ルーチン」に相当し，

引用発明における「ライブラリ（LIBTP）」は，ネイティブファイルシステムに必要に応じてトランザクションセマンティクスを追加あるいは提供するためのコンピュータプログラムルーチンの集合であるので，これは本願発明の「ライブラリ」に相当し，

引用発明におけるライブラリを構成するルーチン（あるいはルーチンファミリ）である「ログマネージャ，バッファマネージャ，ロックマネージャ，トランザクションマネージャの各々を実現するルーチン」の各々は，本願発明の「ルーチン」あるいは「ルーチンファミリ」に相当し，又，引用発明においてもこれらのルーチンが複数あり，本願発明と同様に，「セット」を構成していると言うことができる。

引用発明の「コミット処理及びアボート処理の際参照される事前及び事後のログを取得する処理」は，トランザクション処理におけるログ取得の処理であるので，本願発明の「ファイル又はディレクトリ動作をロールバックするために必要な情報を記憶するコンピュータ命令を含

む実行ルーチン」に相当し，

引用発明の「txn_commit プリミティブによって起動されるコミット処理」は，トランザクション処理におけるコミット処理であるので，本願発明の「関連実行ルーチンの結果をコミットさせるコンピュータ命令を含む終了ルーチン」に相当し，

引用発明の「txn_abort プリミティブによって起動されるアボート処理」は，トランザクション処理におけるロールバック処理であるので，本願発明の「関連実行ルーチンの結果をロールバックさせるコンピュータ命令を含むアンドゥルーチン」に相当している。

したがって，両者は共に，

ネイティブファイルシステムのネイティブファイルおよびディレクトリ動作のセットへトランザクションセマンティクスを追加するためのコンピュータプログラムライブラリであって，前記ライブラリが１つ以上のルーチンファミリのセットを有し，このようなルーチンファミリの各々が，

(a)このようなネイティブファイルまたはディレクトリ動作をロールバックするために必要な情報を記憶するコンピュータ命令を含む実行ルーチンと，

(b)コンピュータに，関連実行ルーチンの結果をコミットさせるコンピュータ命令を含む終了ルーチンと，

(c)コンピュータに，関連実行ルーチンの結果をロールバックさせるコンピュータ命令を含むアンドゥルーチンと，

を有することを特徴とするコンピュータプログラムライブラリ

である点で一致する。

ただ，

相違点(1)

本願発明においては，「ルーチンファミリの各々が最低１つのネイティブファイルまたはディレクトリ動作と関連し，そして最低１つのネイティブファイルまたはディレクトリ動作の代わりに呼び出されるように構成され」，(a)の実行ルーチンにおいて「ファミリの関連ネイティブファイルまたはディレクトリ動作の１つと機能的に同等である結果をコンピュータに提供

させる」とともに，このようなルーチンファミリ「の各々」が上記の(a)から(c)の処理を実現するルーチンを有する，すなわち，ネイティブファイルまたはディレクトリ動作についてのトランザクションを実現するために必要な処理を呼び出す際ネイティブファイルシステム上でネイティブファイル又はディレクトリ操作の呼出のための API をそのまま用いて，かつ，ライブラリに属するルーチンの実行によってネイティブファイル又はディレクトリ動作の機能を実現するように構成されているのに対し，引用発明においては，ネイティブファイル又はディレクトリ操作の呼出のための API とは異なった追加された API(txn_commit プリミティブ等)を用いるように構成され，ライブラリに属するルーチンの実行ではなくネイティブファイルシステムの実行によってネイティブファイル又はディレクトリ動作の機能を実現するように構成されている点，

及び，

相違点(2)

本願発明は「コンピュータ読取り可能な記録媒体」に関する発明であるのに対して，引用発明はコンピュータプログラムライブラリに関する発明である点，

で両者は相違している。

(3)　相違点についての判断

ア　相違点(1)について

例えば，特開平６－１４９６３３号公報（甲３）や特開平６－３５７８２号公報（甲４）に見られるように，一般にシステムコールを用いて実現されている機能の拡張や機能の追加を行うにあたって，システムコールの API と同じ API によって呼び出されそのシステムコールの機能と拡張機能とを実現するように構成されたルーチンを用いることによって，システムコールの呼び出し側であるアプリケーションの修正の手間を軽減あるいは不要にする技術は周知技術である。

また，引用発明においてもトランザクションセマンティクスの導入という機能追加を行うにあたってアプリケーション修正の手間を軽減することは当業者にとって自明の事項である。

又，一般に，複数のルーチンにより実現されている機能を１つのルーチンで実現することは

当業者が通常行うことであって，このことはシステムコールを実現するルーチンについても同様であると言うことができる。

してみると，引用発明においても，こうした周知技術を採用して，ネイティブファイルシステム上でネイティブファイル又はディレクトリ操作の呼出のための API をそのまま用いることを可能とするために，ライブラリに属するルーチンの実行によってネイティブファイル又はディレクトリ動作の機能を実現するように構成して本願発明のようにすることは，当業者が適宜選択設計すべきことであり，その効果も当業者が通常予測すべき範囲のものにすぎないものである。

したがって，相違点(1)は格別の相違点であるものと言うことはできない。

イ　相違点(2)について

一般に，コンピュータ上でコンピュータプログラムライブラリを実行するためには，該プログラムライブラリをコンピュータ読取り可能な記録媒体に記憶させて実行させなければならないものである。したがって，相違点(2)も格別の相違点であるものとは言えない。

したがって，本願発明は，引用発明から当業者が容易に発明することができたものである。

(4)　審決の「むすび」

以上の通りであるので，本願発明は，引用発明から当業者が容易に発明をすることができたものであり，特許法２９条２項の規定により特許を受けることができない。

## 第３　当事者の主張の要点

### １　原告主張の審決取消事由の要点

審決は，以下のとおり，引用発明の認定及び本願発明と引用発明の一致点の認定をそれぞれ誤り，また，両発明の相違点を看過した結果，本願発明が特許法２９条２項の規定により特許を受けることができないと判断したものであるから，取り消されるべきである。

(1)　前提となる技術内容（データベースシステムとファイルシステム）等について

ア(ア)　一般に，コンピュータシステムは，データベースシステムとファイルシステムに分類され，前者においては，データはデータベースに保管され，後者においては，データはファイルに保管される。また，前者におけるトランザクションは，データ動作（データの読み出し及び書き込み。引用例に記載されているものはこれに相当する。）に関するものであり，後者におけるトランザクションは，ファイル動作（ファイルの作成，削除，名前変更等。本願発明の「ネイティブファイルシステムのネイティブファイルおよびディレクトリ動作」はこれに相当する。）に関するものである。

(イ)ａ　この点を敷衍すると，本件出願に係る平成１６年６月１６日付け手続補正書（甲６）による補正後の明細書（特許請求の範囲につき甲６，その余につき甲５。以下「本願明細書」という。なお，これを引用する場合に掲記する引用箇所は，甲５に係るものである。）の記載（６頁２１～２４行）によれば，複合データ処理アプリケーションの操作動作には，「ファイル（操作）動作」と「データ（操作）動作」の２つが存在し，技術的に異なるものとして区別されていることが明らかであるところ，本願明細書及び本件出願に係る図面（甲５）においては，本願発明の実施例として，「ファイルおよびディレクトリ動作」（なお，「ディレクトリ」とは，ファイルの集合を表す単位である。）が，「データ動作」と明確に区別して記載されている。したがって，本願発明の「ネイティブファイルおよびディレクトリ動作」は，「データ動作」を包含するものではない。

ｂ　これに対し，引用例においては，「ファイルおよびディレクトリ動作」にトランザクションセマンティクスを追加することについての記載や，そのためのルーチンファミリについての記載はない。

イ　従来，トランザクションは，データベースシステムにおいて利用されており，ファイルシステムにおいては利用されていなかった（多くのオペレーティングシステムは，本来的に，ファイル及びディレクトリ動作のためのトランザクション処理を提供することができなかった。）。

ウ　本願発明の目的は，データベースシステムにおいて発展してきたトランザクション処理技術をファイルシステムに適用することにあり，本願発明は，ファイル及びディレクトリ動作にトランザクションセマンティクスを付与するものであるところ，引用例には，トランザクション処理技術をファイルシステムに適用することについての記載がない。

(2)　取消事由１（引用発明の認定の誤り）

審決は，「伝統的な UNIX ファイルシステムにおける新しいファイルの生成等のように複数のファイルの複数の部分が原子的に更新される必要がある場合や共有ファイルに対する同時的な更新が望まれる一方で論理的なデータの一貫性を維持する必要が生ずる場合に，ライブラリに属するルーチンに対する適切な呼出を追加することによって，トランザクションセマンティクスを提供するための該ライブラリ（LIBTP）であって，前記ライブラリがログマネージャ，バッファマネージャ，ロックマネージャ，トランザクションマネージャの各々を実現するルーチンを含むものであって，これらのルーチンにおいては，ログマネージャの実行によりコミット処理及びアボート処理の際参照される事前及び事後のログ取得処理を行い，トランザクションマネージャ等を実現する各ルーチンの実行により txn_commit プリミティブによって起動されるコミット処理を行い，トランザクションマネージャ等を実現する各ルーチンの実行により txn_abort プリミティブによって起動されるアボート処理を行うライブラリを含むことを特徴とするファイルの更新処理方法」に関する発明を，引用発明と認定したが，以下のとおり，この認定は誤りである。

ア　審決は，引用発明を，「伝統的な UNIX ファイルシステムにおける新しいファイルの生成等のように複数のファイルの複数の部分が原子的に更新される必要がある場合」（以下「審決が認定した第１の場合」という。）と「共有ファイルに対する同時的な更新が望まれる一方で論理的なデータの一貫性を維持する必要が生ずる場合」（以下「審決が認定した第２の場合」という。）に，トランザクションセマンティクスを提供するためのライブラリ（ＬＩＢＴＰ）の各ルーチンを実行する

ものと認定した。

しかしながら，審決が認定した第１の場合及び第２の場合は，トランザクションが必要な一般的な場合とそれに対する従来の解決方法についての記載である引用例の「１．序論」（なお，以下，外国語で作成された書証中の記載の引用及びその箇所の特定は，翻訳文による。）に記載され，他方，引用例の「３．アーキテクチャ」，「４．実装」等には，引用例の主要なテーマである，データベースシステムに対するデータ動作にトランザクションセマンティクスを付与する技術であるライブラリ（ＬＩＢＴＰ）が記載されているにとどまり（したがって，ファイル及びディレクトリ動作にトランザクションセマンティクスを付与するものではない。），「１．序論」に記載された内容と，「３．アーキテクチャ」，「４．実装」等に記載された内容とを結合した１つの発明は，引用例のどこにも記載されていない（「１．序論」には，従来技術の問題点や引用発明（ライブラリ（ＬＩＢＴＰ））の目的は記載されているが，審決が認定した引用発明（ライブラリ（ＬＩＢＴＰ））の必須の構成は記載されていないし，「１．序論」に記載された構成と「３．アーキテクチャ」，「４．実装」等に記載された構成とが技術的に関連して所定の機能を提供するものとも考えられない。）のであるから，両者を結合した１つの発明を引用発明と認定することは不当である。

なお，被告は，「引用例の『３．アーキテクチャ』，『４．実装』等の記載は，『１．序論』の記載を受け，『従来のＵＮＩＸファイルシステム』にトランザクションセマンテイクスを付与する技術について論じたものである」と主張するが，引用例には，「従来のＵＮＩＸファイルシステム」にトランザクションセマンティクスを付与する技術は記載されていない。

また，被告の主張するとおり，引用例の「３．アーキテクチャ」，「４．実装」等の記載が「１．序論」の記載を受けたものであるならば，被告は，引用例に１つの発明が記載されていないことを自認していることになる。

イ　引用例の「１．序論」に記載された内容と，「３．アーキテクチャ」，「４．

実装」等に記載されたライブラリ（ＬＩＢＴＰ）とは，以下のとおり，技術的な整合性を有しないから，両者を結合しても，１つの発明を構成することはできない。

(ｱ)　審決が認定した第１の場合は，引用例の「１．序論」の第３段落の記載に相当するものであるが，同段落には，審決が認定した第１の場合に，「（書き込みについての）順序制約（ordering constraints）を使用して，クラッシュに直面した場合の回復可能性を達成する」と記載されており，トランザクションセマンティクスを提供するためのライブラリ（ＬＩＢＴＰ）の各ルーチンを実行するとは記載されていない。

(ｲ)　審決が認定した第２の場合は，引用例の「１．序論」の第４段落の記載に相当するものであるが，同段落には，審決が認定した第２の場合の例として，「パスワードファイルが更新される場合」及び「ファイル書き換えとファイルを読み取る別のプロセス間の潜在的競合条件に直面」したとき（これらは，いずれも，データベースにおける処理を問題とするものである。）が挙げられ，トランザクションセマンティクスを提供するためのライブラリ（ＬＩＢＴＰ）の各ルーチンを実行するとは記載されていない。

ウ　審決は，引用発明を，「伝統的なＵＮＩＸファイルシステムにおける新しいファイルの生成等のように複数のファイルの複数の部分が原子的に更新される必要がある場合・・・に，・・・トランザクションセマンティクスを提供するための該ライブラリ（ＬＩＢＴＰ）であって，・・・を行うライブラリを含むことを特徴とするファイルの更新処理方法」に関する発明と認定した。

しかしながら，引用例には，新しいファイルの生成に対してトランザクションセマンティクスを提供するためのライブラリ（ＬＩＢＴＰ）についての記載はないし，また，引用例の「１．序論」の第５段落の記載によれば，引用例に開示されているＬＩＢＴＰは，４．４ＢＳＤ(Berkeley Software Distribution)データベースにおけるデータ動作に対してトランザクションセマンティクスを提供するものであり（これは，ＬＩＢＴＰについて説明する甲７の論文（１９９９年（平成１１年）６

月にモントレーで開催された「１９９９年ＵＳＥＮＩＸ年次技術会議」の予稿集に収載されたMargo Seltzerらによる「Berkeley DB」と題するもの。以下「甲７論文」という。）の記載（２頁３０行～３頁３行）からも明らかである。），ファイルシステム，すなわち，ファイル及びディレクトリ動作に対してトランザクションセマンティクスを提供するものではないから，引用例に「ファイルの更新処理方法」が記載されているとはいえない。

なお，被告は，「引用例の記載によれば，当該ライブラリ（ＬＩＢＴＰ）は，ＵＮＩＸプログラムにトランザクション保護を提供するために適切なコールを追加するものであるから，これを当該データベースアクセス法に限定する必要はない」と主張するが，引用例には，ライブラリ（ＬＩＢＴＰ）をデータベースアクセス法に限定する必要はないとの記載はないし，そのような事実を認めることもできない。

したがって，審決の上記認定が誤りであることは明らかである。

(3) 取消事由２（本願発明と引用発明の一致点の認定の誤り）

ア 審決は，本願発明と引用発明とを対比するに当たって，「引用発明における『新しいファイルの生成等のように複数のファイルの複数の部分が原子的に更新される必要がある場合や共有ファイルに対する同時的な更新が望まれる一方で論理的なデータの一貫性を維持する必要が生ずる場合』に実行されているファイル生成等の具体的なファイル操作は，トランザクションの保護の対象となるファイル操作を含む実行ルーチンであるので，これは，本願発明の『ネイティブファイルシステムにおけるファイル及びディレクトリの操作』を含むルーチンに対応するとともに『関連実行ルーチン』に相当し，」と認定したが，以下のとおり，この認定は誤りである。

(ｱ) 審決の上記認定中，「実行されているファイル生成等の具体的なファイル操作」に関し，審決が認定した第１の場合及び第２の場合は，引用例の「１．序論」に記載されているところ，同箇所には，トランザクションが必要な場合及び従来の解決方法が記載されているのみであるから，これだけでは，審決が認定した第

１の場合及び第２の場合における「実行されているファイル生成等の具体的なファイル操作」が何を意味するのか不明である。また，仮に，「実行されているファイル生成等の具体的なファイル操作」が引用例に記載されたＬＩＢＴＰを意味するのであるとしても，これは，データベースに対するデータ動作にトランザクションセマンティクスを提供するものであり，ファイル及びディレクトリ動作（ファイル生成等の具体的なファイル操作）にトランザクションセマンティクスを提供するものではない。したがって，引用例に，このような「ファイル生成等の具体的なファイル操作」が記載されているとはいえない。

(ｲ)　審決の上記認定中，「ファイル生成等の具体的なファイル操作は，トランザクションの保護の対象となるファイル操作を含む実行ルーチンであるので，」に関しても，上記(ｱ)のとおり，「ファイル生成等の具体的なファイル操作」が何を意味するのか不明であるし，仮に，引用例に記載されたＬＩＢＴＰを意味するのであるとしても，これは，ファイル操作にトランザクション保護を与えるものではない。

(ｳ)　審決の上記認定中，「ファイル生成等の具体的なファイル操作は，・・・本願発明の『ネイティブファイルシステムにおけるファイル及びディレクトリの操作』を含むルーチンに対応するとともに『関連実行ルーチン』に相当し，」に関しても，上記(ｱ)のとおり，「ファイル生成等の具体的なファイル操作」が何を意味するのか不明であるし，引用例に記載されたＬＩＢＴＰとも異なるものであるから，このようなファイル操作が，本願発明の「ネイティブファイルシステムにおけるファイル及びディレクトリの操作」に対応すると認定することはできない。

イ　審決は，本願発明と引用発明とを対比するに当たって，「引用発明における『ライブラリ（LIBTP）』は，ネイティブファイルシステムに必要に応じてトランザクションセマンティクスを追加あるいは提供するためのコンピュータプログラムルーチンの集合であるので，これは本願発明の『ライブラリ』に相当し，」と認定した。

しかしながら，引用発明のライブラリ（ＬＩＢＴＰ）は，引用例の「１．序論」の第５段落において説明されているところ，それによれば，データベース処理に対してトランザクションセマンティクスを提供するものであり，ネイティブファイル又はディレクトリ動作に対してトランザクションセマンティクスを提供するものではないから，同ライブラリ（ＬＩＢＴＰ）は，本願発明の「ライブラリ」に相当するものではない。

したがって，審決の上記認定は誤りである。

ウ　審決は，本願発明と引用発明とを対比するに当たって，「引用発明におけるライブラリを構成するルーチン（あるいはルーチンファミリ）である『ログマネージャ，バッファマネージャ，ロックマネージャ，トランザクションマネージャの各々を実現するルーチン』の各々は，本願発明の『ルーチン』あるいは『ルーチンファミリ』に相当し，又，引用発明においてもこれらのルーチンが複数あり，本願発明と同様に，『セット』を構成していると言うことができる。」と認定したが，以下のとおり，この認定は誤りである。

(ｱ)　引用例の「３．２．モジュールアーキテクチャ」には，ログマネージャ，バッファマネージャ，ロックマネージャ，トランザクションマネージャ等について記載されているところ，そのうち，「３．２．２．バッファマネージャ」の記載によれば，引用発明のバッファマネージャは，ディスクへのデータの書き込みを管理するものであり，データベース処理を管理するものであって，ファイル及びディレクトリ動作に関する処理を行うものではない。したがって，引用発明のバッファマネージャは，本願発明の「ルーチン」あるいは「ルーチンファミリ」に相当するものではない。

(ｲ)　引用例の「３．２．３．ロックマネージャ」の記載によれば，引用発明のロックマネージャは，データの書き込み及び読み出しの競合を管理するものであり，データベース処理を管理するものであって，ファイル及びディレクトリ動作に関する処理を行うものではない。したがって，引用発明のロックマネージャは，本願発

明の「ルーチン」あるいは「ルーチンファミリ」に相当するものではない。

(ｳ)　引用例の「３．２．５．トランザクションマネージャ」の記載によれば，引用発明のトランザクションマネージャは，データベースにおけるデータの更新を管理するものであり，ファイル及びディレクトリ動作に関する処理を行うものではない。したがって，引用発明のトランザクションマネージャは，本願発明の「ルーチン」あるいは「ルーチンファミリ」に相当するものではない。

エ　審決は，本願発明と引用発明とを対比するに当たって，「引用発明の『コミット処理及びアボート処理の際参照される事前及び事後のログを取得する処理』は，トランザクション処理におけるログ取得の処理であるので，本願発明の『ファイル又はディレクトリ動作をロールバックするために必要な情報を記憶するコンピュータ命令を含む実行ルーチン』に相当し，」と認定した。

しかしながら，上記認定における引用発明の「コミット処理及びアボート処理の際参照される事前及び事後のログを取得する処理」は，引用例の「３．２．２．バッファマネージャ」に記載された内容を指すものであり，データベース処理に関するものであって，ファイル又はディレクトリ動作に関するものではないから，本願発明の「ファイル又はディレクトリ動作をロールバックするために必要な情報を記憶するコンピュータ命令を含む実行ルーチン」に相当するものではない。

したがって，審決の上記認定は誤りである。

オ　審決は，本願発明と引用発明とを対比するに当たり，「引用発明の『txn_commit プリミティブによって起動されるコミット処理』は，トランザクション処理におけるコミット処理であるので，本願発明の『関連実行ルーチンの結果をコミットさせるコンピュータ命令を含む終了ルーチン』に相当し，」と認定した。

しかしながら，上記認定における引用発明の「txn_commit プリミティブによって起動されるコミット処理」は，引用例の「３．２．５．トランザクションマネージャ」に記載された内容を指すものであり，データベースシステムにおけるデータ動作に関するものであって，ファイル又はディレクトリ動作に関するものではない

から，本願発明の「関連実行ルーチンの結果をコミットさせるコンピュータ命令を含む終了ルーチン」に相当するものではない。

したがって，審決の上記認定は誤りである。

カ　審決は，本願発明と引用発明とを対比するに当たり，「引用発明の『txn_abort プリミティブによって起動されるアボート処理』は，トランザクション処理におけるロールバック処理であるので，本願発明の『関連実行ルーチンの結果をロールバックさせるコンピュータ命令を含むアンドゥルーチン』に相当している。」と認定した。

しかしながら，上記認定における引用発明の「txn_abort プリミティブによって起動されるアボート処理」は，引用例の「３．２．５．トランザクションマネージャ」に記載された内容を指すものであるところ，同処理は，データベースにされた変更についてｕｎｄｏする処理であり，データ動作に関するものであって，ファイル又はディレクトリ動作に関するものではないから，本願発明の「関連実行ルーチンの結果をロールバックさせるコンピュータ命令を含むアンドゥルーチン」に相当するものではない。

したがって，審決の上記認定は誤りである。

キ　審決は，本願発明と引用発明とが，「ネイティブファイルシステムのネイティブファイルおよびディレクトリ動作のセットへトランザクションセマンティクスを追加するためのコンピュータプログラムライブラリであって，前記ライブラリが１つ以上のルーチンファミリのセットを有し，このようなルーチンファミリの各々が，(a)このようなネイティブファイルまたはディレクトリ動作をロールバックするために必要な情報を記憶するコンピュータ命令を含む実行ルーチンと，(b)コンピュータに，関連実行ルーチンの結果をコミットさせるコンピュータ命令を含む終了ルーチンと，(c)コンピュータに，関連実行ルーチンの結果をロールバックさせるコンピュータ命令を含むアンドゥルーチンと，を有することを特徴とするコンピュータプログラムライブラリ」である点で一致すると認定したが，以下のとおり，

この認定は誤りである。

(ｱ)　審決が認定した引用発明は，引用例の「１．序論」に記載された内容とその後の部分に記載されたＬＩＢＴＰを組み合わせたものであるところ，ＬＩＢＴＰは，データベース処理に関するものであるから，同発明における処理は，データベース処理であるデータ動作に関するものである。

なお，引用例には，ライブラリ（ＬＩＢＴＰ）が，本願発明（ネイティブファイルまたはディレクトリ動作をロールバックするために必要な情報を記憶するコンピュータ命令を含む。）の実行ルーチン，終了ルーチン及びアンドゥルーチンを含むとの記載はないし，そのような事実を認めることもできない。

(ｲ)　したがって，引用発明は，「ネイティブファイルシステムのネイティブファイルおよびディレクトリ動作のセットへトランザクションセマンティクスを追加するためのコンピュータプログラムライブラリ」ではないし，「(a)このようなネイティブファイルまたはディレクトリ動作をロールバックするために必要な情報を記憶するコンピュータ命令を含む実行ルーチン」を含むものでもないから，審決の上記一致点の認定は誤りである。

(4)　取消事由３（本願発明と引用発明の相違点の看過）

本願発明と引用発明は，「本願発明は，『ネイティブファイルシステムのネイティブファイルおよびディレクトリ動作のセットへトランザクションセマンティクスを追加するもの』であるのに対し，引用発明は，『データの読み出し，書き込み等のデータ動作に対してトランザクションセマンティクスを追加するもの』である点」（以下「原告主張相違点」という。）において相違するのであり，審決は，かかる相違点の存在を看過したものである。

なお，被告は，「引用例においては，『１．序論』に記載された『従来のＵＮＩＸファイルシステム』にトランザクションセマンティクスを提供するための具体的な実装の例として，『３．アーキテクチャ』，『４．実装』等に記載されたライブラリ（ＬＩＢＴＰ）についての説明がされている」と主張するが，「１．序論」に

は，審決が認定した引用発明の必須の構成が記載されているわけではなく，また，「１．序論」に記載された内容と「３．アーキテクチャ」，「４．実装」等に記載された内容とが，技術的に関連して所定の機能を提供するものでもないから，被告の上記主張は失当である。

２　被告の反論の要点

(1)　原告の主張(1)（前提となる技術内容等について）に対し

ア　原告は，コンピュータシステムがデータベースシステムとファイルシステムに分類され，引用発明は前者に関するものであり，本願発明は後者に関するものである旨主張する。

(ｱ)　しかしながら，両者は別のものではなく，ともに，コンピュータが大量のデータを記憶装置に格納・保管し，必要に応じてこれを取り出したり，更新・削除を行ったりするシステムである。両者は，データを記憶装置に記憶する処理をハードウェア的側面に重点を置いて見たときには，「ファイルシステム」と捉えられ，記憶装置に記憶されるデータに対する処理というソフトウェア的側面に重点を置いて見たときには，「データベースシステム」と捉えられる。

(ｲ)　そして，平成５年４月２５日発行の坂下善彦ら著「分散システム入門（初版）」（乙１。以下「乙１文献」という。）に記載されている（１５７頁８行～１５８頁９行）とおり，データベースシステムといえども，データは，最終的には，記憶装置（主にハードディスク）に記憶（二次記憶）される（データベースシステムによって処理されたデータは，ファイルシステムに引き渡され，ファイルシステムの機能によって記憶装置（ハードディスク）に記憶される）ことは周知の事項である。すなわち，データベースシステムは，通常，ファイルシステムを利用して構築されているものであり，データベースのデータも，ハードディスクに保管される際には，ファイルシステムの操作により，ファイルとして保管されるのであるから，原告の上記主張は失当である。

イ　原告は，本願発明がファイルシステムにトランザクションセマンティクスを付与するものであるのに対し，引用例にはその点についての記載がない旨主張する。

(ｱ)　しかしながら，引用例の「１．序論」の記載（１頁２９行～３頁５行）によれば，引用例は，汎用的なプログラムが用いるファイルシステムにトランザクションセマンティクスを提供するライブラリについて論じているものであり，「３．アーキテクチャ」及び「４．実装」で説明されているライブラリ（ＬＩＢＴＰ）がそれに当たる。

(ｲ)　そして，上記アのとおり，データベースシステムは，ファイルシステムを利用して構築されるシステムであり，従来は，データベースシステムがファイルシステムを利用するため，ファイルシステムに属するルーチン（open，read，write等）が用いられていたところ，引用例の「４．実装」の記載（１６頁２２行～１７頁１行）によれば，これらのルーチンを，トランザクションセマンティクスを付与するための新たなルーチン（buf_open，buf_get，buf_unpin等）に置き換えればよく（すなわち，buf_open，buf_get，buf_unpin等は，既存のファイルシステムに属するルーチンであるopen，read，write等の機能を含んだ上でトランザクション保護を追加するための機能を持ったルーチンであるとともに，引用例に記載されたライブラリ（ＬＩＢＴＰ）の一部である。），これらの置き換えにより，データベースシステムとしても利用されるファイルシステムに，トランザクションセマンティクスが付与されるものである。

なお，このためには，４．４ＢＳＤデータベースアクセスルーチンの対応するルーチンの呼出しを置き換えるとともに，ファイルシステム側に，置き換えられたルーチンに対応するルーチンを加える必要があるが，当該置き換えられるルーチンが４．４ＢＳＤデータベースアクセスルーチンに特化したものでないことは，引用例の「１．序論」の記載からも明らかである。

(ｳ)　また，平成５年１月２０日発行の塩谷修ら著「実用ＵＮＩＸシステムプログラミング（第２版）」（乙２。以下「乙２文献」という。）に記載されている

（１５６頁２１行～１６０頁３６行）とおり，既存のファイルシステムに属するopen システムコールは，モード指定する際，Ｏ＿＿ＣＲＥＡＴを指定すると，fileで指定した path 名のファイルを作成する動作を行うことが周知であり，これは，本願明細書にいう「ファイル動作」を行うことと同じであるから，引用例に記載されたルーチン open も，本願明細書にいう「ファイル動作」を行っていることになる。

そうすると，既存のファイルシステムに属するルーチン open の機能を含んだ上でトランザクション保護を追加するための機能を持ったルーチン buf_open は，本願明細書にいう「ファイル動作」に対しトランザクションセマンティクスを追加するルーチンであるということができ，したがって，buf_open を含むライブラリであるＬＩＢＴＰが，本願明細書にいう「ファイル動作」のセットに対しトランザクションセマンティクスを追加する構成であることは明らかである。

そして，引用例に記載されたルーチン（「コミット処理及びアボート処理の際参照される事前及び事後のログを取得する処理」等）も，ネイティブファイルシステムのネイティブファイル及びディレクトリ動作のセットに対しトランザクションセマンティクスを追加するルーチン（buf_open 等）の「ログを取得する処理」等を行うものであるから，本願発明の「ネイティブファイルシステムのネイティブファイルおよびディレクトリ動作のセットへトランザクションセマンティクスを追加する」との構成と，引用例に記載されたライブラリ（ＬＩＢＴＰ）とが，技術的に異なるものであるということはできない。

(ｴ)　このように，引用発明は，ファイルシステムにトランザクションセマンティクスを付与するものであるから，原告の上記主張は根拠がなく失当である。

(2)　取消事由１（引用発明の認定の誤り）に対し

ア　原告は，引用例の「１．序論」に記載された内容と，「３．アーキテクチャ」，「４．実装」等に記載された内容とを結合した１つの発明を引用発明とした審決の認定に誤りがある旨主張する。

しかしながら，引用例の「３．アーキテクチャ」，「４．実装」等の記載は，「１．序論」の記載を受け，「従来のＵＮＩＸファイルシステム」にトランザクションセマンティクスを付与する技術であるライブラリ（ＬＩＢＴＰ）について論じたものであるから，引用発明は，従来のＵＮＩＸのファイルシステムにトランザクションセマンティクスを付与するものである。

また，一般に，「序論」は，本論の前置きとして，本論の手がかりとなる事項を記述したものであるところ，引用例においても，本論に相当する「３．アーキテクチャ」，「４．実装」等は，「１．序論」を受けて論じられている。

したがって，原告の上記主張は失当である。

イ　原告は，引用例の「１．序論」に記載された内容と，「３．アーキテクチャ」，「４．実装」等に記載されたライブラリ（ＬＩＢＴＰ）とは，技術的な整合性を有しないから，両者を結合しても，１つの発明を構成することはできない旨主張する。

しかしながら，原告が主張する審決が認定した第１の場合及び第２の場合は，いずれも，引用例に記載された従来技術に関するものであるところ，上記アのとおり，引用例の「３．アーキテクチャ」，「４．実装」等の記載は，「１．序論」の記載を受け，「従来のＵＮＩＸファイルシステム」にトランザクションセマンテイクスを付与する技術について論じたものであるから，「１．序論」に記載された内容と，「３．アーキテクチャ」，「４．実装」等に記載されたライブラリ（ＬＩＢＴＰ）は，１つの発明についてのものであり，したがって，原告の上記主張は根拠がなく失当である。

ウ　原告は，引用例に開示されているＬＩＢＴＰは，４．４ＢＳＤデータベースにおけるデータ動作に対してトランザクションセマンティクスを提供するものであり，ファイル及びディレクトリ動作に対してトランザクションセマンティクスを提供するものではないから，引用例に「ファイルの更新処理方法」が記載されているとはいえない旨主張する。

(ｱ)　しかしながら，引用例の記載（３頁４～８行）によれば，４．４ＢＳＤデータベースアクセス法は，「３．アーキテクチャ」，「４．実装」等に記載された「従来のＵＮＩＸファイルシステム」にトランザクションセマンティクスを付与するライブラリ（ＬＩＢＴＰ）を使用するように修正された上，トランザクションセマンティクスを付与されている。このことからすると，４．４ＢＳＤデータベースアクセス法とライブラリ（ＬＩＢＴＰ）とは，別のものであるといえる。

そして，引用例の記載（３頁８～１１行）によれば，当該ライブラリ（ＬＩＢＴＰ）は，ＵＮＩＸプログラムにトランザクション保護を提供するために適切なコールを追加するものであるから，これを当該データベースアクセス法に限定する必要はない。

(ｲ)　また，上記(1)のとおり，データベースシステムは，ファイルシステムを利用したものであり，他のＵＮＩＸプログラムも，二次記憶を利用する際にはファイルシステムを利用するものであることからすると，引用例に記載されたライブラリ（ＬＩＢＴＰ）は，ファイルシステムについてのライブラリであるといえる。

(ｳ)　なお，原告は，引用例に開示されているＬＩＢＴＰがデータ動作に対してトランザクションセマンティクスを提供するものであることは，甲７論文の記載からも明らかである旨主張するが，甲７論文には，ＬＩＢＴＰがデータベースシステムそのものであるとの記載はないし，上記(ｱ)のとおり，４．４ＢＳＤ（バークレイＤＢ）とＬＩＢＴＰは別のものであり，他のＵＮＩＸプログラムにおいても，ＬＩＢＴＰを利用することが可能であるから，原告の上記主張は根拠がなく失当である。

エ　以上のとおりであるから，審決の引用発明の認定には，何らの誤りもない。

(3)　取消事由２（本願発明と引用発明の一致点の認定の誤り）に対し

ア　原告は，審決がした本願発明と引用発明との対比に関し，引用発明における「実行されているファイル生成等の具体的なファイル操作」が何を意味するのか不明であるし，仮にＬＩＢＴＰを意味するとしても，これはファイル生成等の具体的

なファイル操作にトランザクションセマンティクスを提供するものではないから，審決の「引用発明における・・・ファイル生成等の具体的なファイル操作は，トランザクションの保護の対象となるファイル操作を含む実行ルーチンであるので，これは，本願発明の『ネイティブファイルシステムにおけるファイル及びディレクトリの操作』を含むルーチンに対応するとともに『関連実行ルーチン』に相当」するとの認定は誤りであると主張する。

しかしながら，引用例の「１．序論」に「新たなファイルが作成される場合」と例示されているように，引用例に従来技術として示されているファイル操作は，ファイル生成等の具体的なファイル操作である。

また，引用例の「１．序論」には，従来技術として，審決が認定した第１の場合及び第２の場合における「伝統的なＵＮＩＸファイルシステム」での解決方法が記載されているところ，これを，トランザクションセマンティクスを提供することにより解決する方法が，「３．アーキテクチャ」，「４．実装」等に記載されたライブラリ（ＬＩＢＴＰ）であるから，「引用発明における・・・ファイル生成等の具体的なファイル操作は，トランザクションの保護の対象となるファイル操作を含む実行ルーチンである」こと，これが「本願発明の『ネイティブファイルシステムにおけるファイル及びディレクトリの操作』を含むルーチンに対応するとともに『関連実行ルーチン』に相当」することは明らかである。

したがって，原告の上記主張は失当である。

イ　原告は，審決がした本願発明と引用発明との対比に関し，引用発明のライブラリ（ＬＩＢＴＰ）は，データベース処理に対してトランザクションセマンティクスを提供するものであり，ネイティブファイル又はディレクトリ動作に対してトランザクションセマンティクスを提供するものではないから，審決の「引用発明における『ライブラリ（LIBTP）』は，ネイティブファイルシステムに必要に応じてトランザクションセマンティクスを追加あるいは提供するためのコンピュータプログラムルーチンの集合であるので，これは本願発明の『ライブラリ』に相当」すると

の認定は誤りであると主張する。

しかしながら，上記(2)ウのとおり，引用例に記載されたライブラリ（ＬＩＢＴＰ）は，ファイルシステムについてのライブラリであるから，本願発明のライブラリに相当する。したがって，原告の上記主張は根拠がなく失当である。

ウ　原告は，審決がした本願発明と引用発明との対比に関し，引用発明のバッファマネージャ，ロックマネージャ及びトランザクションマネージャは，ファイル及びディレクトリ処理動作に関する処理を行うものではないから，審決の「引用発明における・・・『・・・バッファマネージャ，ロックマネージャ，トランザクションマネージャの各々を実現するルーチン』の各々は，本願発明の『ルーチン』あるいは『ルーチンファミリ』に相当し，又，引用発明においてもこれらのルーチンが複数あり，本願発明と同様に，『セット』を構成していると言うことができる。」との認定は誤りであると主張する。

しかしながら，上記(1)のとおり，データベースは，ディスクに対応する二次記憶へのデータの書き込みにファイルシステムを利用しており，ディスクへのデータの書き込みを管理するものも，ファイルシステムに属するのであるから，引用発明のバッファマネージャ，ロックマネージャ及びトランザクションマネージャは，いずれも，ファイルシステムのルーチンであるといえる。

そうすると，審決の上記認定には何らの誤りもなく，したがって，原告の上記主張は失当である。

エ　原告は，審決がした本願発明と引用発明との対比に関し，引用発明の「コミット処理及びアボート処理の際参照される事前及び事後のログを取得する処理」は引用例の「３．２．２．バッファマネージャ」に，「txn_commit プリミティブによって起動されるコミット処理」及び「txn_abort プリミティブによって起動されるアボート処理」はいずれも引用例の「３．２．５．トランザクションマネージャ」に，それぞれ記載された内容を指すものであり，データベース処理に関するものであって，ファイル又はディレクトリ動作に関するものではないから，審決がし

た「引用発明の『コミット処理及びアボート処理の際参照される事前及び事後のログを取得する処理』は，トランザクション処理におけるログ取得の処理であるので，本願発明の『ファイル又はディレクトリ動作をロールバックするために必要な情報を記憶するコンピュータ命令を含む実行ルーチン』に相当し」との認定，「引用発明の『txn_commit プリミティブによって起動されるコミット処理』は，トランザクション処理におけるコミット処理であるので，本願発明の『関連実行ルーチンの結果をコミットさせるコンピュータ命令を含む終了ルーチン』に相当し」との認定，及び「引用発明の『txn_abort プリミティブによって起動されるアボート処理』は，トランザクション処理におけるロールバック処理であるので，本願発明の『関連実行ルーチンの結果をロールバックさせるコンピュータ命令を含むアンドゥルーチン』に相当している」との認定は，いずれも誤りであると主張する。（取消事由２のエないしカ）

しかしながら，上記(1)のとおり，データベースシステムは，ファイルシステムを利用して構築されるシステムであること，上記(2)アのとおり，引用発明は，「従来のＵＮＩＸファイルシステム」にトランザクションセマンティクスを付与するものであること，引用例においては，「１．序論」に記載された「従来のＵＮＩＸファイルシステム」にトランザクションセマンティクスを提供するための具体的な実装の例として，「３．アーキテクチャ」，「４．実装」等に記載されたライブラリ（ＬＩＢＴＰ）についての説明がされていることに照らすと，引用発明の「コミット処理及びアボート処理の際参照される事前及び事後のログを取得する処理」，「txn_commit プリミティブによって起動されるコミット処理」及び「txn_abort プリミティブによって起動されるアボート処理」は，それぞれ，本願発明の「ファイル又はディレクトリ動作をロールバックするために必要な情報を記憶するコンピュータ命令を含む実行ルーチン」，「関連実行ルーチンの結果をコミットさせるコンピュータ命令を含む終了ルーチン」及び「関連実行ルーチンの結果をコミットさせるコンピュータ命令を含むアンドゥルーチン」に相当するといえるから，原告の上

記各主張はいずれも理由がなく，審決の上記各認定に誤りはない。

オ　原告は，引用発明のＬＩＢＴＰは，データベース処理に関するものであって，同発明における処理は，データベース処理であるデータ動作に関するものであるから，引用発明は，「ネイティブファイルシステムのネイティブファイルおよびディレクトリ動作のセットへトランザクションセマンティクスを追加するためのコンピュータプログラムライブラリ」ではないし，「(a)このようなネイティブファイルまたはディレクトリ動作をロールバックするために必要な情報を記憶するコンピュータ命令を含む実行ルーチン」を含むものではないとして，本願発明と引用発明とが，「ネイティブファイルシステムのネイティブファイルおよびディレクトリ動作のセットへトランザクションセマンティクスを追加するためのコンピュータプログラムライブラリであって，前記ライブラリが１つ以上のルーチンファミリのセットを有し，このようなルーチンファミリの各々が，(a)このようなネイティブファイルまたはディレクトリ動作をロールバックするために必要な情報を記憶するコンピュータ命令を含む実行ルーチンと，・・・を有することを特徴とするコンピュータプログラムライブラリ」である点で一致するとした審決の認定が誤りであると主張する。

しかしながら，上記エのとおり，データベースシステムは，ファイルシステムを利用して構築されるシステムであること，引用発明は，「従来のＵＮＩＸファイルシステム」にトランザクションセマンティクスを付与するものであること，引用例においては，「１．序論」に記載された「従来のＵＮＩＸファイルシステム」にトランザクションセマンティクスを提供するための具体的な実装の例として，「３．アーキテクチャ」，「４．実装」等に記載されたライブラリ（ＬＩＢＴＰ）についての説明がされていることにかんがみれば，引用発明の「ライブラリ（ＬＩＢＴＰ）」は，「ネイティブファイルシステムのネイティブファイルおよびディレクトリ動作のセットへトランザクションセマンテイクスを追加するためのコンピュータプログラムライブラリ」であるとともに，「このようなネイティブファイルまたは

ディレクトリ動作をロールバックするために必要な情報を記憶するコンピュータ命令を含む実行ルーチン」を含んでいるといえるから，審決のした一致点の認定に誤りはなく，原告の上記主張は失当である。

カ　以上のとおり，審決がした本願発明と引用発明との対比・一致点の認定に誤りはない。

(4)　取消事由３（本願発明と引用発明の相違点の看過）に対し

原告は，本願発明と引用発明は，原告主張相違点において相違するものであり，審決は，かかる相違点の存在を看過したものであると主張する。

しかしながら，上記(3)エのとおり，引用例においては，「１．序論」に記載された「従来のＵＮＩＸファイルシステム」にトランザクションセマンティクスを提供するための具体的な実装の例として，「３．アーキテクチャ」，「４．実装」等に記載されたライブラリ（ＬＩＢＴＰ）についての説明がされていることに照らすと，引用発明は，「ネイティブファイルシステムのネイティブファイルおよびディレクトリ動作のセットへトランザクションセマンティクスを追加するもの」であるということができるから，原告主張相違点は存在せず，審決に相違点看過の誤りはない。

第４　当裁判所の判断

１　前提となる技術内容（引用例に記載された技術が本願明細書にいう「ファイルおよびディレクトリ動作」に対しトランザクションセマンティクスを付与するものといえるか）について

(1)　引用例には，次の各記載がある。

ア「概要

・・・。従来のＵＮＩＸシステムにおいては，トランザクションを使用するための唯一容易な方法は，データベースシステムを購入することである。このようなシステムは処理速度が遅くコストがかかることが多く，所望されている通りの機能性を提供しないこともある。本論文

は，４．４ＢＳＤ（Berkeley Software Distribution）データベースアクセスルーチン（ｄｂ（３））を使用する簡易型非プロプライエタリ・トランザクションライブラリーであるＬＩＢＴＰの設計，実装および性能について提示している。」（１頁５～１３行）

イ「１．序論

トランザクションはデータベースシステムで使用されて，同時アクセスするユーザーがデータベースのインテグリティを破壊することなくマルチオペレーション更新を適用できるようにする。これは，原子性，一貫性，分離性および耐久性という特性を提供する。原子性とは，トランザクションを構成する更新一式が単一のユニットとして適用されるべきである，つまりすべてがデータベースに適用されるか，すべてが欠けているかのいずれかでなければならないことを意味する。一貫性とは，トランザクションがデータベースをある論理的一貫性の状態から別の論理的一貫性の状態に移すことを要求するものである。分離特性は，同時トランザクションが，トランザクションを順次実行することによって得られる結果と区別できない結果を生成することを要求するものである。最後に，耐久性とは，トランザクションがコミットされると，その結果がシステム障害を越えて保存されることを要求するものである・・・。

これらの特性についてはデータベースと関連してしばしば議論されるところであるが，これらは，より汎用なアプリケーションにとって有用なプログラミングパラダイムである。現在のアドホック機構に取って代り，トランザクションを使用できる複数の様々な状況がある。

そうした状況の１つは，複数のファイルまたは複数のファイルの一部が原子的に更新される必要がある場合である。例えば，従来のＵＮＩＸファイルシステムは順序制約を使用して，クラッシュに直面した場合の回復可能性を達成する。新たなファイルが作成される場合，その新たなファイルがディレクトリ構造に追加される前にそれのｉノードがディスクに書き込まれる。」（１頁１７行～２頁８行）

ウ「現行のアドホック機構に取って代り，トランザクションが使用できる第２の状況は，共有ファイルの同時更新が所望されるが，データの論理的一貫性を維持する必要があるというアプリケーションにおいてである。例えば，パスワードファイルが更新される場合，ファイルロックを使用して同時アクセスを許可しない。パスワードファイルに対するトランザクションセ

マンティクスでは，パスワードデータベースの論理的一貫性を維持しつつ同時更新を可能にする。」（２頁２０～２５行）

エ「本論文において，トランザクションセマンティクス（原子性，一貫性，分離性および耐久性）を提供する簡易型ライブラリーを提示する。４．４ＢＳＤデータベースアクセス法は，このライブラリーを使用するために修正されており，任意に，アプリケーション間の共有バッファ管理，ロッキングおよびトランザクションセマンティクスを提供する。ｄｂ（３）ライブラリーによるトランザクション保護をリクエストすることによって，・・・ＵＮＩＸプログラムはそのデータをトランザクション保護することができる。」（３頁４～１１行）

オ「３．アーキテクチャ

ライブラリーは，トランザクション処理に必要なサービスに洗練されたインタフェースを提供するように設計されている。これらのサービスは回復，同時実行制御および共有データ管理である。まず，回復，同時実行制御およびバッファ管理実装における設計トレードオフについて論じてから，ライブラリーアーキテクチャおよびモジュール全体について説明することにする。」（４頁２８行～５頁３行）

カ「４．実装」（１３頁８行）

キ「４．４トランザクション保護アクセス法

・・・

アクセス法にトランザクション保護を追加するのに必要な修正は極めて単純であり局所的である。

１．ファイルｏｐｅｎをｂｕｆ＿ｏｐｅｎで置換する。

２．ファイルｒｅａｄおよびｗｒｉｔｅのコールをバッファマネージャコール（ｂｕｆ＿ｇｅｔ，ｂｕｆ＿ｕｎｐｉｎ）で置換する。

３．バッファマネージャコールの前に適切な（ｒｅａｄまたはｗｒｉｔｅ）ロックコールを置く。

４．更新前にロギング命令を発行する。

５．データにアクセスされた後に，バッファマネージャピンを解除する。

６．定義されたログレコードのタイプごとに取り消し／やり直しコードを提供する。」（１６頁８行～１７頁１行）

(2)　また，ＵＮＩＸシステムに関する乙２文献には，次の各記載がある。

ア「６．１．３　ファイルの作成とオープン

・・・

ここでは，creat，open システムコールを紹介します。

【creat】

・・・

ファイルを作り，書き込みの準備を行います。

すでに存在するファイルに上書きの準備を行います。

・・・

・存在するファイルに対して creat コールを行った場合，ファイルのサイズは０にリセットされます。モードとオーナは変わりません。

・ファイルが存在していない場合，ファイルのオーナおよびグループＩＤは・・・設定されたファイルが作られます。」（１５６頁２１行～１５７頁「説明」５行）

イ「【open】

・・・

ファイルをオープンします。」（１５９頁５～７行）

ウ「O_CREAT

ファイルが存在している場合，意味はありません。存在していない場合，creat コールと同じ機能が得られます。」（１６０頁１４～１６行）

(3)　上記(1)の各記載によれば，引用例は，既存のファイルシステムであるＵＮＩＸシステムのような汎用的なプログラムが用いるファイルシステムに対し，トランザクションセマンティクスを提供するライブラリについて論じているものであり，引用例の「３．アーキテクチャ」及び「４．実装」において，そのライブラリにつき，具体的に説明しているものと認められる。

また，上記(1)キのとおり，引用例に記載されたライブラリ（ＬＩＢＴＰ）においては，ファイル処理にトランザクション保護機能を追加するために，プログラム中の open，read 及び write を，それぞれ buf_open，buf_get 及び buf_unpin に置き換えることが行われているのであるから，buf_open，buf_get 及び buf_unpin は，それぞれ既存のファイルシステムに属するルーチンである open，read 及び write の機能を含んだ上で，トランザクション保護機能を追加するための機能を有するルーチンであると理解することができる。

さらに，上記(2)の各記載によれば，既存のファイルシステムであるＵＮＩＸシステムに属するシステムコール open は，「ファイルを作り，書き込みの準備を行う」システムコールと同じ機能を有していることが認められる（なお，乙２文献が，平成５年１月２０日発行のＵＮＩＸシステムプログラミングに関する一般的な概説書であることにかんがみれば，このことは，本件特許出願当時は，当業者に周知の事項であったものと認められる。）。

加えて，そもそも，ファイルシステムは，オペレーティングシステムが有する機能の１つであって，アプリケーションプログラムに応答し，外部の記憶媒体（ハードディスク等）に対してデータの書き込み，変更（更新，消去），読み出し等の動作を行うものであるところ，新しいファイルの生成，ファイル名の変更，ファイルの削除等の処理も，オペレーティングシステムからみれば，データの処理に他ならないことは，経験則上明らかである（例えば，ファイルの生成は，新しいファイルについてのデータを生成することであり，ファイル名の変更は，ファイル名のデータを更新することであり，ファイルの削除は，ファイルについてのデータを削除することである。）。

以上からすると，引用例には，本願明細書にいう「ファイルおよびディレクトリ動作」に対しトランザクションセマンティクスを付与する技術についての記載があると認めるのが相当であり，引用例（特に，「３．アーキテクチャ」及び「４．実装」の各欄）に当該記載がない旨をいう原告の主張は，これを採用することができ

ない。

(4)　そこで，以上を前提に，以下，各取消事由について検討する（したがって，以下，引用例（「３．アーキテクチャ」及び「４．実装」の各欄を含む。）に記載された技術が，本願明細書にいう「ファイルおよびディレクトリ動作」に対しトランザクションセマンティクスを付与するものではないことを前提とする原告の主張は，いずれも，その前提を欠くものとして失当である。）。

２　取消事由１（引用発明の認定の誤り）について

(1)　上記１のとおり，引用例（「３．アーキテクチャ」及び「４．実装」の各欄を含む。）には，本願明細書にいう「ファイルおよびディレクトリ動作」に対しトランザクションセマンティクスを付与する技術についての記載があると認められるから，審決が引用する引用例の記載事項（前記第２の３(1)）によれば，引用発明は，審決が認定したとおりのもの（同）であると認めるのが相当である。

(2)ア　原告は，引用例の「１．序論」に記載された内容と，「３．アーキテクチャ」，「４．実装」等に記載された内容とを結合した１つの発明は，引用例のどこにも記載されていないと主張する。

しかしながら，原告の上記主張は，その前後の主張の内容から明らかなとおり，「引用例の『３．アーキテクチャ』，『４．実装』等には，引用例の主要なテーマである，データベースシステムに対するデータ動作にトランザクションセマンティクスを付与する技術であるライブラリ（ＬＩＢＴＰ）が記載されているにとどま（る）（したがって，ファイル及びディレクトリ動作にトランザクションセマンティクスを付与するものではない。）」ことを前提とするものである。

そうすると，上記１のとおり，原告の上記主張は，その前提を欠くものとして失当である。

イ　原告は，審決が認定した第１の場合及び第２の場合が記載された引用例の「１．序論」には，いずれも，トランザクションセマンティクスを提供するための

ライブラリ（ＬＩＢＴＰ）の各ルーチンを実行するとは記載されていないから，引用例の「１．序論」に記載された内容と，「３．アーキテクチャ」，「４．実装」等に記載された内容とは，技術的な整合性を有しておらず，両者を結合しても，１つの発明を構成することができないと主張する。

しかしながら，審決（前記第２の３(1)）が引用するとおり，引用例の「３．アーキテクチャ」及び「４．実装」の欄には，トランザクションセマンティクスを提供するためのライブラリ（ＬＩＢＴＰ）の各ルーチンを実行することについての記載があるのであるから，原告の上記主張は，要するに，引用例の「３．アーキテクチャ」及び「４．実装」に記載された技術が，本願明細書にいう「ファイルおよびディレクトリ動作」に対しトランザクションセマンティクスを付与するものではないことを前提とするものと理解せざるを得ない。

そうすると，上記１のとおり，原告の上記主張は，その前提を欠くものとして失当である。

(3)　以上のとおり，審決の引用発明の認定に原告主張の誤りはないから，取消事由１は，理由がない。

３　取消事由２（本願発明と引用発明の一致点の認定の誤り）について

(1)　前記１のとおり，引用例（「３．アーキテクチャ」及び「４．実装」の各欄を含む。）には，本願明細書にいう「ファイルおよびディレクトリ動作」に対しトランザクションセマンティクスを付与する技術についての記載があると認められるから，本願発明と引用発明とは，審決が両発明を対比した上で認定したとおりの点（前記第２の３(2)）で一致すると認めるのが相当である。

(2)　原告は，審決が，本願発明と引用発明との対比において，「引用発明における『新しいファイルの生成等のように複数のファイルの複数の部分が原子的に更新される必要がある場合や共有ファイルに対する同時的な更新が望まれる一方で論理的なデータの一貫性を維持する必要が生ずる場合』に実行されているファイル生

成等の具体的なファイル操作は，トランザクションの保護の対象となるファイル操作を含む実行ルーチンであるので，これは，本願発明の『ネイティブファイルシステムにおけるファイル及びディレクトリの操作』を含むルーチンに対応するとともに『関連実行ルーチン』に相当する」と認定したことに関し，当該「実行されているファイル生成等の具体的なファイル操作」が何を指すのか不明であるなどと主張する。

しかしながら，原告の上記主張は，その前後の主張の内容から明らかなとおり，「審決が認定した第１の場合及び第２の場合は，引用例の『１．序論』に記載されているところ，同箇所には，トランザクションが必要な場合及び従来の解決方法が記載されているのみである」ことを前提とするものである。

そして，前記１において説示したところに照らし，審決（前記第２の３(1)）が摘記した引用例の「３．アーキテクチャ」及び「４．実装」の各欄には，審決の上記認定における「実行されているファイル生成等の具体的なファイル操作」についての具体的記載があることは明らかであるから，原告の上記主張は，要するに，引用例の「３．アーキテクチャ」及び「４．実装」に記載された技術が，本願明細書にいう「ファイルおよびディレクトリ動作」に対しトランザクションセマンティクスを付与するものではないことを前提とするものと理解せざるを得ない。

そうすると，前記１のとおり，原告の上記主張は，その前提を欠くものとして失当である。

(3)　以上のとおり，本願発明と引用発明の一致点についての審決の認定に原告主張の誤りはないから，取消事由２は，理由がない。

４　取消事由３（本願発明と引用発明の相違点の看過）について

前記１において説示したところに照らせば，本願発明は，原告主張相違点に係る引用発明の構成を備えるものであり，そうすると原告主張相違点は存在しないから，審決に本願発明と引用発明の相違点を看過した誤りはなく，取消事由３は，理由が

ない。

5　結論

よって，原告の主張する審決取消事由はいずれも理由がないから，原告の請求は棄却されるべきである。

知的財産高等裁判所第４部

裁判長裁判官　石　原　直　樹

裁判官　榎　戸　道　也

裁判官　浅　井　憲