7 ファイル管理

7–1 基本的な考え方

ファイルシステム存在の必然性：

- 計算・処理すべきデータやその結果などが大量に発生すると、
  それらを蓄積する必要が生じる。
- コンピュータを利用する人たちが増えると、
  それらの人たちの間でデータを相互に交換する必要が生じる。

⇨ 記録媒体上に格納し共通的に扱えるデータ型式
が必要になってくる。

. . . . . .

⇨ ファイルシステム

p.15 からの引用：
入出力機器は機器毎に制御の仕方が違う。
⇨ 入出力などの共通機能を容易に実行できる環境をハードウェアに付随して提供すれば、プログラマの生産性向上につながる。

各種入出力機器内のデータの統一的な扱い：

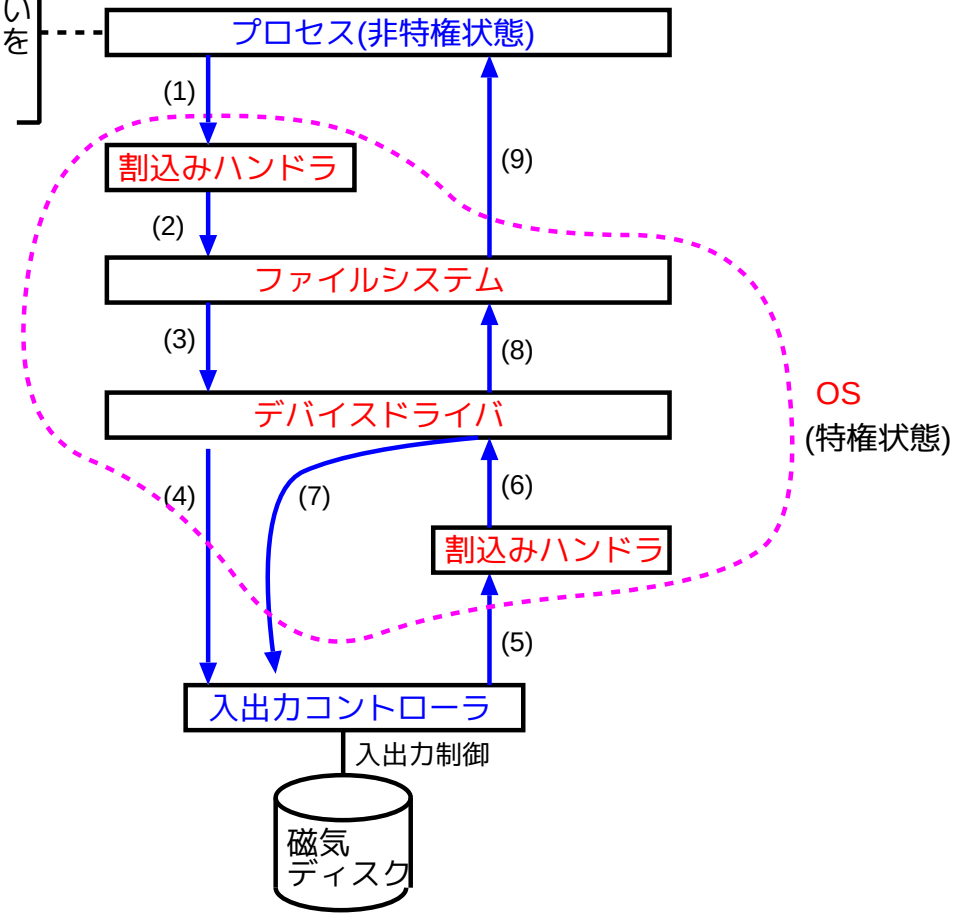
- データは様々な入出力機器内に蓄積されている。 例えば、磁気ディスク1, 磁気ディスク2, CD-ROM, DVD-ROM, MO, USBメモリ, ... 。これらの装置を統一的に扱いたい。
- データを処理する プログラムの側から見れば 、入力データの入ったファイルもキーボードも「データ列を提供してくれるもの」という点では同じ。　また、出力データを入れるファイルもディスプレイも「出力データ列を受け取るもの」という点では同じ。

⇨ 入出力機器の扱うものは全てファイルと考える。
これらのファイルを統一的に扱うための仕組みが ファイルシステム。

ファイル管理の目標：

- 各々のファイルを識別するための論理的な(i.e.物理的な媒体を意識させない)名前がある。

物理的な媒体を感じさせない
論理的なインターフェースを
通じて入出力機器内の情報
(ファイル)を扱う



これにより、例えば、
ファイルを記録する媒体が(容量拡大などのために)変わっても、一般プログラマは従来のファイル操作のプログラムを変更する必要がない。

ファイル管理の目標：

- 各々のファイルを識別するための論理的な
  (i.e. 物理的な媒体を意識させない) 名前がある。
- 記憶媒体の容量を無駄なく使い切る。
- ファイルへのアクセス時間が短くなるようにする。
- データの信頼性を保つ。(e.g. **fail-safe**)
- プログラムの中からファイルを操作するための
  豊富なインターフェースが用意されている。

7–2 ファイルアクセス高速化の工夫

7–2–1 ブロッキング

レコードの概念：

OSが利用者プログラムに対して入出力機能を提供する際の入出力の基本単位 (i.e. 1回の入出力操作でOSとの間で受け渡しするデータ) を論理レコード、またはレコードと呼ぶ。

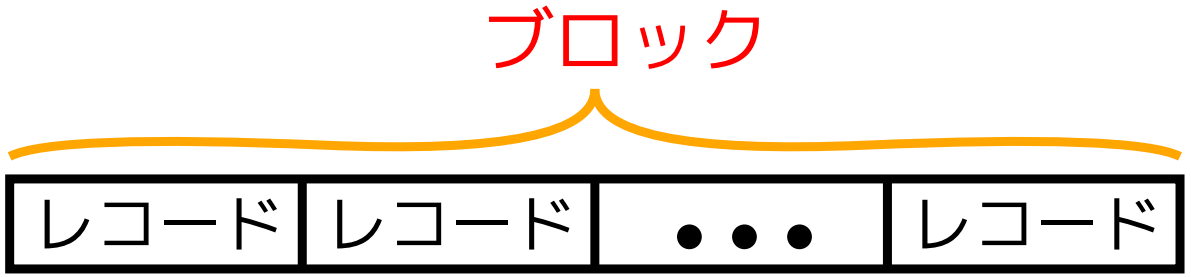
ブロッキング：

入出力実行回数を減らして総計の入出力時間を短縮させることをねらって、複数のレコードをまとめて1回の入出力を行う方法がある。これをブロッキングという。

補足：「レコード」という用語は次の様な意味で使われることもある。

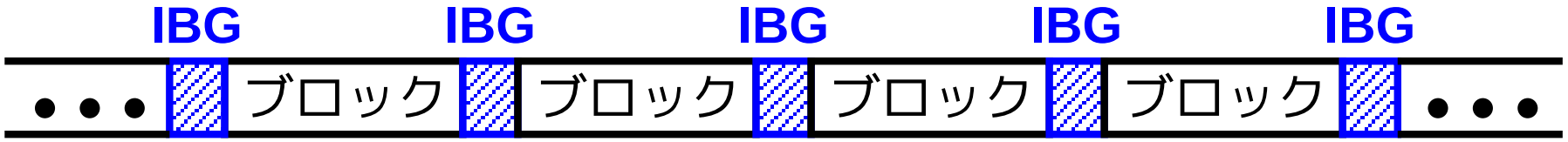
- ファイル構成の際の 基本単位。
- プログラミングにおいて、 関連するデータを 1 つにまとめたもの。C言語では構造体と呼ぶ。

- ブロック ··· 入出力の単位となるレコードの集まり
- ブロッキング・ファクタ ··· 1つのブロックの中のレコード数



磁気テープにブロック列を記録する場合：

ブロックとブロックの間に**IBG**(**Inter Block Gap**)あるいは**IRG**(**Inter Record Gap**)と呼ばれる領域をおく。



⇨ 磁気テープに格納される実効的なデータの割合

$$= \frac{\text{ブロックサイズ}}{\text{ブロックサイズ}+\text{IBGサイズ}}$$

ブロッキング・ファクタをどう選ぶか?:

ブロッキング・ファクタが大きければ大きい程1つのレコードに対する平均のアクセス時間を短くすることが出来る。 しかし、次の点も考慮しなければならない。

- ブロックの先頭レコードに対するアクセス時間が大きくなり過ぎないか？
- 入出力バッファのサイズが大きくなり過ぎないか？
- 実際のCPUオーバヘッドはどの程度になるのか。(ブロッキングの効果はどの程度か。)

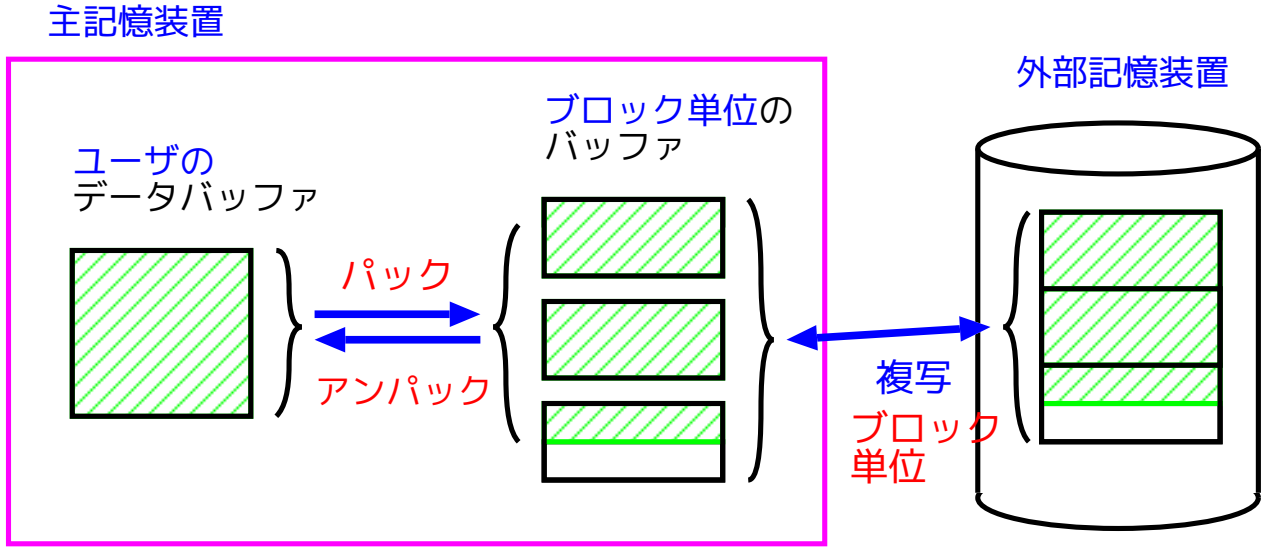
1つのレコードに対する平均のアクセス時間の改善
(CPUオーバヘッドの改善)

$\Longleftarrow$ tradeoff $\Longrightarrow$

- ブロックの先頭レコードに対するアクセス時間の改善
- 入出力バッファのサイズを小さくできる

7–2–2 バッファリング

並行して動作するCPUと入出力コントローラの同期を取るために、CPUと入出力コントローラとの間にバッファと呼ばれる主記憶内のデータ受渡し場所を用意する方法がある。



複数のブロックから構成されるファイルを読み書きする際は、

- バッファ⟺入出力装置間のデータ転送と
- プロセスの作業領域⟺バッファ間のデータ転送

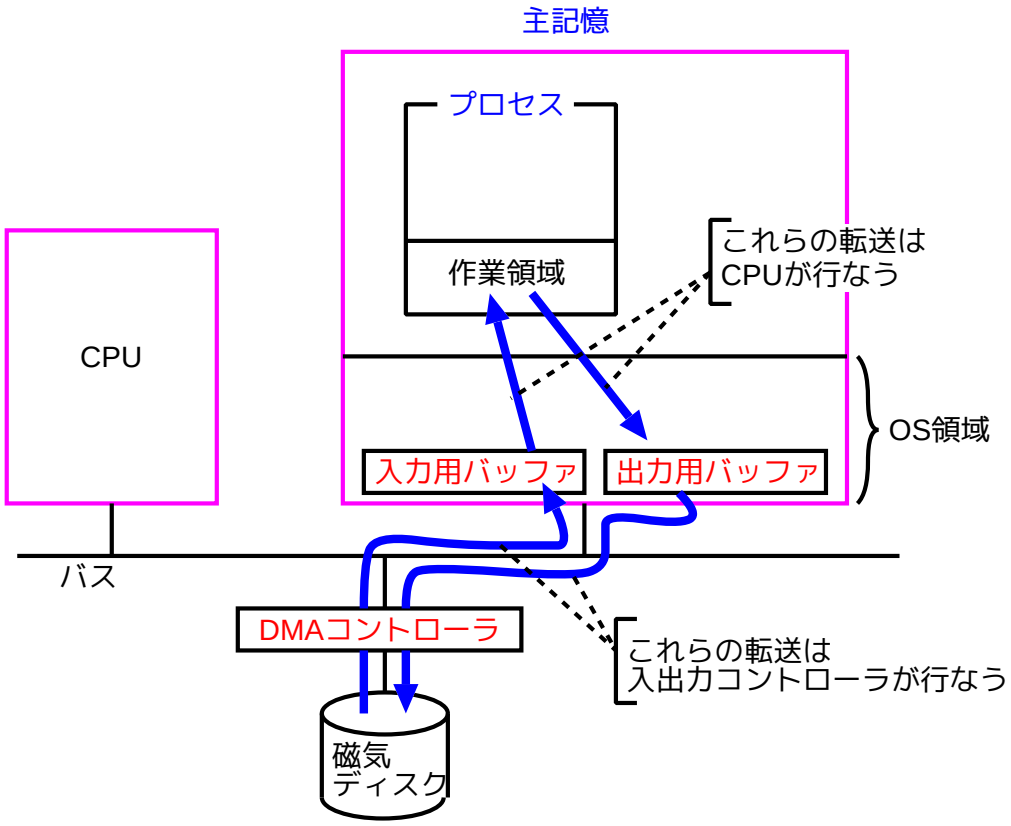
が入出力コントローラとCPUにより交互に繰り返される。

1面バッファリング：

入力用,出力用に容量が1ブロックのバッファが1面ずつ用意されている場合を考える。この場合に4つのブロックから構成されるファイルを読み出そうとすると、

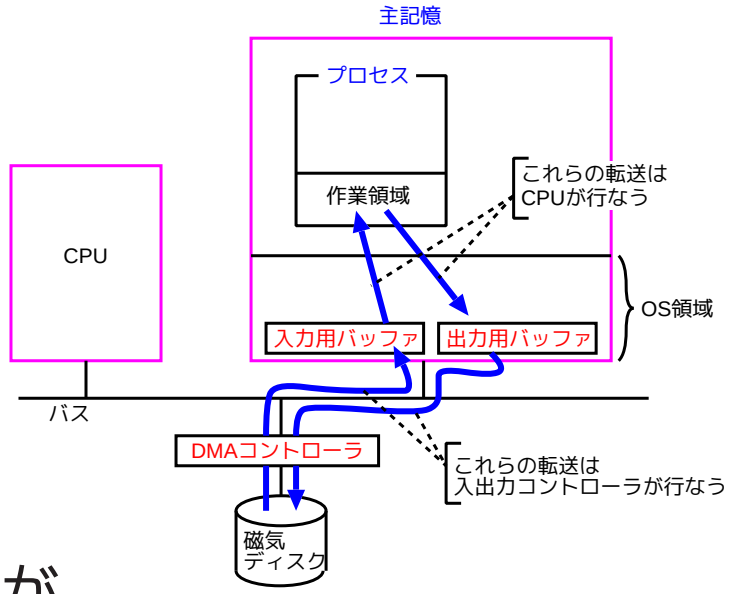
- DMAコントローラによる　磁気ディスク⟶バッファ
- CPUによる　バッファ⟶プロセスの作業領域

のデータ転送が交互に繰り返される。

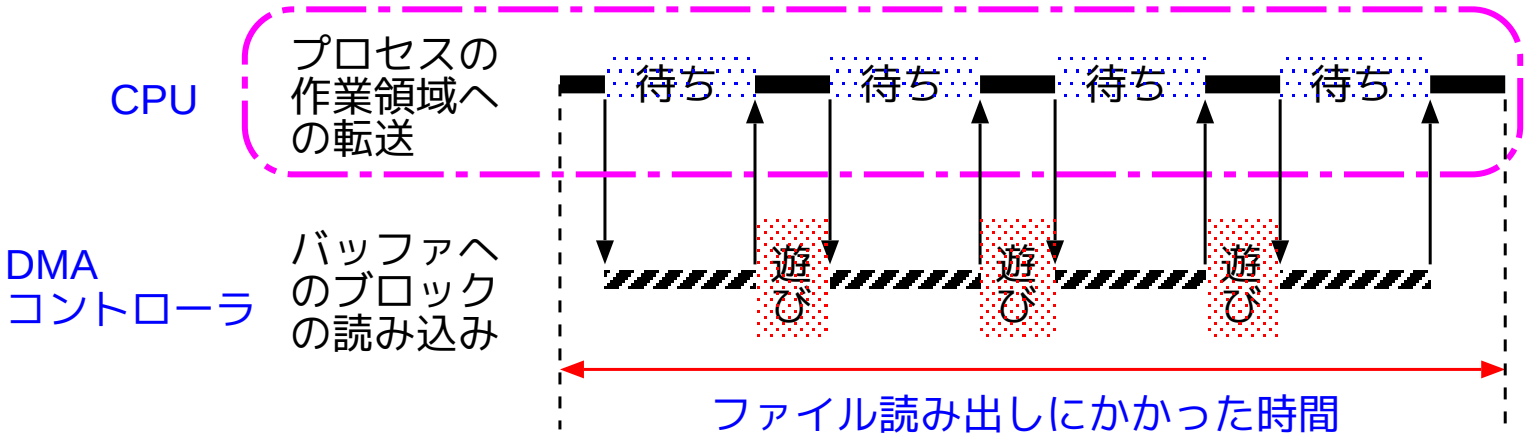


注目：

DMA コントローラと CPU は同時に 1 つのバッファにアクセスしない。

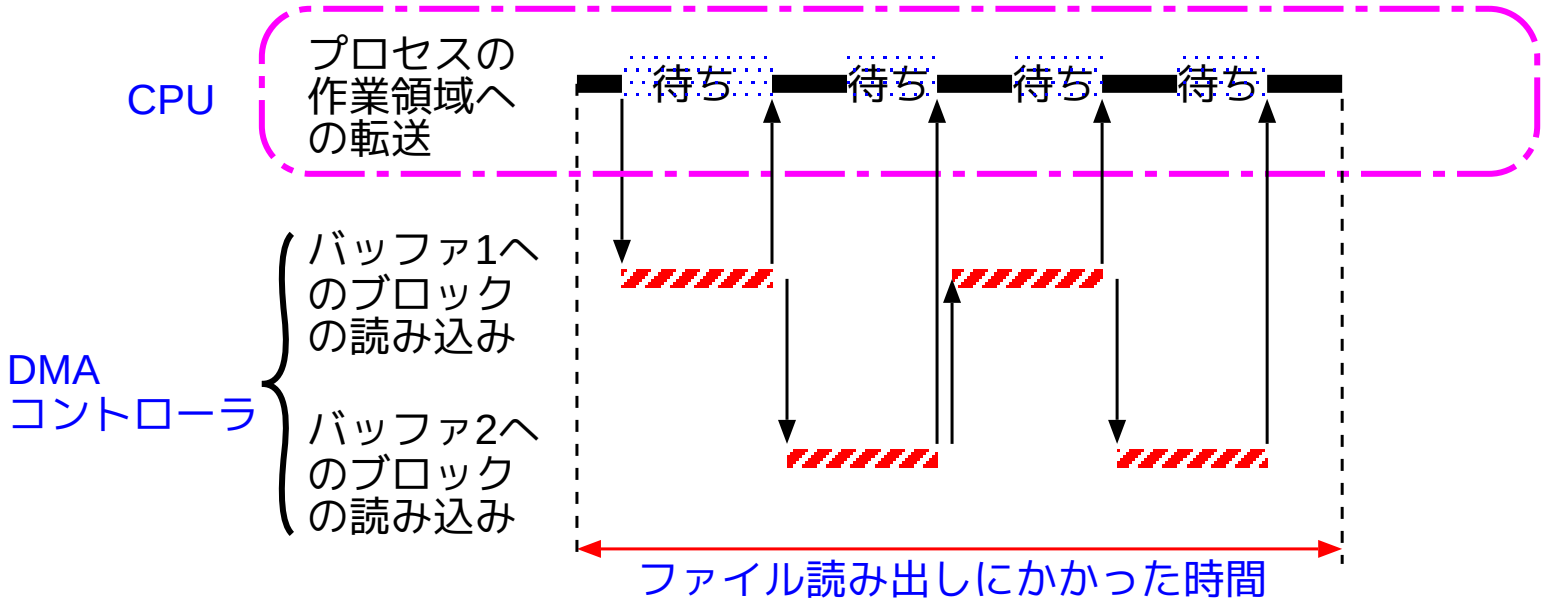


この様に、
容量が1ブロックのバッファが1個だとCPUが
プロセスの作業領域$\longleftrightarrow$バッファ
間のデータ転送を行っている間は(CPUに比べて低速な) 入出力装置が遊んでしまう。



2面バッファリング：

容量が1ブロックのバッファを2個用意して、バッファが満杯になるか入出力要求が出されるかする毎にバッファを交互に切替えて使うようにすれば、CPUがバッファにアクセスしている時にも入出力装置を動作させることが出来るようになる。



多面バッファリング：

出力の場合は、バッファの個数が多いほど出力待ちの可能性が少なくなり、効率の良いプログラム実行が可能になる。一般に、複数個のバッファを用いたバッファリングを多面バッファリングと呼ぶ。

補足：

バッファリングに関しては、読む本によって用語の使い方が違う。例えば、

- 岩波情報科学辞典では、...
  バッファを複数個もつことにより入出力の効率を上げる方法を「バッファリング」と呼んでいる。
- 清水5.2節では、...
  プロセス内の作業領域もバッファと見て、プロセス内の作業領域とOS内に用意される転送用バッファを一組にして使用する方法を「ダブルバッファリング」と呼んでいる。

7–2–3 キャシング

外部記憶のブロックを一時的に置くための領域(キャッシュと呼ぶ) を主記憶上に用意し、そこに外部記憶上の頻繁にアクセスするブロック(ホットスポットと呼ぶ)を保持する。

これをキャシングと言う。

キャッシュ記憶： ···(これはハードウェアの話, p.6)

CPUから主記憶内のデータへのアクセス速度を見かけ上高速化するための高速記憶。

- アクセスされた主記憶の情報はキャッシュ記憶内に一時的に格納され、近い将来再びアクセスされた時はキャッシュ記憶から取り出される。
- メモリへのアクセス回数を抑えるため、キャッシュ記憶に無いデータをメモリから読み込む場合には、必要とするデータだけでなく近くのデータも一緒に読み込んで複製しておく。
- プログラムの実行が局所参照性を持っているので、キャッシュ記憶を用いることによって主記憶内のデータへの実際上のアクセススピードを上げることが可能になる。

キャッシュを用いた場合の注意点：

- キャッシュ上の情報と磁気ディスクなどの記録媒体上の情報が一致しないことがある。
  ⇨ システムダウンの時は新しい情報が失われてしまう。

  何らかの時点で、キャッシュ上の情報更新の結果を記録媒体上に書き込まなければならない。

- キャッシュ内のブロック領域が不足した時にどのブロックを解放するかをうまく決めないといけない。

  「最近使用されたブロックほど近い将来も再度参照される可能性が高い」という仮定の下に、解放するブロックを決めることがある。

UNIXの場合：

- カーネルが自動的に入出力のキャッシングを行っている。
- 一般的なファイルアクセスを行っている場合はキャッシュに情報が存在する。
- キャッシュ上の情報と磁気ディスクなどの記録媒体上の情報の不一致を強制的に解消するためには、システムコール `sync()` を実行する。

7–3 信頼性確保の工夫

現代のコンピュータは計算だけでなくデータ処理に利用されている。

⇨ ファイルシステムを誤り無しに保つことは重要。

7–3–1 ファイルシステムになぜ誤りが生じるのか

例7. 1 (キャッシュ上のデータの損失)
コンピュータシステムが不意にダウンすると、キャッシュ上に行われたデータ更新記録が全て失われてしまう。

⇨ この誤りは別の所に波及しないし、ある程度は仕方ない。
(ファイルのアクセススピード改善とのtradeoff。)

例7. 2 (ファイルシステムの矛盾)

一般にファイルシステムは、

- 記憶媒体中の不使用領域を登録した管理テーブル,
- 各々のファイルの所在場所を記録した管理テーブル (i.e. ディレクトリ),
- ファイル (ディレクトリ以外) の実体

から出来ている。

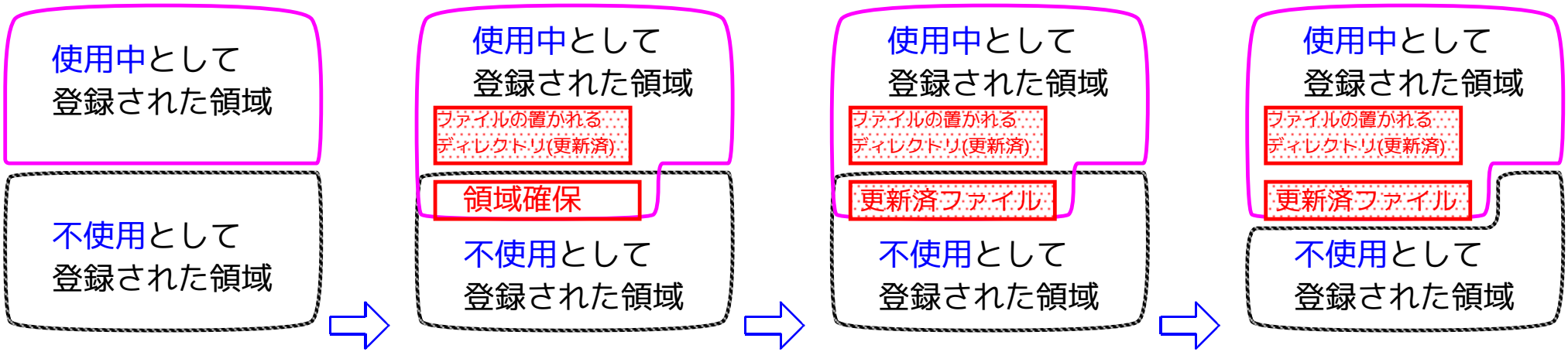
⇨ 不使用領域の管理テーブルとファイルの所在場所の管理テーブルを同時に更新できないので、ファイルシステムが一連の操作を行っている途中の段階では一時的にデータ構造内に矛盾が生じてしまう。

⇨ これらの情報更新の最中にコンピュータシステムが不意にダウンすると、ファイルシステムに矛盾が残ったままになってしまう。

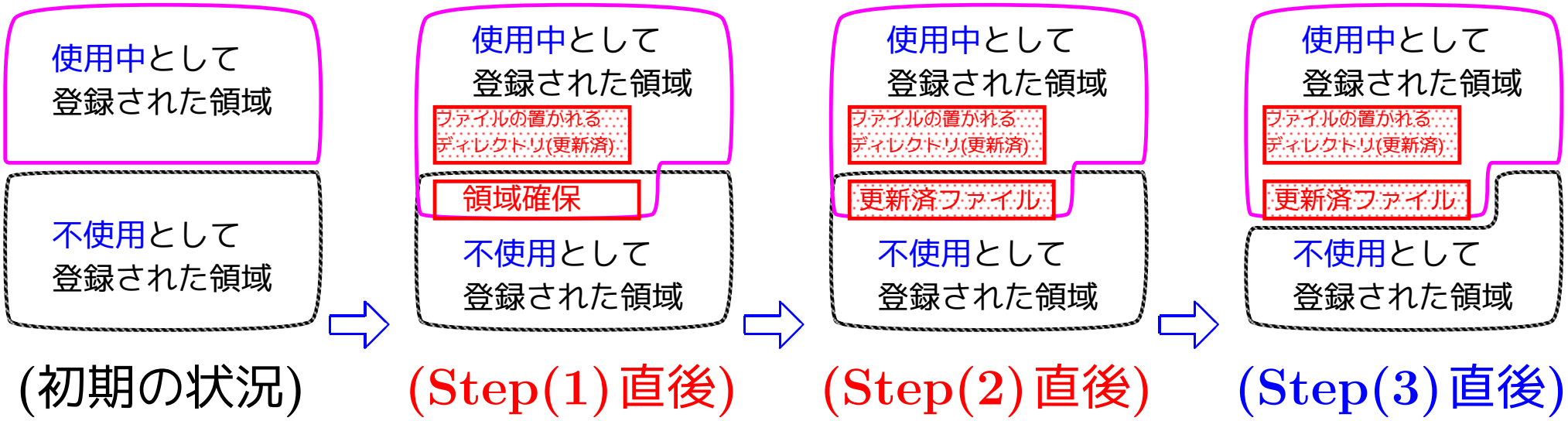
例えば、ファイルの新規生成に伴う情報の更新を

(1) 各々のファイルの所在場所を記録した管理テーブル (i.e. ディレクトリ
(2) ファイル (ディレクトリ以外) の実体,
(3) 記憶媒体中の不使用領域を登録した管理テーブル

という順に行った場合、...



(初期の状況) (Step(1) 直後) (Step(2) 直後) (Step(3) 直後)

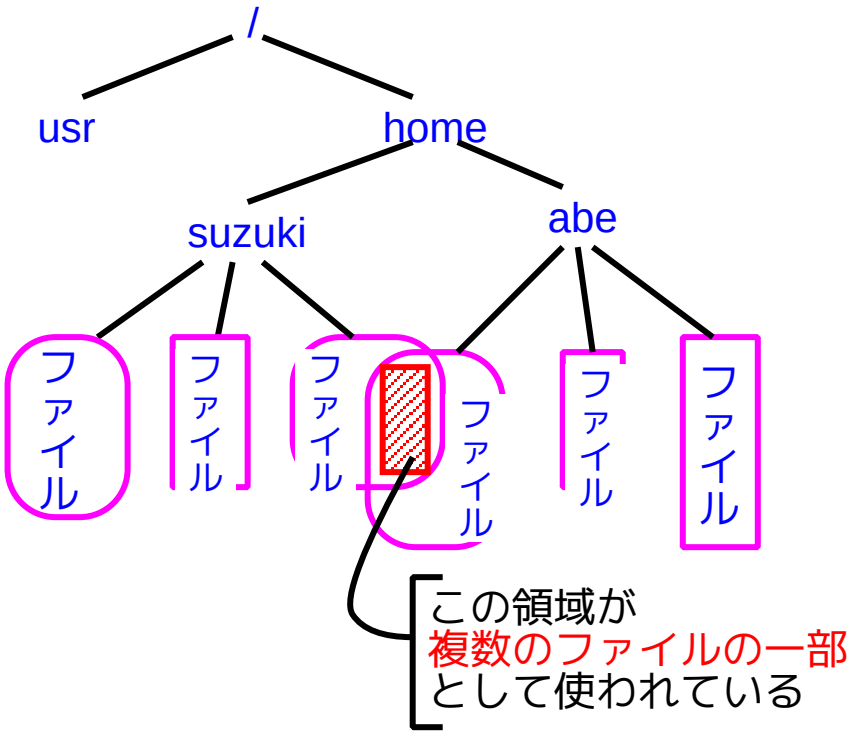


ステップ(1)の直後には

新規ファイルのための領域は確保されてはいるが、それはまだ不使用領域として登録されたままの状態になっている。

⇨ ステップ(1)完了からステップ(3)完了前までの間にコンピュータシステムがダウンすると、ファイルの実体が不使用領域としても登録されたままという矛盾が残り、

その後この領域が別のファイルのために使われると、同一の領域が複数のファイルの一部として使われるという事態になってしまう。



例7. 3 (矛盾の残存・波及)

一旦 上図のような矛盾に陥ってしまうと、この

1つの領域が**2**重に使われている状況

からはなかなか抜け出せない。この領域を使っているファイルが削除されても、この領域の登録が不使用領域の管理テーブルに移されるだけで再びこの領域が利用されると再び上図の状況に戻ってしまう。

この領域を使っているファイルが**2**つとも削除されるなどしてこの領域が完全に(**i.e.2**重に)解放されれば、不使用領域の管理テーブルにこの領域を登録しようとした時に矛盾が検知されるかも知れない。

7–3–2 fail-safeなシステム

ファイル管理においてはfail-safeなシステムを作ることが重要である。

⇨ • 仮にコンピュータシステムがダウンしてもファイルシステムの中に悪質な矛盾が生じたり、誤りが伝搬したりするのは絶対に避けなければならない。

• そういった矛盾を早期に発見したり、ファイルシステムの崩壊の場合に回復したりする手立ても用意しておかなければならない。

フォールト・トレランス (fault tolerance) ··· 故障によっても、全面的なシステムの停止に至ることはなく、...
フェール・ソフト (fail-soft) ··· (failが発生してもsoftなものにする、ということ。)
フォールト・アボイダンス (fault avoidance) ···
フェール・セーフ (fail-safe) ··· 故障が発生しても、予め定められた安全状態にシステムを固定し、...

具体的には、ファイルシステムの信頼度を保つために次のような手立てが取られている。

- ファイル管理情報更新の順序を工夫する：
  ファイルを新規登録する場合は、ファイルシステムの情報を
  (1) 記憶媒体中の不使用領域を登録した管理テーブル,
  (2) ファイル(ディレクトリ以外)の実体,
  (3) 各々のファイルの所在場所を記録した管理テーブル(i.e.ディレクトリ),
  という順に更新する。

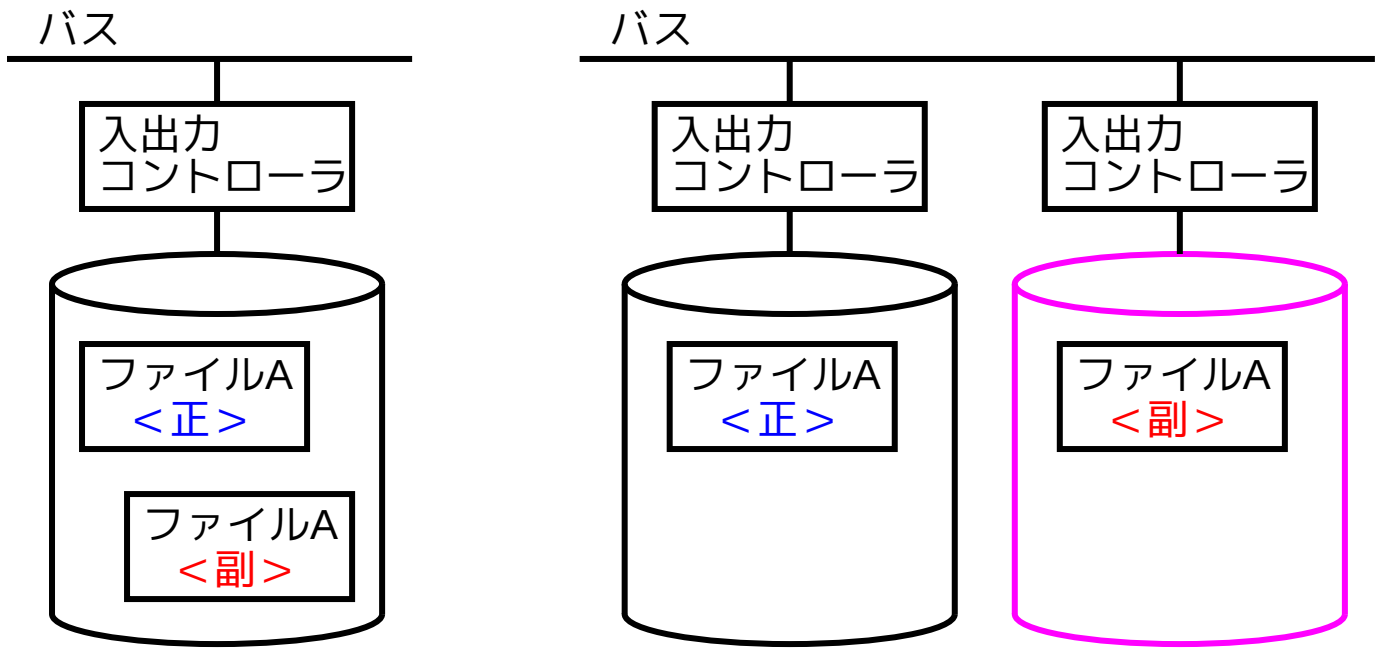
  この様に変更した場合でも、ステップ(1)完了からステップ(3)完了前までの間にコンピュータシステムがダウンすると、どの管理テーブルにも登録されない領域が出来るという矛盾が出来るが、2重の領域割当てのような深刻な問題は引き起こさない。そういう意味で、確かに fail-safe なシステムになっている。

- ユーティリティソフト：
  ファイルシステムの保守も大事で、各種ユーティリティの支援の下で定期的または不定期的にファイルシステムの診断が為される。

- 多重ファイル方式： システムに冗長度を持たせる。

  ファイル・ミラーリング ··· 下図 左の通り。
  1つの入出力制御機構に接続された装置上に同一のファイルを2重に保有する。

  ファイル・デュプレックス ···下図 右の通り。
  ファイルだけでなく、それを入れる入出力ハードウェアも2重化する。



ミラーリング　　デュプレックス

- バックアップ：
  障害時の回復のために、磁気ディスク上のファイルを全て磁気テープ上にコピーするなど。自動化されたハードウェア機構もある。

  場合によっては、システムを運用している間に行う、オンラインバックアップという方法が取られることもある。

- ジャーナル：
  OLTP(**On-Line Transaction Processing**)においては、信頼性の要求が厳しい。
  システム障害があった時でも、数秒~ 数十秒後(?)には各ファイルの内容を最新の状態に戻しておかなければならない。

  ⇨ オンラインバックアップに加えてファイルの更新履歴をトランザクション毎に取っておく。

  このようなファイル更新や端末へのメッセージを記録した情報をジャーナルと呼ぶ。

7–4 UNIXにおける論理化したインターフェース

ファイルシステムは次の様なファイル操作のインターフェース、機構を我々ユーザに提供しなければならない。

(a) 各々のファイルを論理的に (i.e. 物理的な記憶媒を意識することなく) 識別する機構,

(b) ファイルアクセスの保護機構,

(c) ディレクトリやファイルの新規作成,

(d) ファイルの読み込み, 加工, 更新,

(e) ファイルやディレクトリの消去

これらの要請の各々をUNIXがどのように実現しているかを、以下では概観しよう。

(a) 各々のファイルを論理的に識別する機構：

- 木構造でファイルを分類して管理。
- 木構造の葉節点がファイル、内部節点 (i.e. 葉以外の節点) がディレクトリに相当する。
- 例えば、下図では、右下の `stdio.h` とラベル付けされた節点位置に配置されたファイルは

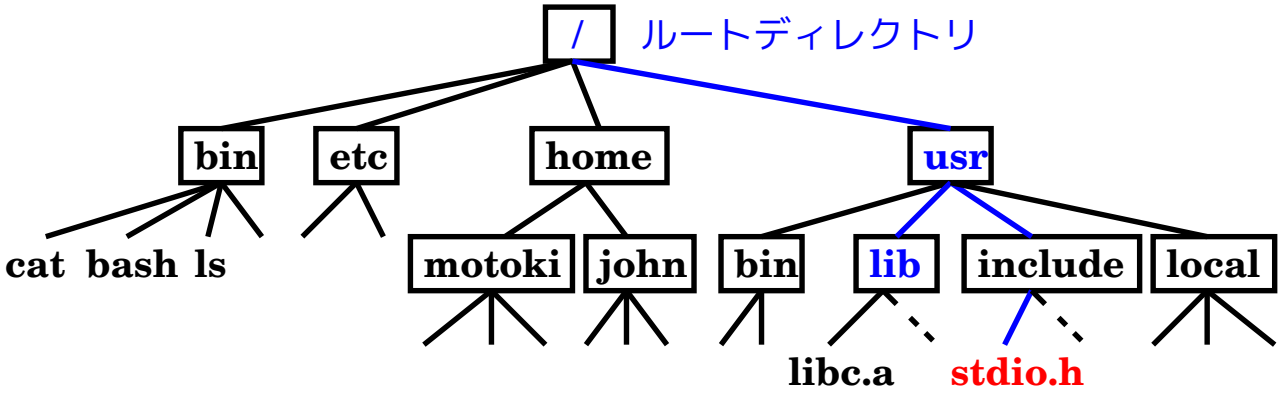
  `/usr/include/stdio.h`

  という名前 (絶対パス名と言う) で識別される。また、

  `/usr` 上で作業をしている時は　`include/stdio.h`、

  `/usr/lib` 上で作業をしている時は　`../include/stdio.h`

  という名前 (相対パス名と言う) で識別される。



(b) ファイルアクセスの保護機構：

- ファイルの中には、記録・保存したいデータそのものの他に次の様な属性が格納されている。

ファイル本来の性格 (本人に対するアクセス権限)	読み出し可能かどうか, 書き込み可能かどうか, 実行可能かどうか
グループ内ユーザに対する アクセス権限	読み出し可能かどうか, 書き込み可能かどうか, 実行可能かどうか
任意の他人に対する アクセス権限	読み出し可能かどうか, 書き込み可能かどうか, 実行可能かどうか
その他の ファイル属性	作成者, 作成年月日, 最終更新年月日, ファイルサイズ, ファイル実体の所在マップ, ・・・・・・・・・・・・・・・・・

(c) ディレクトリやファイルの新規作成：

- ディレクトリもファイルの一種。
- 一般のファイル作成のためのシステムコール
  `creat()`, `open()`
- ディレクトリ作成のためのシステムコール
  `mkdir()`, `mknod()`
- ファイル作成時にアクセス権限の情報を与える。

(d) ファイルの読み込み,加工,更新：

- ファイルオープンのシステムコール`open()`
- ファイルオープン時に、そのファイルをどのようにプログラム内で使用するかも宣言する。

補足：
そのファイルの保護属性と照らし合わせて、そういう使いかたが許されているかどうかがチェックされる。

- ファイルオープンが完了すると、ファイル記述子 (小さな整数) が返される。このファイル記述子をトークン (アクセス許可証) として用いて読み書きやクローズのシステムコールを実行する。
- ファイル読込みのシステムコール`read()`
- ファイル書込みのシステムコール`write()`
- ファイルクローズのシステムコール`close()`

(e) ファイルやディレクトリの消去：

- ファイルやディレクトリ消去のコマンド`rm`
- ファイル消去のシステムコール`unlink()`
- ディレクトリ消去のシステムコール`rmdir()`

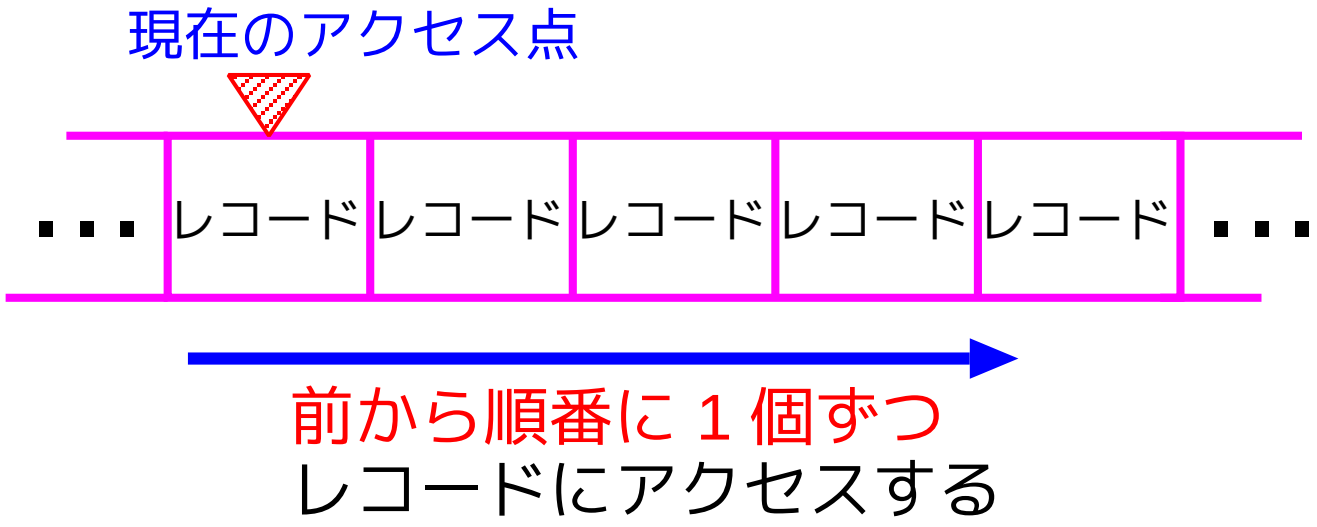
7–5 順編成 vs. 乱編成

ファイルはその論理構造、すなわちその中のレコードの編成方法(とそれに伴うレコードへのアクセス方法)によって次のように分類される。

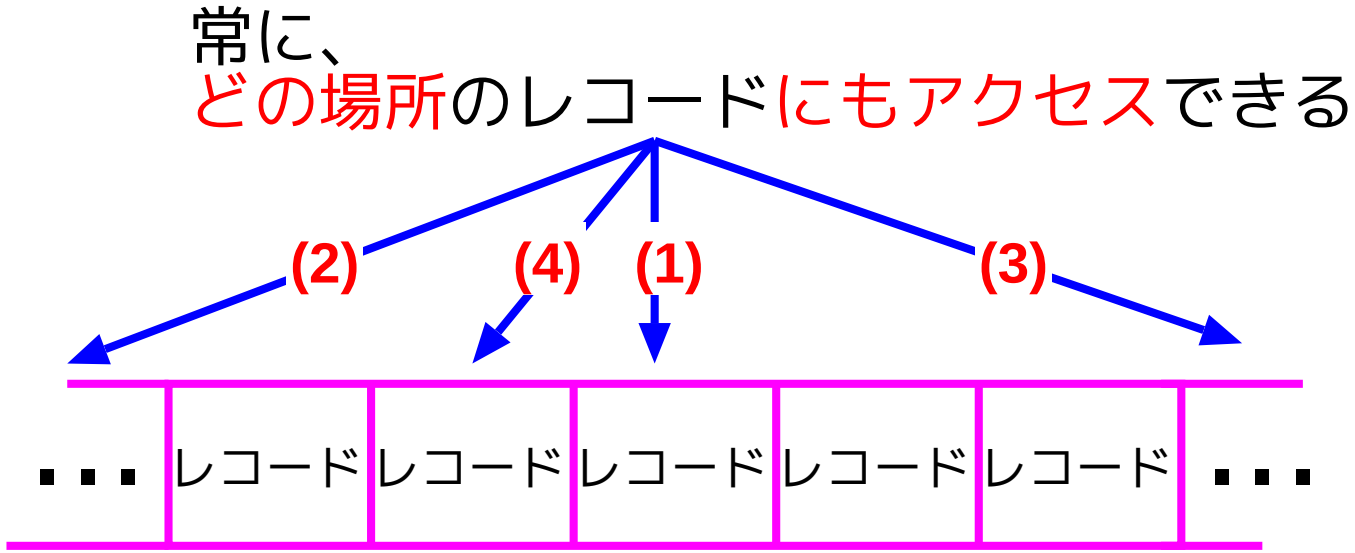
(⟹岩波情報科学辞典)

- 順編成 ··· ファイル中のレコードがファイル生成時に決まる順番に配置され、その順番でしかアクセスできないようなファイル編成法。

  IBM社のMVSでは順編成ファイルに対して SAM(Sequential Access Method)と呼ばれるアクセス法をプログラマに提供していた。



- 乱編成 ··· ファイル中のレコードをその識別子または属性値に基づいて順不同に呼び出せるようなファイル編成法の総称。



乱編成ファイルは、さらにレコードへのアクセス方法によって次のように分類される。

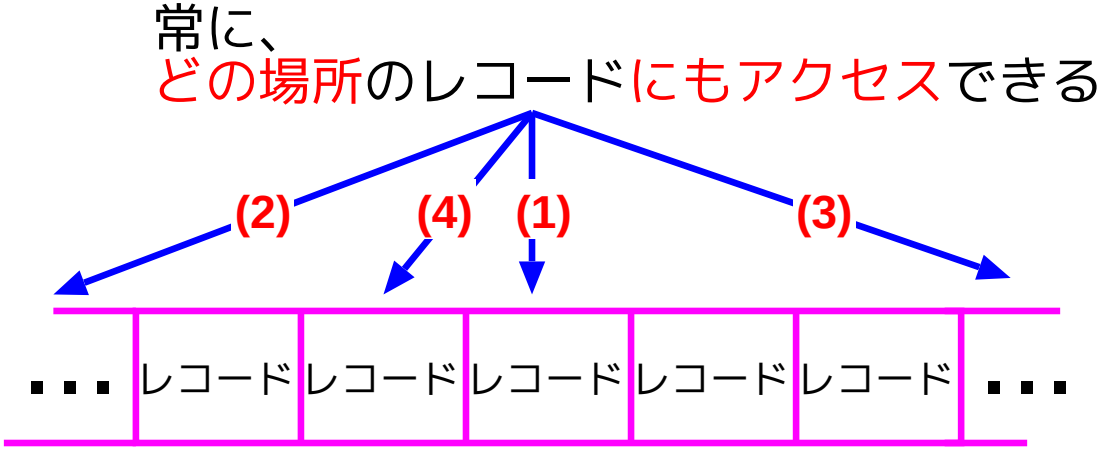
乱編成ファイルは、さらにレコードへのアクセス方法によって次のように分類される。

- 直接編成 · · · ファイル中のレコードのキー値に関数を作用させて補助記憶装置上の物理的な位置を決め、そこにレコードを格納するファイル編成法。

  IBM社のMVSでは直接編成ファイルに対してDAM**(Direct Access Method)と呼ばれるアクセス法をプログラマに提供している。**

- 索引編成 · · · ファイル中のレコード(の識別子)と補助記憶装置上の物理的な位置との対応表(索引)を用いてレコードへのアクセスを行うファイル編成法。

- ……



アクセスの仕方に関しては、

- 順アクセス,
- 乱アクセスまたは直接アクセス

の2つに分類されるだけである。

磁気テープ vs. 磁気ディスク：

磁気テープ装置の場合は順編成のファイルしか実現できない。
磁気ディスク装置だと順編成も乱編成も実現できる。

順編成の利点：

順編成だとファイルを構成するブロックの使用予測が容易なので、磁気ディスク装置上でも順編成の方がアクセス速度の点では好ましい。

例7. 4 (順編成ファイルの処理)

master flie
従業員の業務記録
(従業員番号順に並んでいる)

transaction file
残業、出張など、日々に発生する細かな業務データの束
(従業員番号順に並んでいる)

統合

新しい master flie
従業員の業務記録
(従業員番号順に並んでいる)

transaction :
An act of doing business.

batch 処理, 一括処理 :
(1) 蓄積したジョブをまとめて (一括して) 処理する時の技法。 (2) 一定期間内に溜ったデータをまとめて処理する時の技法。

例7. 5 (乱編成ファイルの処理)

オンラインシステムにおいて数多く見られる。

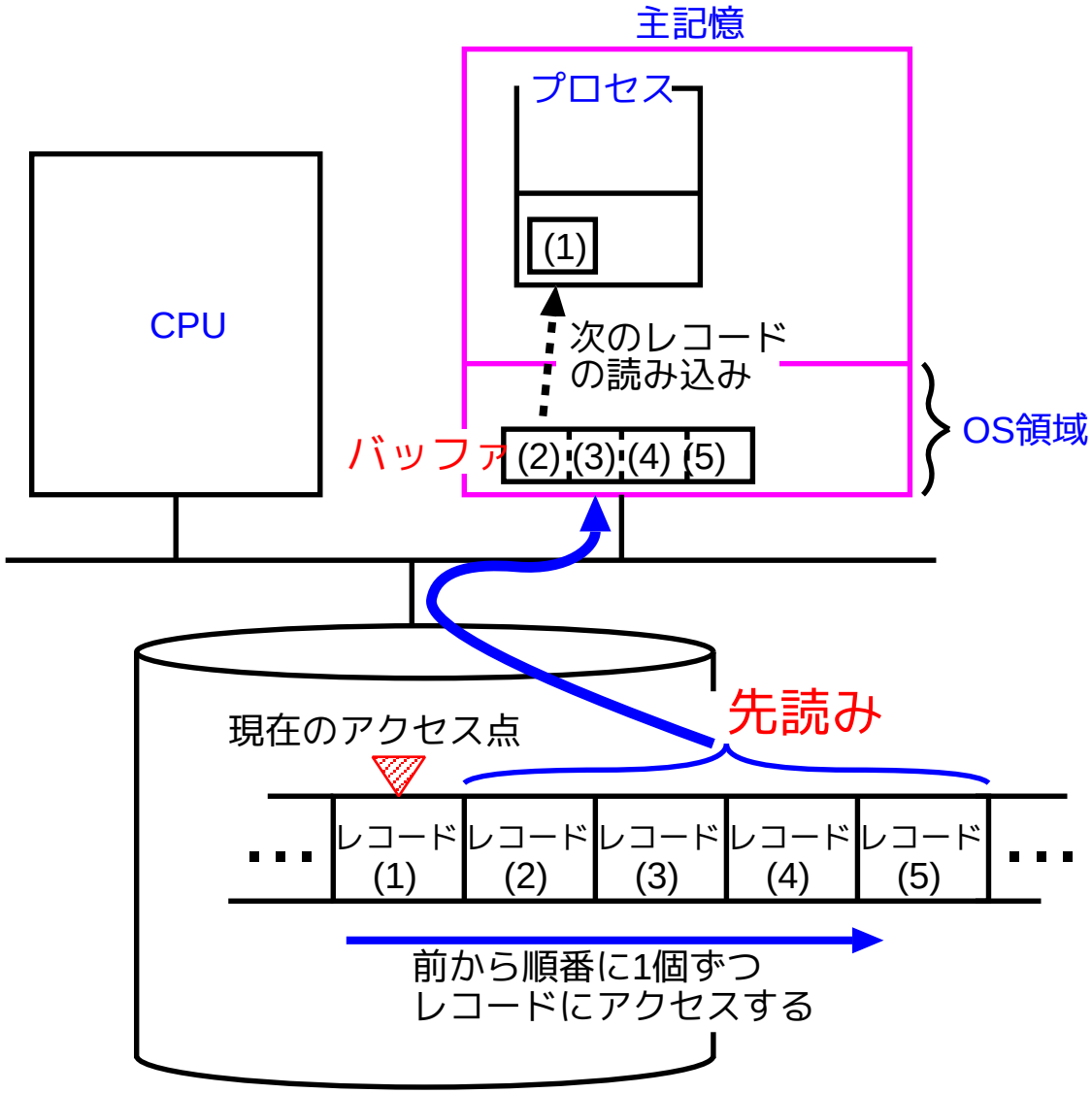
- 銀行の預金引出しの場合 : 口座番号 $\Longrightarrow$ 個人の預金情報のレコード
- 座席予約システムの場合 : 利用日+便番号 $\Longrightarrow$ 予約状況のレコード

順編成ファイルへのアクセスの高速化：

順編成ファイルでは、中のレコードが並んだ順にアクセスされ処理されることが決まっている。

⇨ 次に処理するブロックをバッファリング手法で次々と先読みするだけで、ファイルアクセスの高速化を実現できる。

- アクセススピードはシステムバッファの利用状況に依存する。
- ユーザの手を煩わせることなく、...



乱編成ファイルへのアクセスの高速化：

乱編成ファイルでは、ファイルを構成するどのブロックに対するアクセスが次に起こるかの予測は一般には全く出来ないので、バッファリングはアクセスの高速化にはあまり繋がらない。しかし、ランダムにアクセスされるとは言っても、ファイルを構成するブロックが一様にランダムにアクセスされることは滅多になく、多くの場合、一部のブロックにアクセスが集中する。

⇨ キャッシングが有効。
アクセスが集中するブロックをうまく認識してキャッシュ内に保持することが出来れば、ファイルへのアクセスを見かけ上高速に行えたことになる。

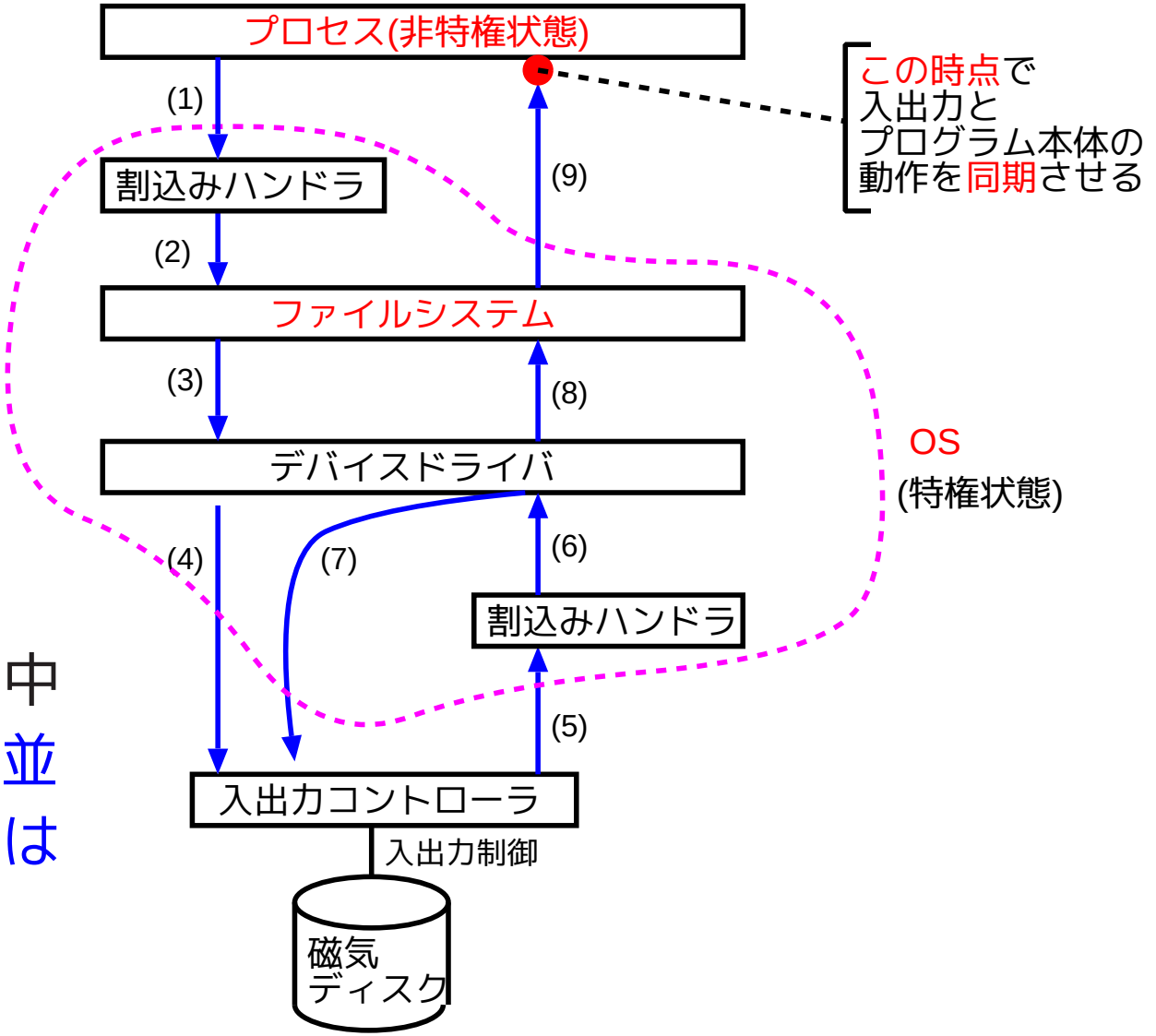
- 必要なキャッシュ容量は状況によって異なる。
- キャッシングの効果を高めたければ、特定の利用に限定した十分な解析が必要になる。

7–6 同期入出力 vs. 非同期入出力

同期入出力：

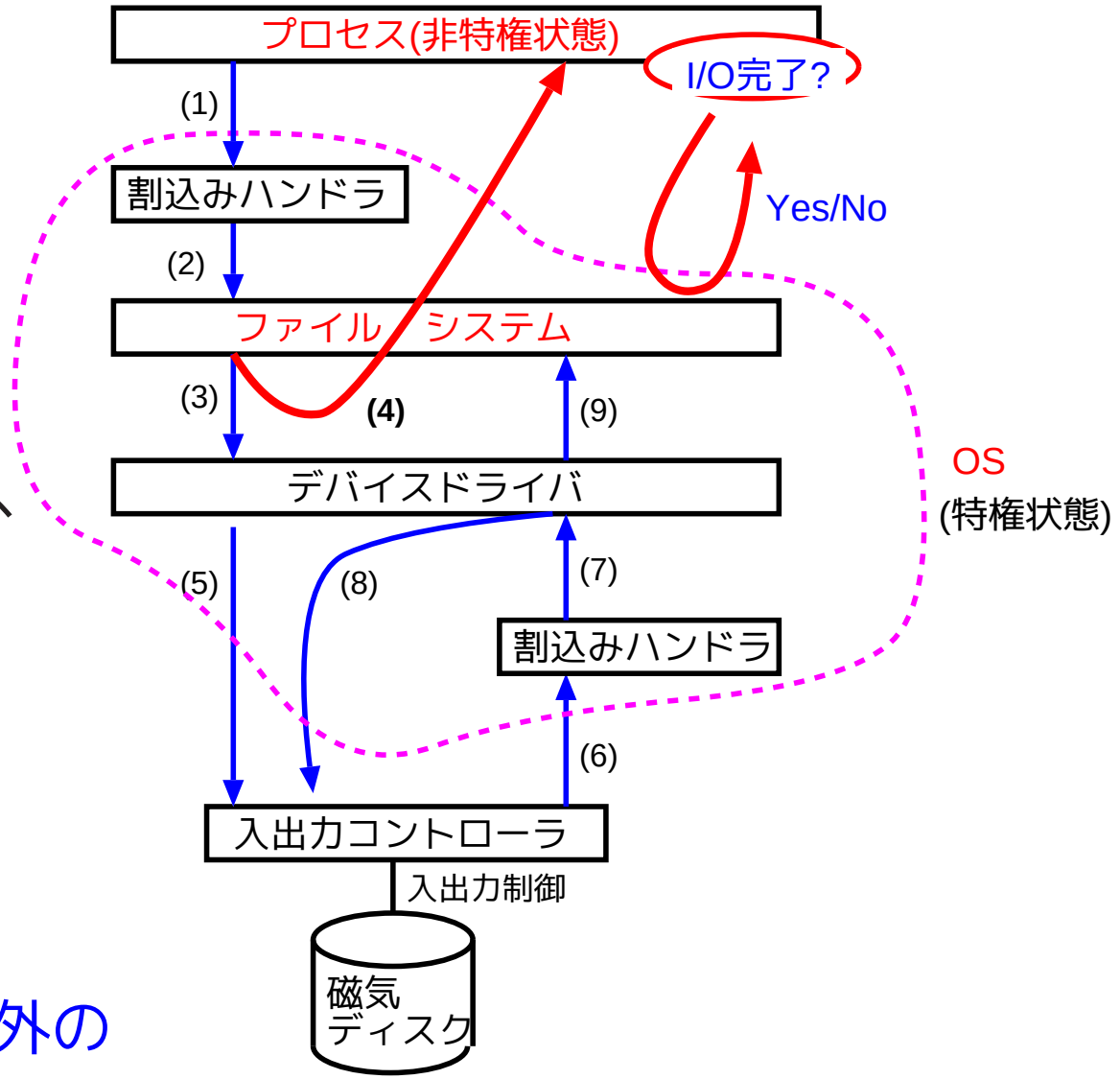
UNIXの下では、プログラムの中で入出力のシステムコールが発せられると、その入出力が完了するまでファイルシステムから元のプログラムに制御が戻ってこない。

⇨ ユーザプログラムの中で複数の入出力装置を並行して動作させることは出来ない。



非同期入出力：

入出力の完了が確認される前に元のプログラムに制御を戻すファイルシステムもある。



このようなシステムの下では、

- プログラム内の依存関係を保証するために入出力の完了を待つ (i.e. 最小限の同期を取る) ためのシステムコールが必要になるが、
- プログラムの中の入出力以外の部分と全ての入出力を並行して (非同期に) 動作させることが出来る。

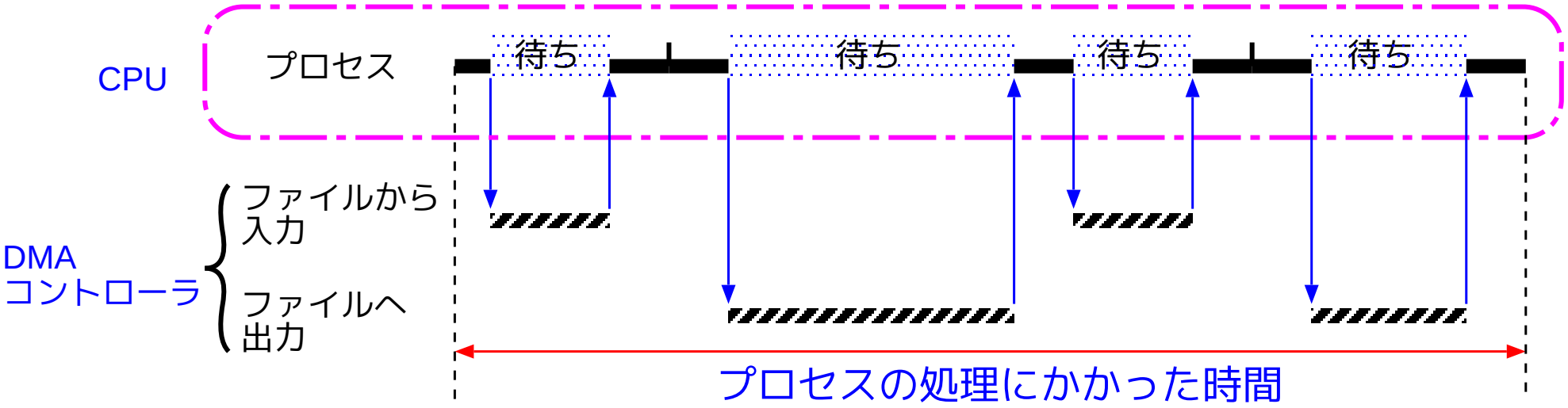
同期入出力vs.非同期入出力(まとめ):

	同期入出力	非同期入出力
簡単な説明	入出力が完了するまで入出力を呼んだ側に制御を戻さないことによって、入出力をプログラム本体の動作と同期させる。	入出力完了を確認する前に入出力を呼んだ側に制御を戻すことによって、入出力をプログラム本体と非同期に動作させる。
代表的なシステム	**UNIX**	**IBM社MVS**
利点	プログラマにとって分かり易い。 同期を明示的に取る面倒がない。	複数のファイル入出力を並行して行うことが出来る。
欠点	複数のファイル入出力を並行して行えない。 ⇨ ユーザプロセスのレベルではバッファリングは実現できない。	プログラマは入出力とプログラム本体との同期を明示的に取る必要がある。

	同期入出力	非同期入出力

2つのファイルを並行して処理したい場合の例	プログラム例：
	`while(1) {`
	`  .....`
	`  read(rfd, rbuf, rlen);`
	`  /* 読込みデータの更新 */`
	`  .....`
	`  write(wfd, wbuf, wlen);`
	`  .....`
	`}`
	⇨ 2つのファイルを交互に処理
	図を参照
		プログラム例：
		`while(1) {`
		`  .....`
		`  read(rfd, rbuf, rlen);`
		`  /* 読込みデータの更新 */`
		`  .....`
		`  rwait(rfd);`
		`  /* 読込み完了を待つ */`
		`  ...(読込みデータの処理)...`
		`  .....`
		`  wwait(wfd);`
		`  /*前回の書込み完了を待つ*/`
		`  write(wfd, wbuf, wlen);`
		`  .....`
		`}`
		⇨ 2つのファイルを並行処理
		図を参照

(同期入出力の場合)



(非同期入出力の場合)