

システムソフトウェア

2020年度

第10回 (11/9)

月曜7-8限・木曜7-8限(Zoom)

講義担当：渡部卓雄 (Takuo Watanabe)

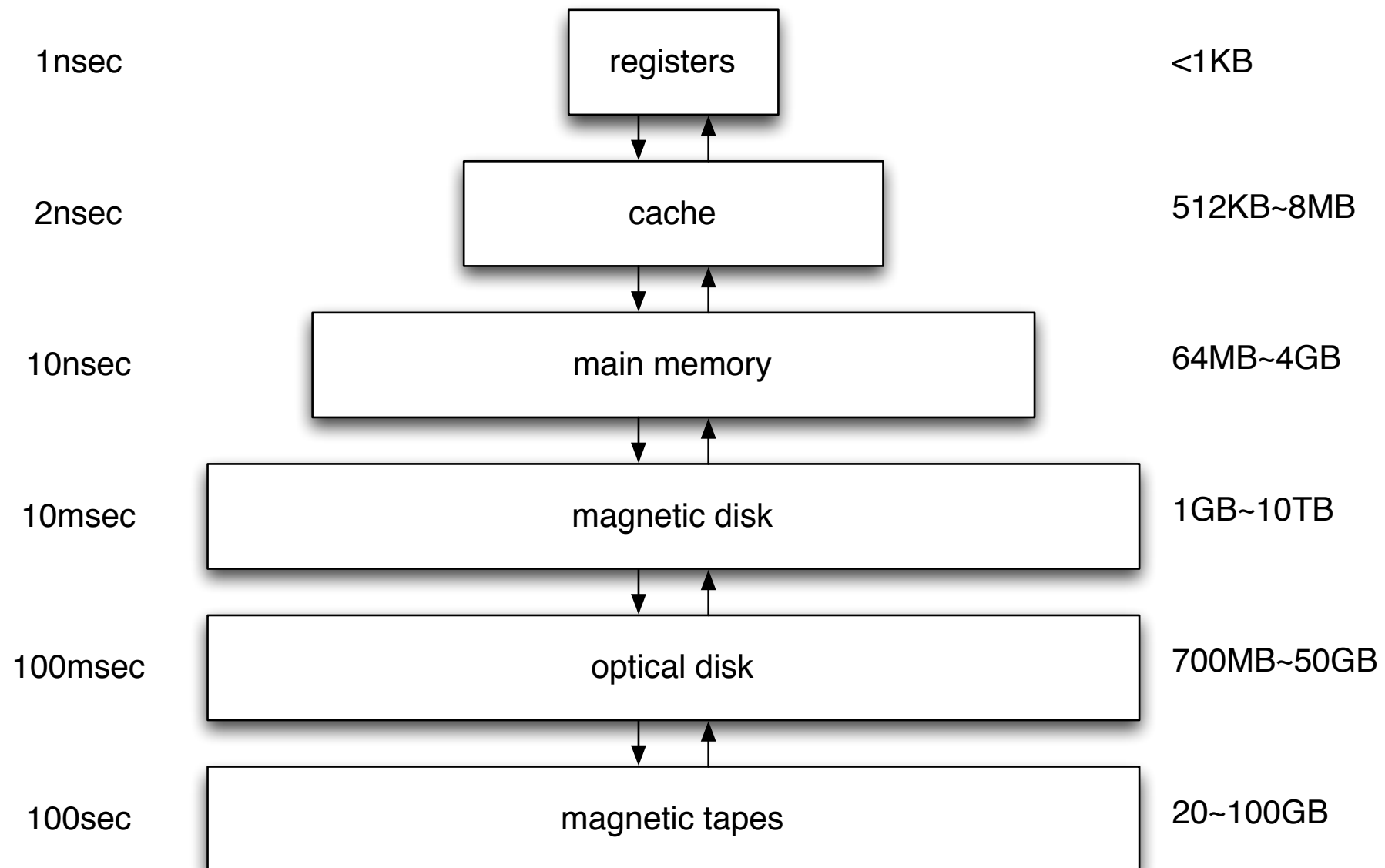
<http://titech-os.github.io>

e-mail: takuo@c.titech.ac.jp

本日のメニュー

- ファイルシステム(1)

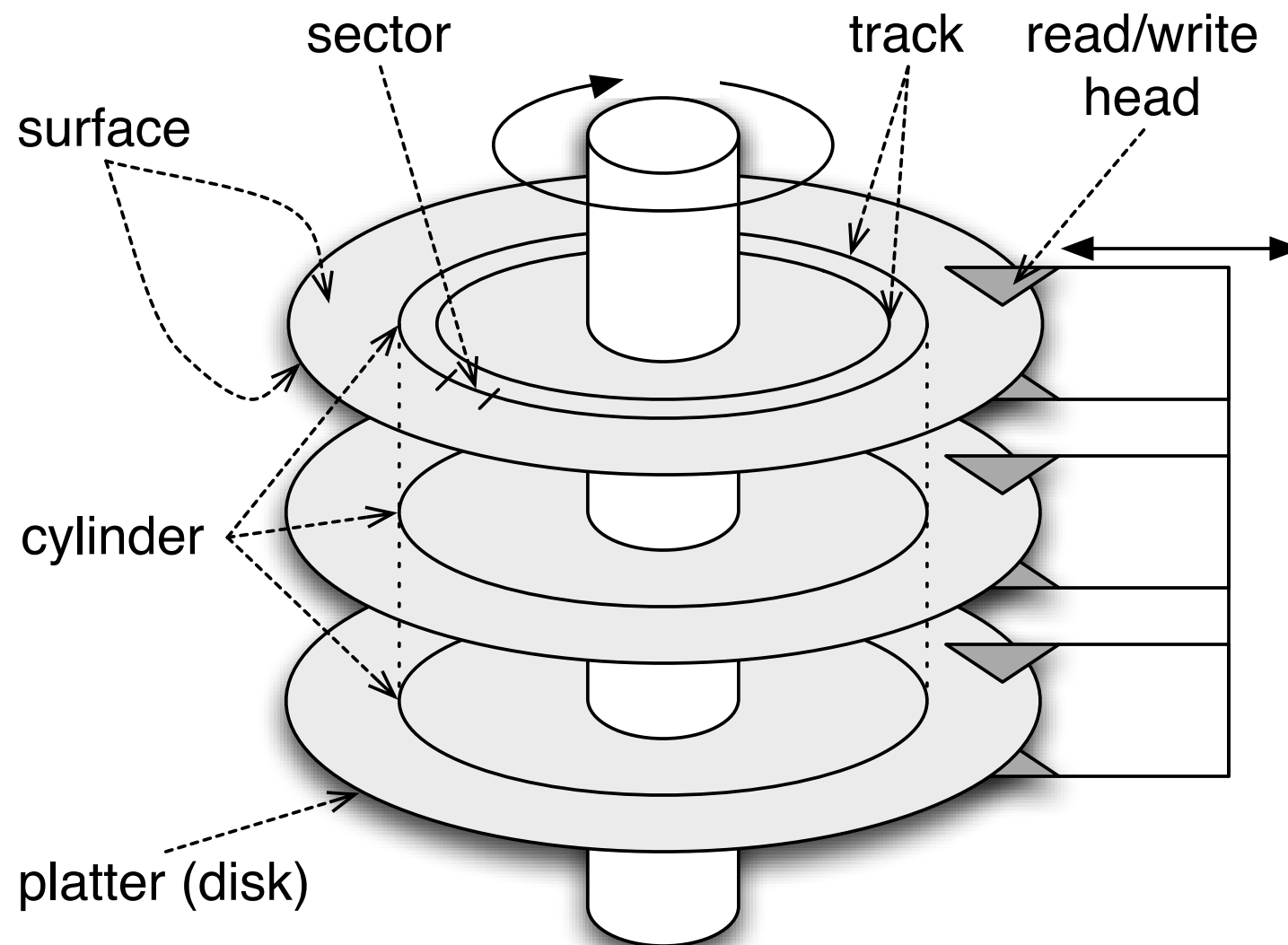
記憶装置の階層



二次記憶装置

- 大容量
 - － 主記憶に比べて大規模なデータを格納できる.
- 永続性(persistence)
 - － ディスクなどの磁気記憶装置やフラッシュメモリは、電源を切っても書き込んだデータは残る.
 - 主記憶に使われる半導体メモリ(DRAM等)は、普通は揮発(volatile)なものが多い.
 - － 特殊な機器では、SRAMにバッテリバックアップを用いて主記憶を不揮発にすることがある.
- (主記憶と比べて) 遅い

ハードディスク



ハードディスクの入出力

- セクタ(sector)と呼ばれるトラック上の塊の単位で読み書きを行う
 - 1セクタ=512B~4KB (典型例)
 - 目的のセクタにアクセスするためには、ヘッドを適切なシリンダまで動かし(シーク)、セクタが来るまで待つ(回転待ち).
- 性能例
 - 平均シーク時間 10ms
 - 平均回転待ち時間 4ms (7200rpm)
 - 転送速度 40MB/sec

ファイルシステム

- 二次記憶(ディスク)を簡単に使いたい
 - － ハードディスクの低レベルな入出力は不便
- ディスク以外の機器の入出力も統一的に扱いたい.
 - － スペシャルファイル(コンソール等)
- システムの各種情報もファイルとして扱えるとプログラミングが楽
 - － プロセスファイルシステム等

ファイル

- データを格納するための論理的な記憶装置で、多くの場合二次記憶の最小の割当単位
- ファイルに格納されるもの
 - － プログラム
 - － データ
 - 数値, 文字, その他任意のバイト列

ファイルの構造

- 種類
 - － 構造なし（単なるバイト列）
 - － 単純なレコード構造
 - 行，固定長レコード，可変長レコード
 - － 複雑な構造
 - 構造付文書，実行可能ファイル
 - リソースフォーク
- 構造を規定するもの
 - － OS
 - － アプリケーションプログラム

ファイルの型

- ファイルの型
 - － 実行可能形式, オブジェクトコード, ライブラリ
ソースコード, スクリプト, テキスト, アーカイ
ブ, マルチメディア, その他特定のアプリケーション
用ファイル
- 型の指定
 - － ファイル名 (多くの場合, 拡張子)
 - － マジックナンバー
 - － その他のファイル属性 (メタデータ)

ファイルの属性

- 名前
 - － ユーザが区別するための名前（文字列）
- 識別子
 - － ファイルシステム内で一意な番号
- 型
 - － ファイルの種類
- 位置
 - － ファイルの実体へのポインタ
- サイズ
 - － ファイルの大きさ
- アクセス制御情報
- 時刻
 - － 作成, 修正, 読み出し時刻
- ユーザ識別子
 - － ファイルの所有者, 利用可能者

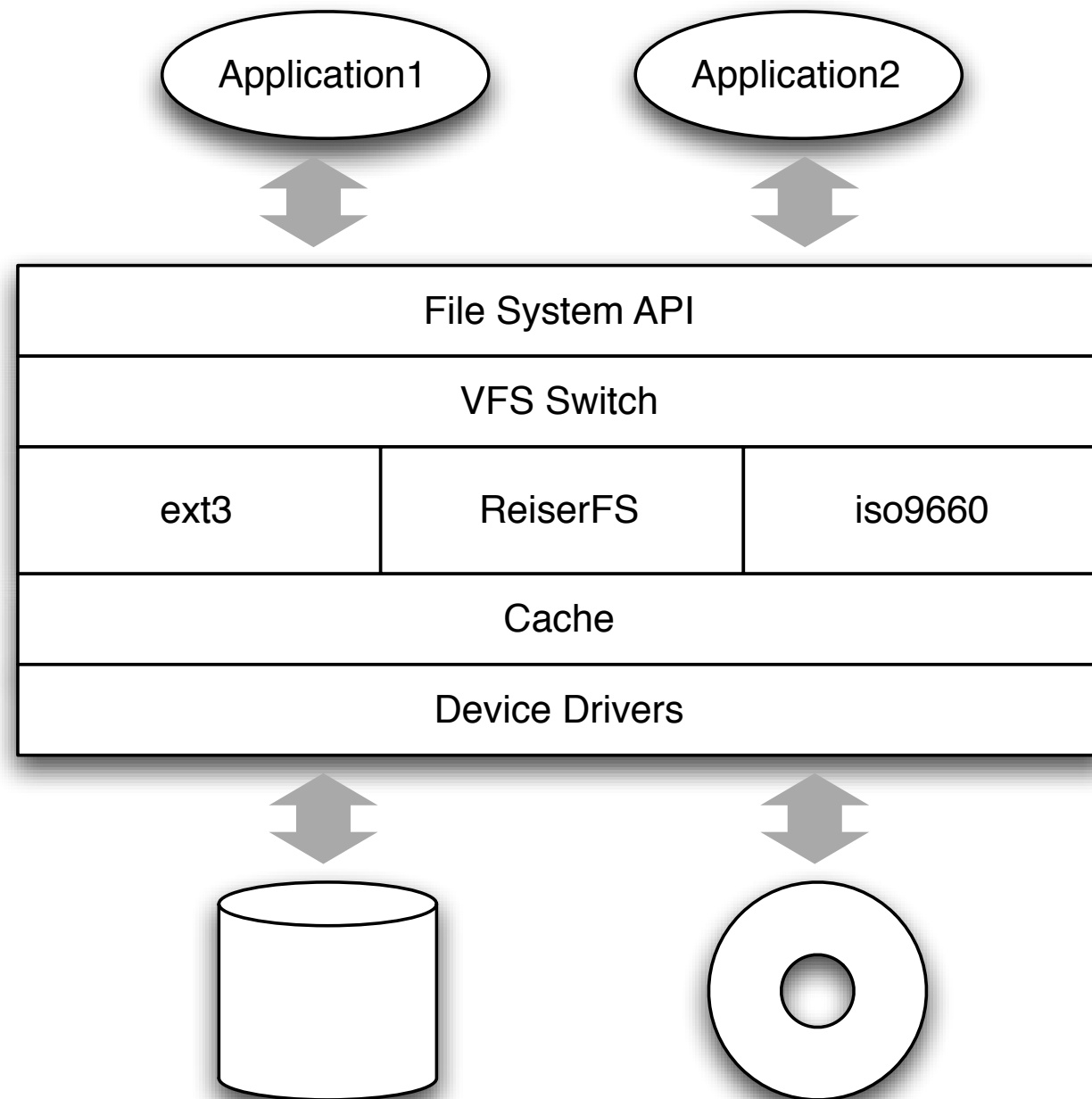
ファイルシステムAPI

- ファイルは以下のような操作を提供する抽象データ型(abstract data type)とみなせる.
 - 作成 (create)
 - 書き込み (write)
 - 読み出し (read)
 - 読み書き位置の変更 (seek)
 - 削除 (delete)
 - 切り詰め (truncate)
 - 追加 (append)
 - 名前替え (rename)

APIとファイルシステムの実装

- 多くのOSではファイルシステムのAPIを共通化し、様々な実装に対応できるようにしている
 - － 様々な記憶メディアへの対応
 - HDD, フラッシュメモリ, 光学ドライブ
 - メモリ
 - － ファイルシステムAPIを利用したシステムインターフェースの提供
 - デバイスファイル (Unixの/dev)
 - プロセスファイル (Linuxの/proc)
 - ネットワーク (Plan9の/net)

Linuxのファイルシステム



ファイル割当て

- ディスクはセクタの巨大な配列と見なすことができる.
- ファイルシステムは, これをうまく分割して, 多くのファイルに見せる仕組みである.
- ファイル割当て方式
 - 連続
 - 連結リスト
 - FAT
 - i-node

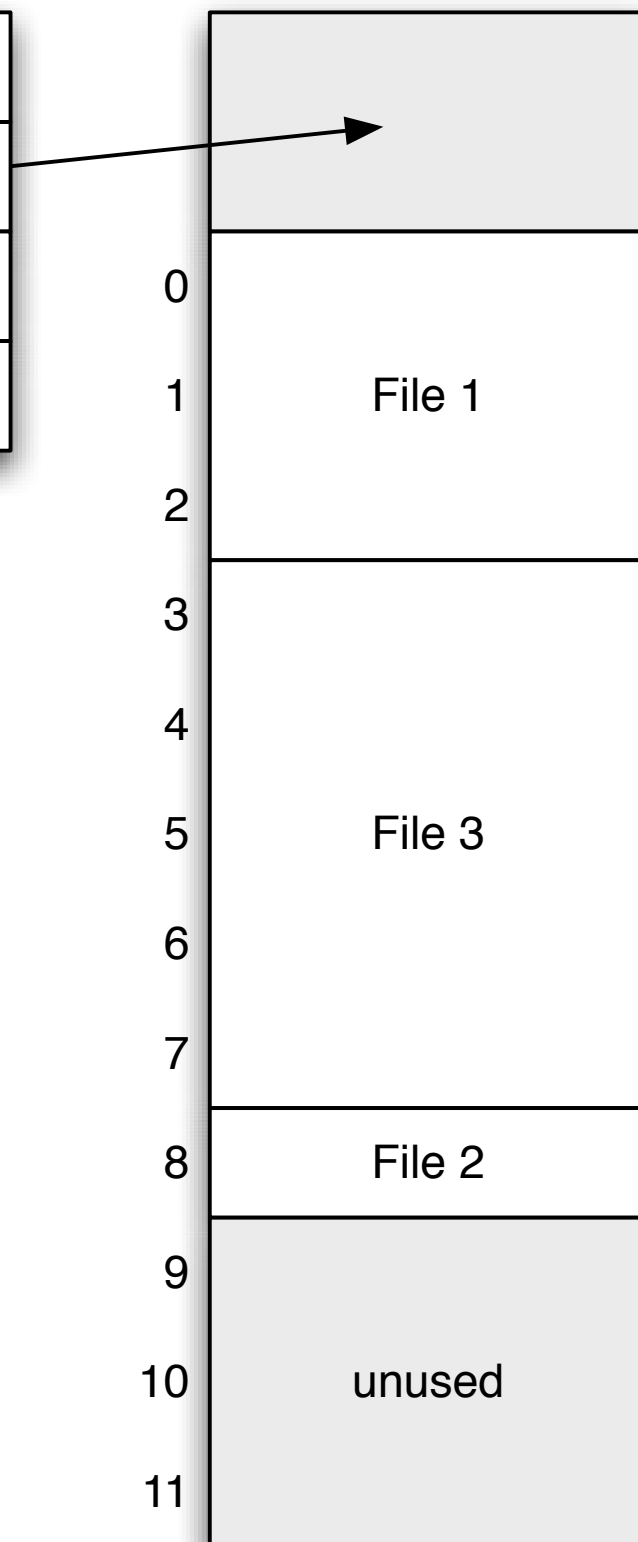
ブロック

- 1～数個のセクタをまとめたもので、ファイルを構成する最小単位
 - － 例
 - ext2/ext3 (Linux): 1024/2048/4096バイト
 - HFS+ (Mac OS X): 512/1024/2048/4096バイト
 - － 1Gバイトより大きなディスクボリュームでは4096バイト
 - － 1バイトのファイルでもディスク上の1ブロックを消費する.
- 各ブロックには番号がつけられ、それによって管理される.

連続割当て (contiguous allocation)

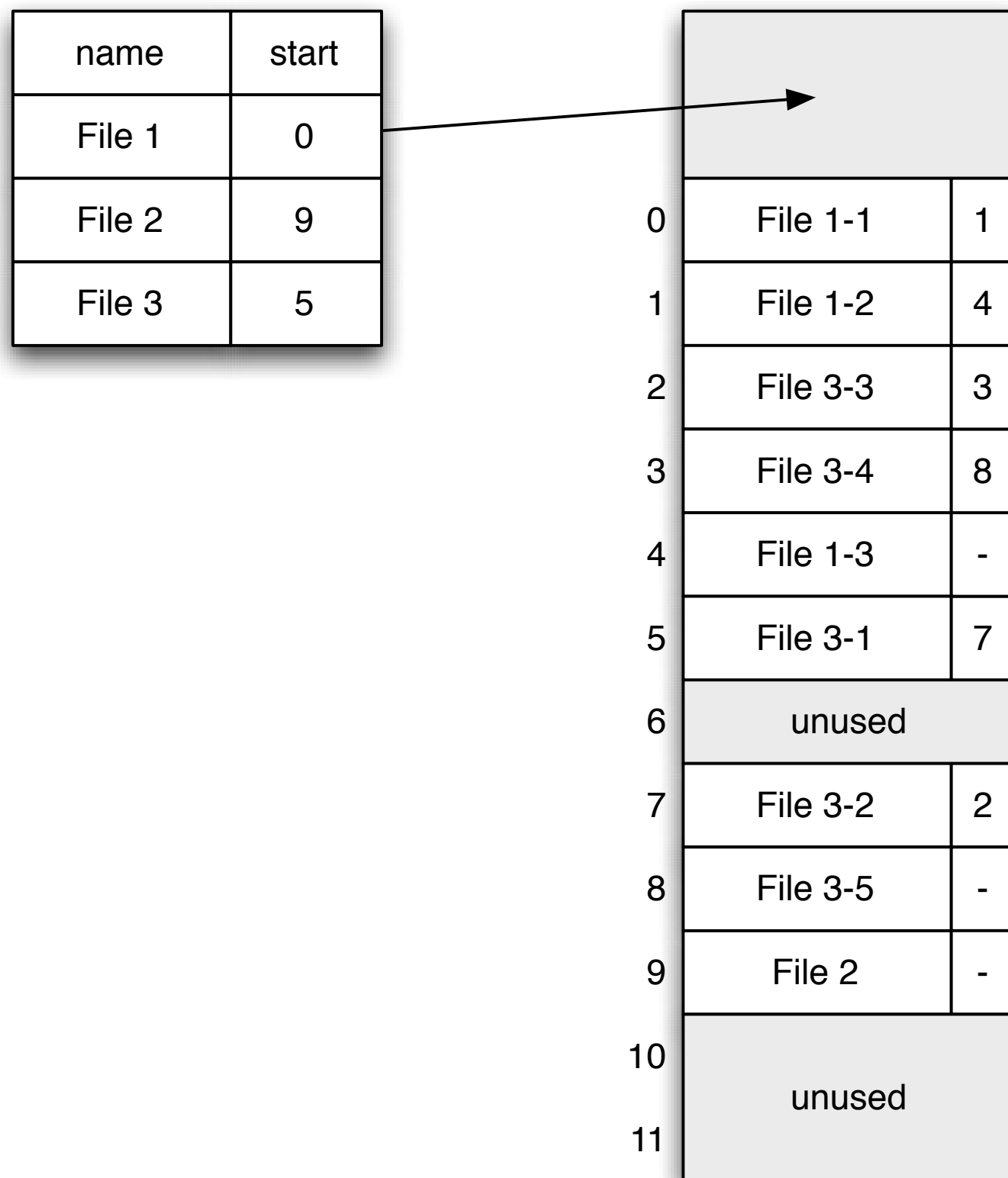
- 連続したブロックをファイルに割り当てる.
 - ブロック：いくつかのセクタからなる固定長の領域で、OSはブロック単位で入出力を行う.
- pros
 - 実装が単純
- cons
 - ファイルサイズの変更が難しい.
 - 断片化が生じる.

name	start	end
File 1	0	2
File 2	8	8
File 3	3	7



連結リストによるファイル割当て (linked allocation)

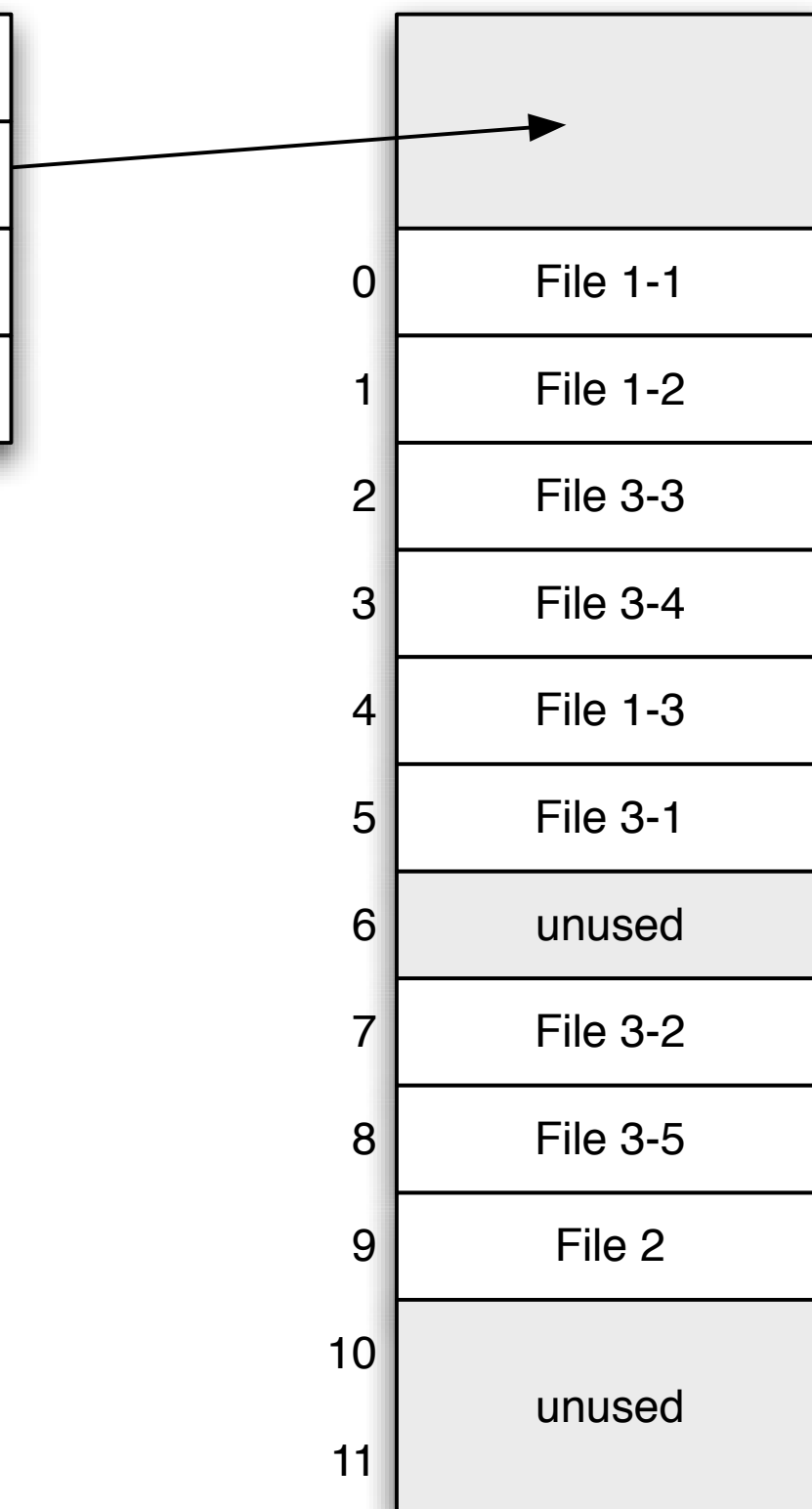
- 連結リストの形でつながった一連のブロックをファイルに割り当てる.
- pros
 - ブロックは連続した領域にある必要がないため、ファイルサイズの変更が容易になり、断片化も生じない.
- cons
 - リンクをたどるため直接アクセスが遅い.
 - ポインタ(ブロック番号)用の半端な領域が必要.



FAT(File Allocation Table)

- 連結リストのリンク情報のみを独立に管理.
- 実行効率をあげるため, リンク情報(FAT)をメモリ上にも置く.
 - FAT16, FAT32 (DOS/Windows)
- pros
 - 直接アクセスも高速になる.
 - 各ブロックにリンクのための領域がいらぬい.
- cons
 - FATがメモリを消費する.

name	start
File 1	0
File 2	9
File 3	5



FAT	
0	1
1	4
2	3
3	8
4	-
5	7
6	
7	2
8	-
9	-
10	
11	

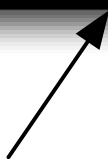
インデックスによる割り当て (indexed allocation)

- ファイルを構成するブロックの番号の列(インデックステーブル)を独立に管理する.
- pros
 - ランダムアクセスに有利.
 - 使用する分のみメモリにキャッシュできる.
- cons
 - インデックステーブルのためのブロックが必要
 - 実装が若干複雑になる.
 - ファイルサイズが大きくなった場合に間接参照が必要となる.

i-node

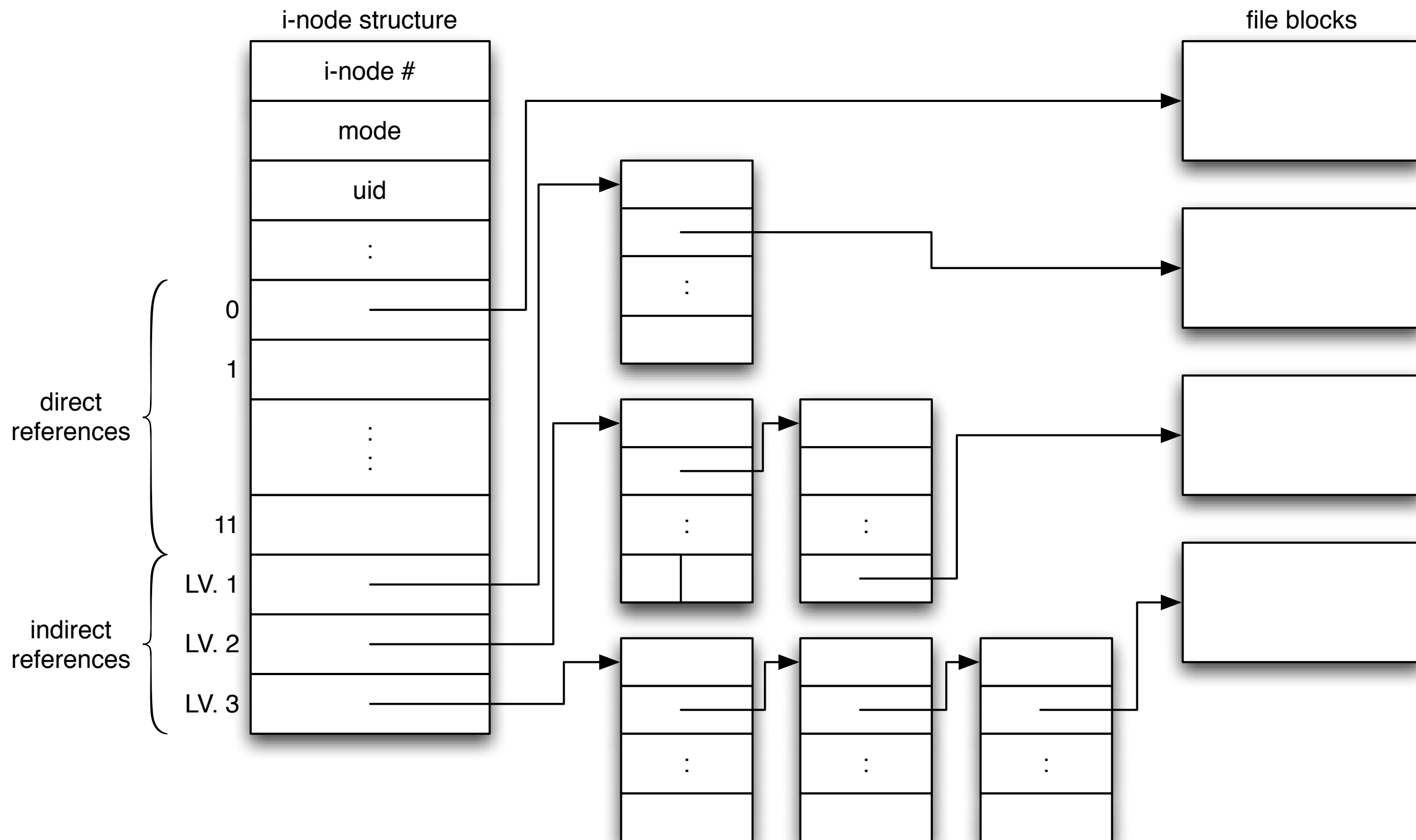
- インデックステーブルのエントリ
 - － 使用するブロック(を指すポインタ)の列
 - 通常エントリサイズは固定なので、大きなファイルを扱うときは間接参照(後述)を用いる.
 - － 他に、ファイルタイプ、モード、所有者、グループ、アクセス・変更日時等の情報を持つ.
- i-nodeとファイル(ディレクトリ)が対応
- i-node番号
 - － 各i-nodeが持つ固有の番号

name	used blocks
File 1	0, 1
File 2	9
File 3	5, 7, 2, 3, 8

i-node


0	File 1-1
1	File 1-2
2	File 3-3
3	File 3-4
4	File 1-3
5	File 3-1
6	unused
7	File 3-2
8	File 3-5
9	File 2
10	unused
11	

UFS(Unix File System)の 組み合わせ型多重レベルインデックス



最大ファイルサイズの例

- ブロックサイズ: 4KB
- ポインタサイズ: 32bit
- 直接参照ブロック数: 12
 - 直接参照: $12 \times 4\text{KB} = 48\text{KB}$
- 間接参照ブロックレベル: 3
 - 間接ブロック内のポインタ数: $4096/4 = 1024$
 - 第1間接参照(LV. 1): $1024 \times 4\text{KB} = 4\text{MB}$
 - 第2間接参照(LV. 2): $1024^2 \times 4\text{KB} = 4\text{GB}$
 - 第3間接参照(LV. 3): $1024^3 \times 4\text{KB} = 4\text{TB}$
- 合計: $48\text{KB} + 4\text{MB} + 4\text{GB} + 4\text{TB}$

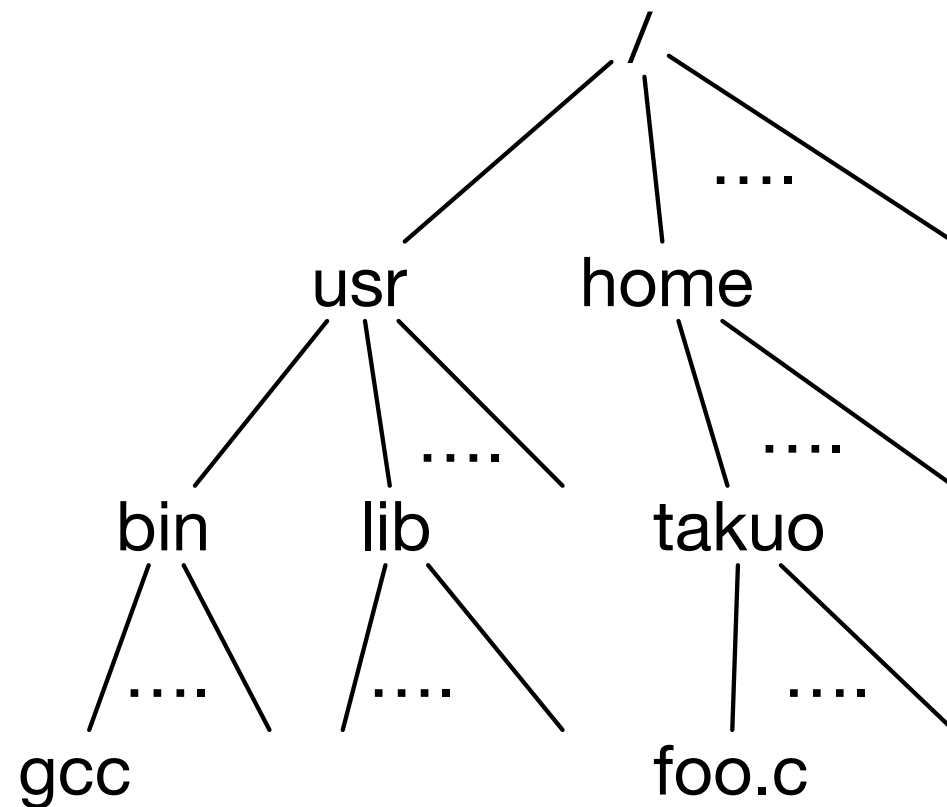
インデックステーブルの管理

- ブロックアルゴリズム
 - i-nodeを管理するブロック(i-nodeブロック)を線形探索する. ファイル数が増えると遅い.
 - 例: ext2, ext3
- 木構造を使った管理アルゴリズム
 - B-木とその派生アルゴリズムが多く用いられる.
 - 例: JFS, XFS, NTFS (B+-tree), ReiserFS, HFS+(B*-tree)

空きブロックの管理

- ビットベクタ(ビットマップ)による方法
 - － n個のブロックをnビットのベクタで管理する.
 - $\text{bit}[i] = 0 \Rightarrow$ 未使用
 - $\text{bit}[i] = 1 \Rightarrow$ 使用済み
 - － 連続したブロックを確保しやすい.
 - － ビットベクタのための領域が必要.
- フリーリストによる方法
 - － 空きブロックを連結リストとして管理する.
 - － 余分な領域は不要.
 - － 連続したブロックが欲しいときに面倒.

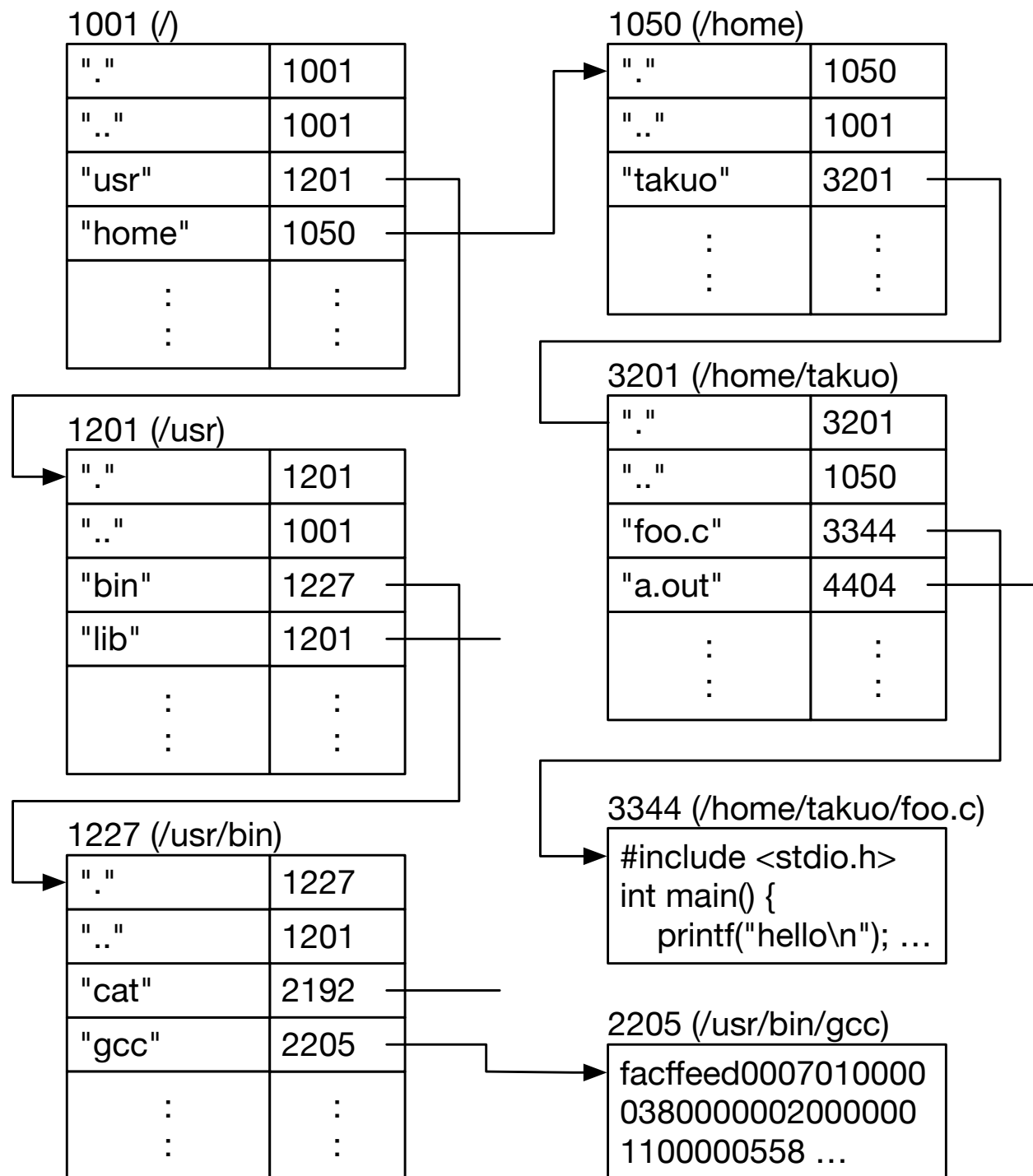
ディレクトリ



- ファイルには名前を付けることができる.
 - i-node番号で表すのは不便
- 名前空間はディレクトリによって階層化される.
 - 多くのファイルを平坦な名前空間で管理するのは大変.

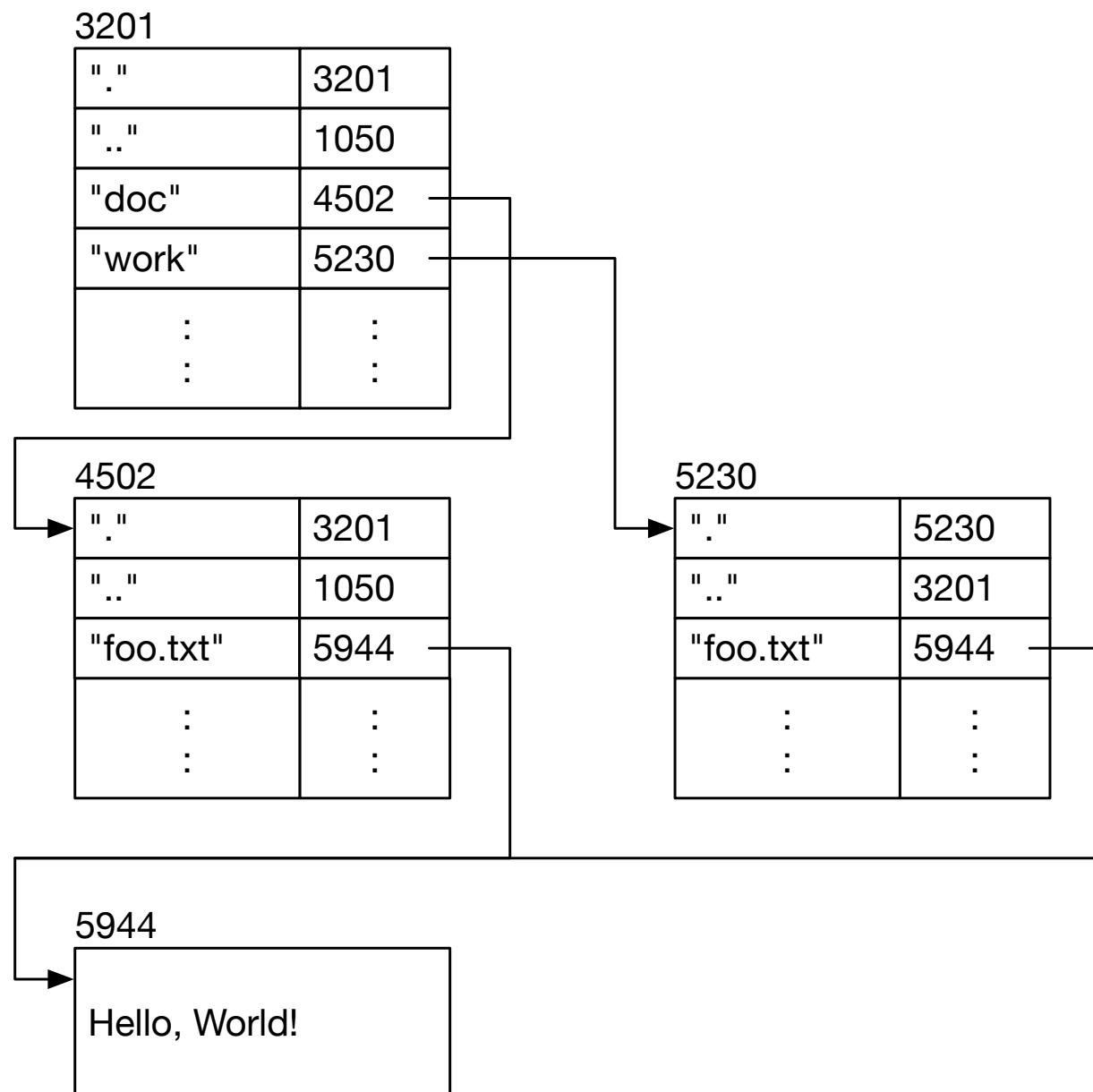
- ルートディレクトリ
 - ディレクトリ階層の最上位に位置するディレクトリ
- パス（ファイルパス）
 - 絶対パス：ファイルをルートディレクトリからの経路で表す
 - 例：/usr/bin/gcc, /home/takuo/foo.c
 - 相対パス：あるディレクトリを起点とした相対位置でファイルを表す
 - 例：../lib/libc.a
 - /usr/binを起点として/usr/lib/libc.aを表した場合
- カレントディレクトリ
 - プロセスが現在着目しているディレクトリ
 - システムコールはカレントディレクトリからの相対パス（あるいは絶対パス）でファイルを指定する.

Unixのディレクトリの構造



- ディレクトリは、ファイル名とi-node番号の組の配列を内容とするファイル
 - ファイル名はディレクトリに格納されることに注意
- 各ディレクトリは自身(.)と親ディレクトリへの参照(..)をもつ。

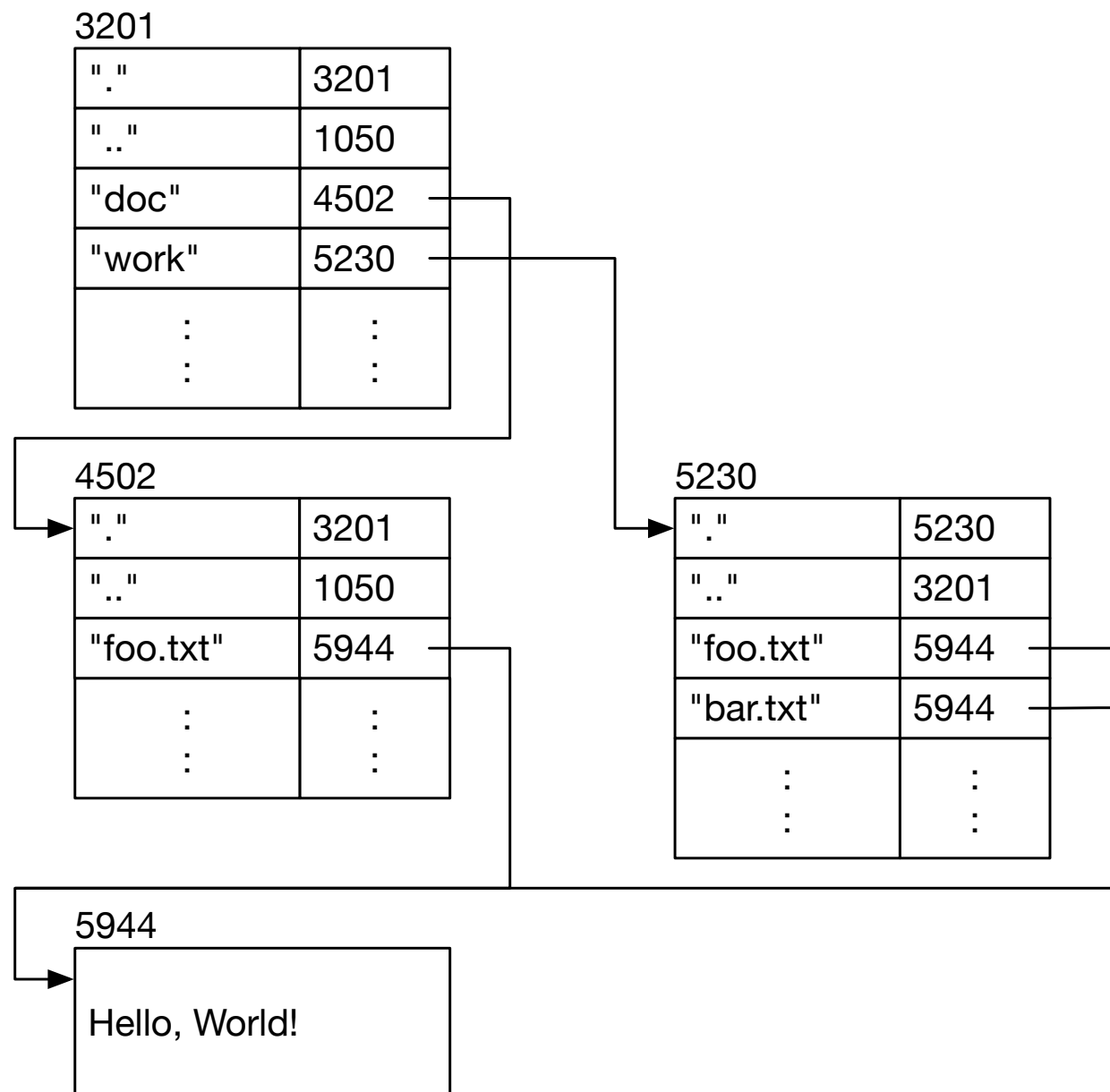
リンク（ハードリンク）(1)



```
$ pwd
/home/takuo/doc
$ ls
foo.txt
$ cd ..
$ mkdir work
$ cd work
$ ln ../doc/foo.txt .
$ ls
foo.txt
```

上記を実行するとdocとworkにそれぞれfoo.txtというファイルが作られたように見えるが、その実体は一つである。

リンク（ハードリンク）(2)



```
$ pwd
/home/takuo/work
$ ls
foo.txt
$ ln foo.txt bar.txt
$ ls
bar.txt  foo.txt
```

上記を実行するとworkにfoo.txtとbar.txtという2つのファイルができたように見えるが、これらの実体は同一である。

ファイルの削除

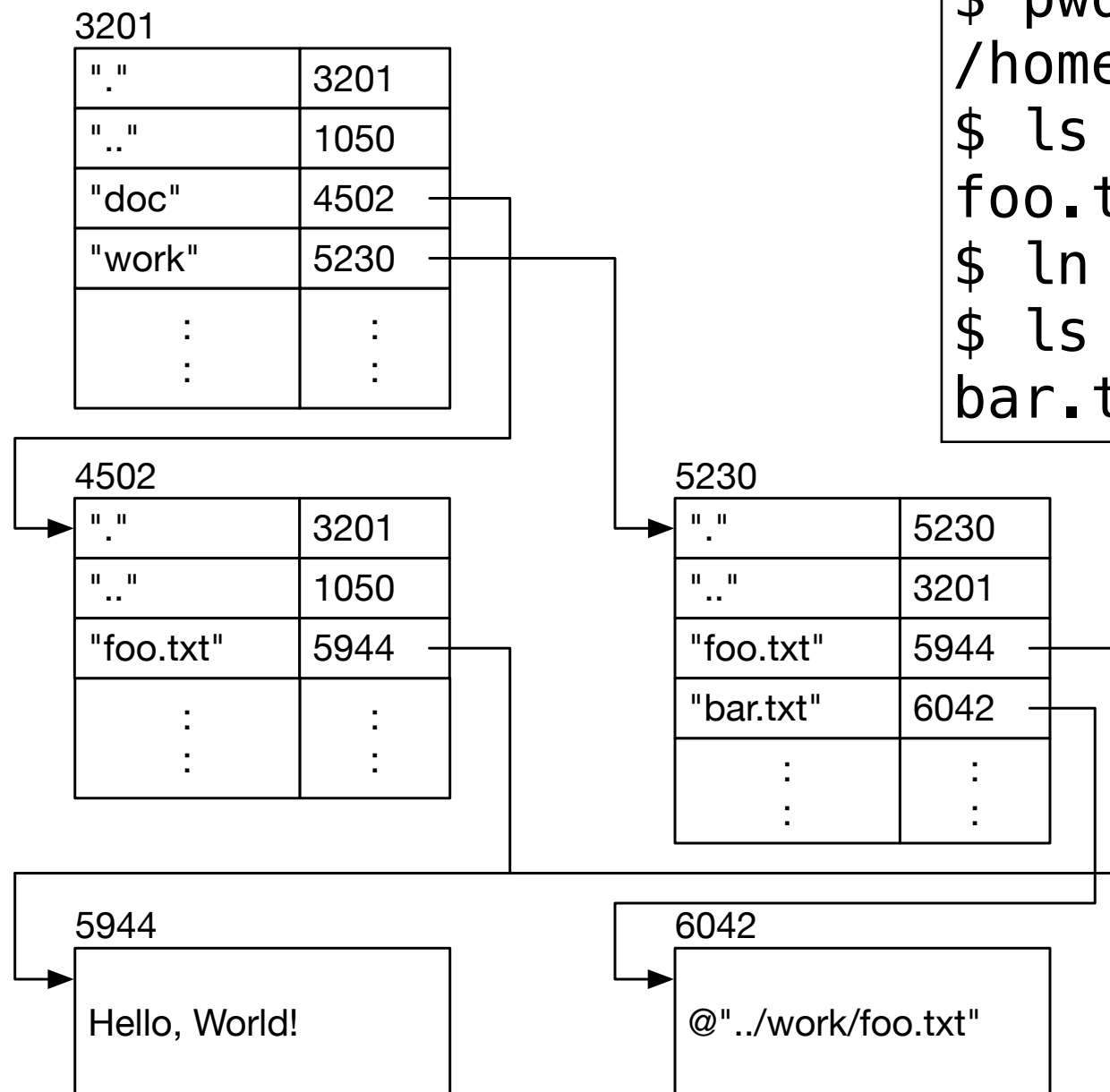
- ファイルの実体(inode)は複数箇所（パス）から参照されることがあり得る。
 - － inodeは参照数を管理するためのカウンタ（リファレンスカウンタ）を持っている。
- unlinkシステムコールは指定されたパスのファイルを削除するが、その際にまずリファレンスカウンタを1減らし、0にならない限りファイルの実体(inode)を削除しないようにしている。

ファイルの削除

スライド p. 34 の状態より

```
$ cd ..
$ pwd
/home/takuo
$ ls doc
foo.txt
$ ls work
bar.txt  foo.txt
$ rm work/bar.txt      # リファレンスカウンタ: 3 -> 2
$ ls work
foo.txt
$ rm doc/foo.txt      # リファレンスカウンタ: 2 -> 1
$ ls work
foo.txt
$ rm work/foo.txt     # リファレンスカウンタ: 1 -> 0, 実体を削除
```

シンボリックリンク



```
$ pwd
/home/takuo/work
$ ls
foo.txt
$ ln -s ../doc/foo.txt bar.txt
$ ls
bar.txt  foo.txt
```

上記を実行して作られた bar.txtは "../doc/foo.txt" という相対パスを含む特殊なファイルである. このファイルに対する削除以外の操作は ../doc/foo.txt に対する操作とみなされる.

ディレクトリのマウント

- 複数のファイルシステムを一つにまとめ、位置透過なディレクトリ空間を提供すること.
 - ー ファイルシステムAの中のファイル(マウントポイントとよばれる)pに、他のファイルシステムBのルートを重ね合わせ、pから下があたかもAの一部であるかのように見せることができる.

バッファキャッシュ

- ファイル(ブロック)の内容をメモリ上にキャッシュし, I/O速度を向上させる.
- 方式
 - ライトスルー(write through)
 - キャッシュとディスク両方に書き込む.
 - ライトバック(write back)
 - ディスクへの書き込みを遅延させる.
- バッファ不足時
 - バッファの内容をディスクに書き戻して空き領域を確保. LRUで書き戻すバッファを決める.

バッファ書き出し

- 単純なLRUのみではまずいことがある.
 - システムがクラッシュしたときにファイルシステムの一貫性が失われると困るので、データを書き戻さずにキャッシュに長期間おくのは避ける.
 - メタデータ (i-node等) はライトスルーとする.
 - syncシステムコール

統合バッファキャッシュ(Unified -)

- read/writeによる入出力と, mmapによるI/Oによる入出力を比べてみる.
 - read/write
 - バッファキャッシュを介して読み書きを行う
 - mmapによるI/O
 - ページの内容をメモリに読み込む(ページキャッシュ)が, その際バッファキャッシュを介する. このような状況をダブルキャッシングという.
- 統合バッファキャッシング
 - バッファキャッシュをページキャッシュとして用いることでダブルキャッシングを防ぎ, 性能と安全性を向上させる.

fsck

- ファイルシステムの整合性検査と修復を行う.
 - 各ブロックはi-nodeかフリーリストから一度だけ参照されていない.
 - ファイルの参照数はi-node内の参照数と一致していない.
 - ディレクトリ . は自分自身を, .. は親ディレクトリを参照していない.
 - etc.

ジャーナリング(Journaling)

- ファイル操作は、一般に複数のメタデータを修正する。修正中にOSがクラッシュすると、ファイルシステムの一貫性が失われる。
- ジャーナリング(ロギング)
 - ー ブロックに書き込む前に、メタデータの変更要求列をディスク上のジャーナルと呼ばれる場所に記録(ログ)する。ブロックへの書き込みが終了したときに、完了マークをジャーナルに記録する。
 - ー リブート時に完了マークがないログは再実行し、ファイルシステムの一貫性を保つ。

まとめ

- ファイルシステム(1)
 - － 記憶階層
 - － ファイル, ファイルシステムAPI
 - － ファイル割当て
 - 連続, 連結リスト, FAT, i-node
 - － ディレクトリ, バッファキャッシュ
 - － ジャーナリング