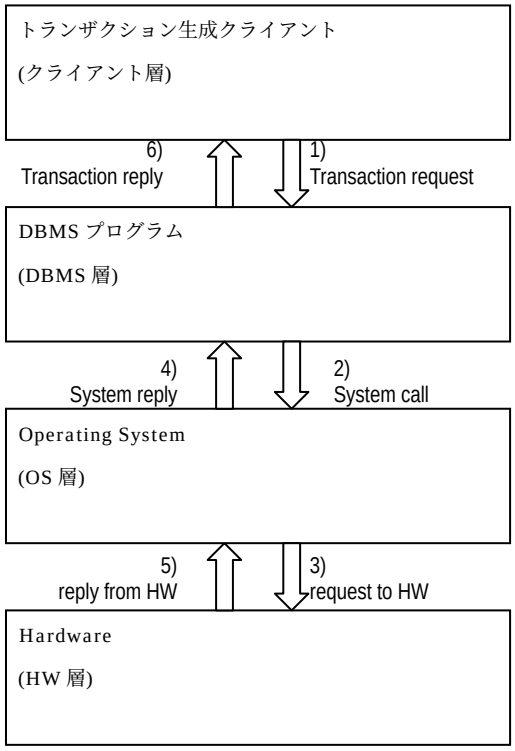
A DBMS の動作とマイクロベンチマーク

A.1 DBMS の性質とベンチマーク

広義に考えた場合のデータベースマネジメントシステム(Database Management System/DBMS)は図 A.1-1 のような階層構造と処理の流れになる。



1) クライアント層は、生成した Transaction request を DBMS 層に送る。
2) DBMS 層では各 Transaction request を処理するために、OS 層に対しいくつかの system call を発行する。
3) OS 層は、各 system call を処理するために、いくつかの request to HW を発行する。
4) HW 層は request to HW に従って処理を行う。この結果は reply from HW として OS 層へ送られる。
5) OS 層は reply from HW を受け取って各 request to HW が終了した事を確認しながら、system call などの処理を進める。OS 層が system call 処理を完了すると、system reply を DBMS 層に送る。
6) DBMS 層は system reply を受け取って、system call の完了を確認しながら、Transaction request 処理を進める。最終的に Transaction request 処理が終了すると Transaction reply をクライアント層に送る。

図 A.1-1 広義の DBMS 階層構造と処理の流れ

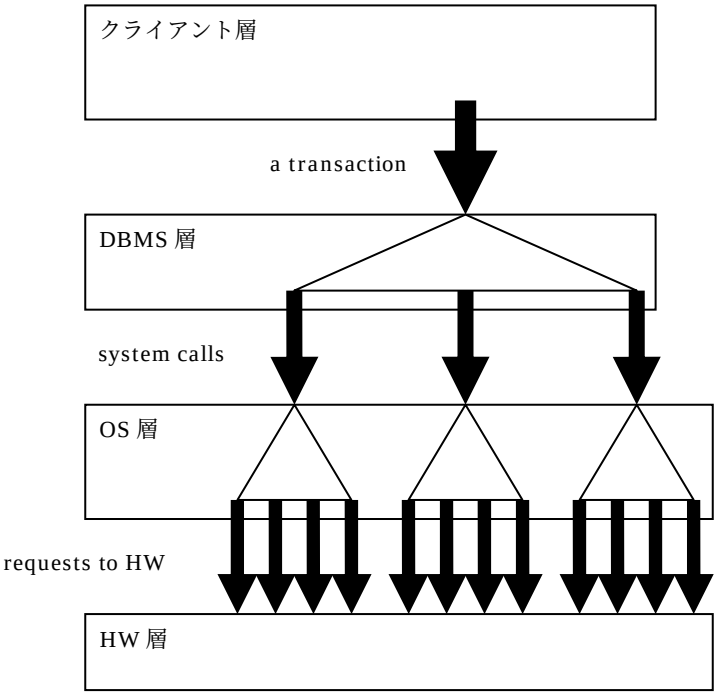


図 A.1-2 各層における request 数の増幅

図 A.1-2 に示すようにクライアント層が生成したトランザクションは、DBMS プログラムが受け取った後、一つ以上の request to HW へと変換される。クライアント層で発生した request 数は増幅されながら HW 層へ伝わることになる。

効率のよい DBMS であるためには、スループット[1]の悪化原因となる各層での不必要なリクエストを抑制しつつ、クライアント層からの transaction を HW 層へのリクエストに変換すべきことを示している。複数トランザクションを同時処理する場合は、ターンアラウンドタイム[2]を悪化させる要因であるリソースの競合を抑制することも考慮しなければならない。

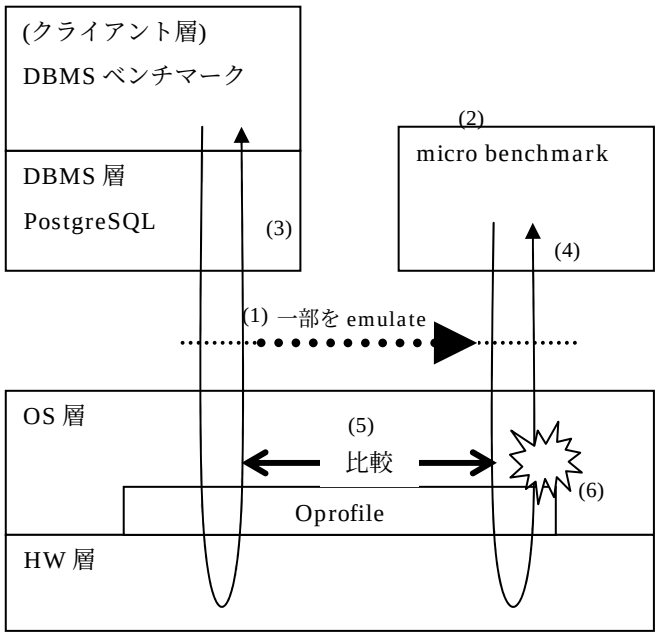
一方、ユーザの挙動で決まるクライアント層からの transaction が持つ性質に依存して、DBMS の最適な構造は変わってしまう。そこで典型的なクライアント層を想定して、負荷トランザクションを擬似的に生成し、DBMS 全体の処理能力を調べることが行われている。DBMS ベンチマークとは、このような擬似的なトランザクション層であり、負荷トランザクション数に対するスループットとターンアラウンドタイムを計測している。

[1]一定時間当たりの処理可能トランザクション数
[2]各トランザクション処理を完了するまでの時間

A.2 DBMS システムのボトルネックとマイクロベンチマーク

クライアント層の負荷がどのように HW 層への負荷となって現れるかは容易には予測できない。予測が難しい理由は、DBMS プログラム層と OS 層は共に複雑なシステムであり、かつ相互作用するためである。

DBMS ベンチマークは、複雑なシステムに、現実を模した負荷をかけて DBMS の有効性を計測するため、ボトルネックの発見には不向きである。



(1) DBMS 層が OS 層に与える負荷を単純にモデル化

(2) (1)に基づきマイクロベンチマークを作成

(3) プロファイル測定を実施しつつ、DBMS ベンチマークを動作実施

(4) プロファイル測定を実施しつつ、マイクロベンチマークを動作実施

(5) (3)と(4)のベンチマークの計測結果の相関性を確認。相関がある場合、双方のプロファイル情報は同じ要因をボトルネックとして提示しているはずである。

(6) より単純である(4)のプロファイル結果からボトルネックを発見

図 A.2-1　OS 層に存在するボトルネックの発見方法

そこで、マイクロベンチマークを使ったボトルネック発見方法が多く用いられてきた。主に OS 層に存在するボトルネックの発見方法は図 A.2-1 のようになる。

図 A.2-1 に示すように、DBMS 層で生成される OS 層への負荷の一部をエミュレートするようなプログラムを作成する事を考える。このような一部のみをエミュレートするプログラムを、マイクロベンチマークと呼ぶ。マイクロベンチマークによる負荷生成時と、DBMS ベンチマークによる負荷生成時の OS 層の挙動やプロファイル結果を比較することによって、OS 層が持つボトルネック要因が調べやすくなる。マイクロベンチマークと DBMS ベンチマークが同じようなプロファイル結果を示すならば、OS 層の主なボトルネックはマイクロベンチマークが生成している system call に対処する過程で発生していると言える。逆にマイクロベンチマークと DBMS ベンチマークの各プロファイル結果が全く異な

る結果を示すならば、マイクロベンチマークが提示していないどこかがボトルネック要因だということになる。

OS 層でのボトルネックを調査するにはマイクロベンチマークプログラム以外に、OS 層におけるプロファイラが必要になる。

あるプログラムを実際に実行しておいて、その挙動において繰り返し呼び出されている部分や、長い時間を消費しているルーチンなどを求める手法をプロファイリングと呼ぶ。プロファイリングのために必要なプログラムをプロファイラと言う。図 A.2-1 では、プロファイラとして Oprofile を利用した場合を示している。

今回用いた Oprofile ver.0.5.4 は、周期的サンプリングという手段を用いて、関数が消費している時間を計測している。

Oprofile はまず、全論理空間を一定のメモリサイズからなる区画に分ける。そして、各区画ごとに利用頻度カウンタを用意し、Oprofile 起動時にそのすべてを 0 に初期化する。

プログラム(OS を含む)実行中は、一定周期(周期自身は指定可能)ごとに CPU が実行している論理アドレスを調査し、対応するメモリ区画の利用頻度を 1 増やす。

プロファイル終了後、シンボルテーブル表を用いて、各メモリ区画に対応する関数の実行時間を利用頻度とサンプリング周期から割り出す。ただし、単一メモリ区画内に複数の関数が存在する場合は、代表となる関数を 1 つ決め、その関数の実行時間とする。

マイクロベンチマークで検出できるボトルネックについては、OS 層、DBMS 層が共に固定されている間は徐々に発見され改善されて減っていくと予測される。したがって、長期的に残ることはなく、将来的には容易には予測できないボトルネックのみがシステムに残ることになる。一方で、どちらかの層が変更されると、容易に予測されるボトルネックは潜在的に増大する事が予測される。

A.3 Linux の採用と容易に予測できるボトルネックの調査

昨今のトレンドとして、旧来のプロプライエタリ OS に代わって Linux を OS 層に用いる事が増えている。先に述べたように、OS 層の変更は容易に予測できるボトルネックが復活した可能性が高い事を意味している。

Linux は他のプロプライエタリ OS と比べ、DBMS 用 OS として使われた歴史は短く DBMS を運用した場合に適した動作をしているとは限らない。また、DBMS は多くの OS に対応するため、旧来のシステムコールのみを使用している場合が多く、Linux で実装されている比較的新しく効率的なシステムコール等を使用していない可能性がある。

B PotgreSQL におけるトランザクション処理と disk の I/O

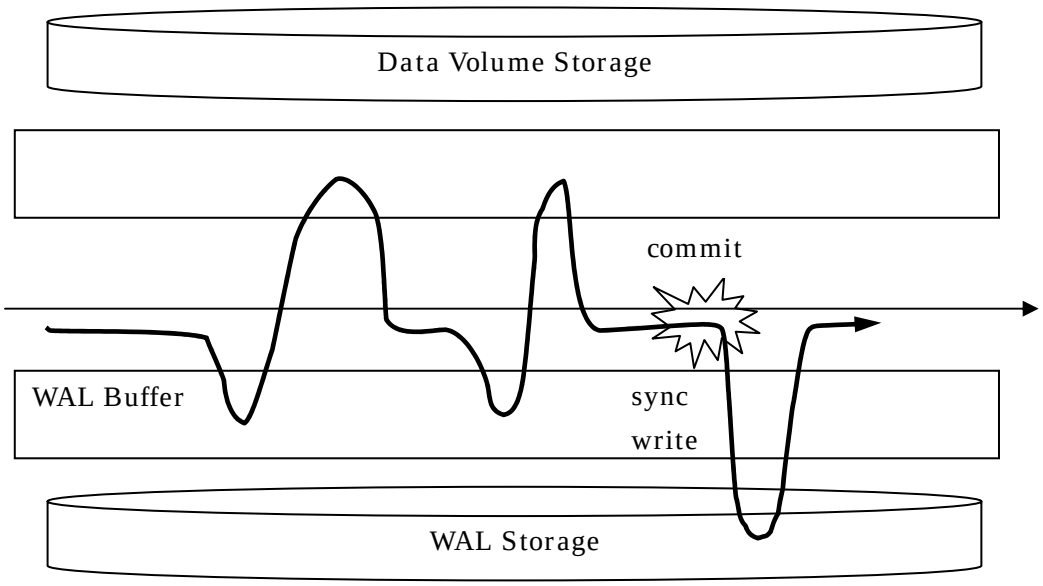


図 A.3-1 トランザクション処理中の書き込みの流れ

図 A.3-1 に、あるトランザクション処理に着目した、共有バッファならびにストレージへの書き込みを例として示す。トランザクション中の処理を実行するために、まず WAL Buffer へ変更処理を記録し、続いて Data Volume Buffer を更新する。更新を必要なだけ(図では 2 回)行った後、Commit 処理がスタートすると WAL Buffer 中のデータを WAL storage に同期書き込みを行う。

この例では ２つの buffer が一杯になる場合は考慮していない。WAL buffer が一杯になった場合は、commit 処理以外のタイミングでも同期書き込みが行われる。

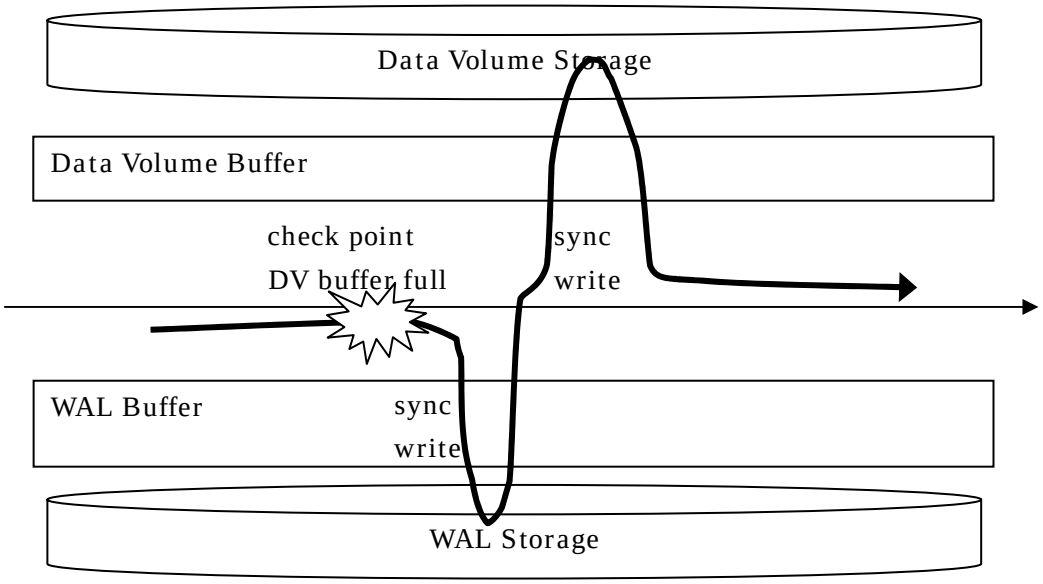


図 A.3-2　check point　あるいは Data Volume Buffer full 発生時の処理

図 A.3-2 に check point、あるいは Data Volume Buffer full が発生した場合の処理を示す。まず、WAL buffer の内容が WAL Storage に同期書き込みを用いて反映される。次に、Data Volume Buffer の内容が Data Volume Storage に同期的に書き込まれる。

非同期書き込みは、実際には kernel が管理するファイルキャッシュに変更が反映されるだけで、Storage への更新は kernel 上のバッファメモリ不足や、一定時間以上の dirty 状態の保持、swapd による sync 命令などが発生するまで実行されない。また、DBMS ベンチマークごとに書き込み量なども異なり、さらに物理メモリ量によってその挙動は変化する。

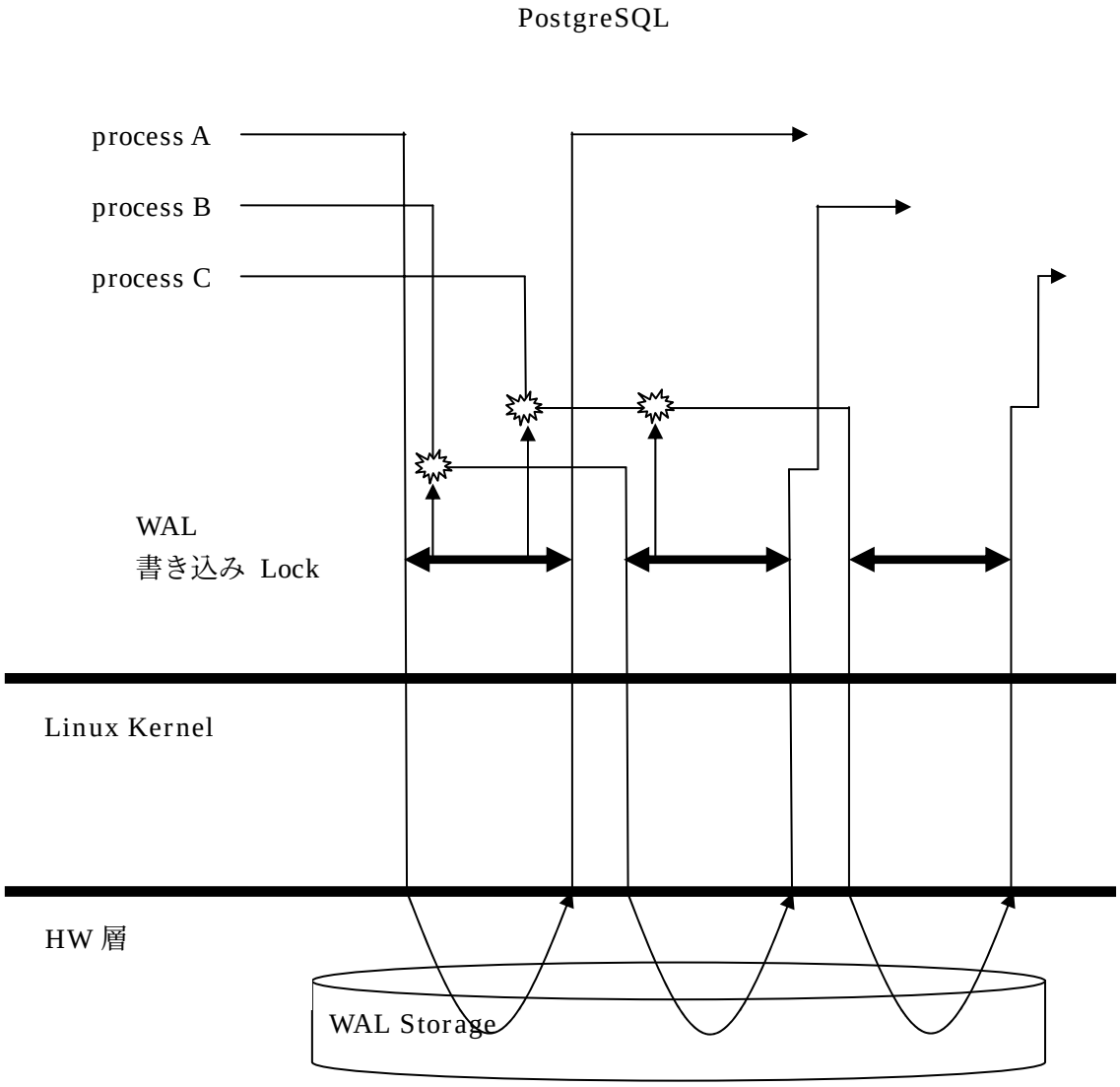


図 A.3-3　複数プロセスの WAL への書き込み

図 A.3-3 に、複数の PostgreSQL プロセスから WAL ファイルへの書き込み要求があった場合の動作を示す。

この図では Process は A,B,C の 3 種類があり、それぞれ何らかの理由で WAL 書き込みが必要になったとする。PostgreSQL は自前の排他制御用ロックを持っており、WAL 書き込みは一度に 1 つのプロセスしか実行できない。図では、Process A が WAL への同期書き

込みを行っている間、Process B, C は共にブロックされる事、Process B の番が回ってきても Process C はまだロックされている事を示している。

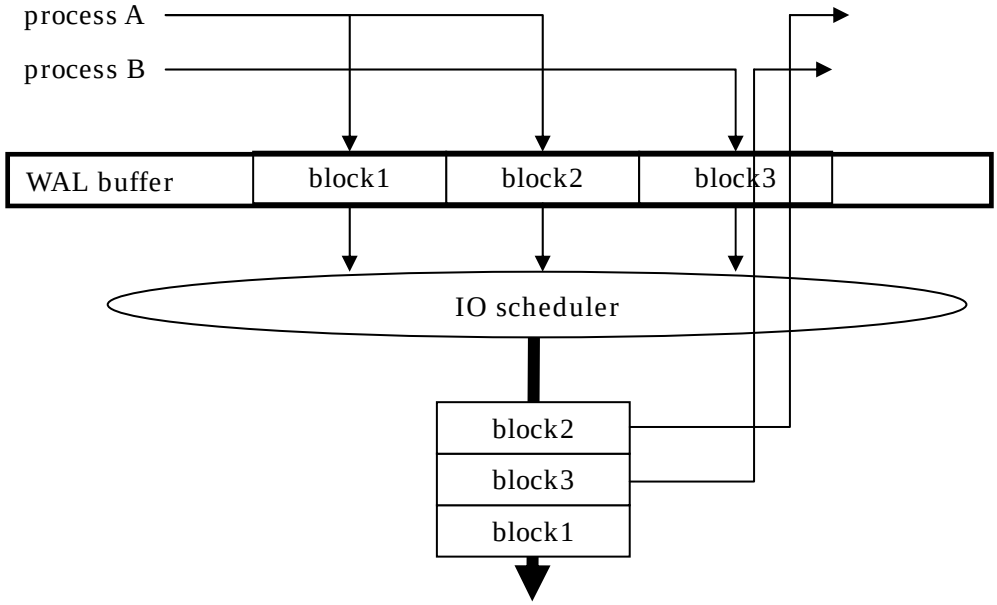


図 A.3-4　I/O scheduler の影響

PostgreSQL が WAL 書き込みについて自前の排他機構を持っているのは、WAL の書き込み順序を保障するためである。これは I/O scheduler というものが影響している。図 A.3-4 に I/O Scheduler の影響を示す。この図では process A は WAL buffer の block1 と 2 を、process B は block3 を、それぞれ同期書き込みしようとしているが、PostgreSQL の実際の実装と異なり、process 間で排他制御が無い場合を示している。

process A は先に block 1,2 の同期書き込みを要求したとする。当然、process A は block1,2 の両方が書き終わるまで、制御は戻ってこない。その直後に process B が block3 への書き込みを要求したとする。どちらも commit が同期書き込み要件の発生理由だとしよう。

何らかの理由で I/O scheduler は指定された 3 つのブロックを 1,3,2 の順序で storage に書き込む事を決定したとしよう。この場合、block3 が書き終わった段階で process B への制御は戻るが、process A は block2 が書き込まれていないため制御は戻らない。process B は WAL 書き込みが完了したため、commit 完了をクライアント層に通達する。

仮にこの直後、block2 への書き込みが完了する前にシステムがダウンしたとする。再開後、storage 上の WAL を調べると block1 は書き込み完了、block2 は書き込みが終了していない。従って、PostgreSQL は block1 に書き込まれている情報までを元に、データベースを再構築する。block3 を書き込んで commit されたはずの情報は復元できない。

このような障害を回避するには block3 は block2 を書き終わるまで storage に書き込ん

ではならない。しかし、一般的なファイルシステムでは異なるプロセスからの同一ファイルへの書き込みは、書き込み領域に重複が無い限り順序不同でも構わない、としているため OS 層以下で順序制御をする事はできない。PostgreSQL の WAL 書き込みロックはこのような block 書き込み順序を確定するために用いられている。

WAL 書き込みロックの存在は、WAL ファイルへの書き込み命令が、OS 層に対し複数同時に発行される事は無い、という事を意味する。純粋に書き込み要求負荷という観点から見ると、同期書き込み要求は 1 つのプロセスから繰り返し発行しても、複数のプロセスが(プロセス間で排他制御しながら)繰り返し発行しても、変わりは無い。

C Linux のファイル I/O 方式

1) open の方法

open の方法は非同期型 open と同期型 open の 2 種類がある。

・Async open は非同期型 open を表わし、open(2)に対して同期を指定するようなフラグを与えずにファイルを開く方法である。Async open を行った場合には write(2)を行うだけでは同期が取れない。従って、同期を取るためには、fsync(2)または fdatasync(2)を呼び出すか、sync(2)を 2 回呼び出す必要がある。

・O_SYNC open、O_DSYNC open はそれぞれ同期型 open の一方式を表わし、open(2)に O_SYNC,O_DSYNC をそれぞれフラグとして与えることで指定できる。各同期型 open を用いれば write(2)を行うだけで同期的に書き込むことができる。

2) AIO

io_submit(2),io_getevents(2)は Linux kernel 2.5 から登場した非同期に I/O を実現するシステムコールである。

・io_submit(2)は非同期な I/O リクエストを処理待ちキューに登録する。

・io_getevents(2)は完了キューから非同期な I/O イベントを読みだす。

3) mmap

mmap(2), mmap2(2), mmap64(2)は、ファイルやデバイスをメモリにマップするシステムコールである。

・msync(2)でファイルやデバイスとマップしたメモリを同期させることができる。