ディレクトリのx権限は「読む」ことか

～*Unix v6*に基づく検索と参照の技術的再検討～

要約

UNIXファイルシステムにおけるアクセス制御は、i-nodeを単位とした`rwx`パーミッションに基づいて動作するが、ディレクトリに対する`x`（検索）権限と`r`（読み取り）権限の意味的・構造的差異は直感に反し、誤解されやすい。本稿では、Unix v6のカーネル実装を前提に、`namei()`によるパス解決処理の実装、`iget()`や`readi()`を用いたブロックアクセスの動作を検証し、`x`のみで可能な検索行為と、`r`が必要な列挙操作とを厳密に分離する。最終的には、両者の分離がファイルシステムにおける最小権限設計やアクセス責務の明確化にどのように寄与しているかを構造的に総括する。

第1章　問題の所在──「xで読める」とはどういうことか

1.1 ファイルとディレクトリにおけるrwx権限の定義

UNIXファイルシステムにおいて、すべてのファイルおよびディレクトリはi-nodeによって管理され、それぞれに読み取り（r）、書き込み（w）、実行（x）の3種のアクセス権限が付与される。ファイルにおけるrwxの意味は比較的直感的であり、rは内容の読み取り、wは内容の変更、xはプログラムの実行を意味する。これに対して、ディレクトリに対するrwxの意味はやや抽象的である。rはディレクトリの内容（すなわち、含まれるファイル名とi-node番号の一覧）の読み出しを、wはディレクトリ構造の変更（ファイルの追加・削除など）を、xは「検索」もしくは「通過」を意味し、特定の名前に対してi-nodeに到達することを許可する。

1.2 標準的な誤解と教科書的説明の限界

多くのUNIX解説書や入門教材では、rとxの違いについて十分な説明がなされていない。たとえば、「x権限があればディレクトリを『読める』」という表現がしばしば用いられるが、これは誤解を招くものである。実際には、r権限がなければlsコマンドによるファイル一覧の表示は不可能であり、x権限だけではディレクトリの中身を「知る」ことはできない。一方で、x権限があれば、たとえr権限がなくても、指定された名前をもとに内部のi-nodeへ到達し、そのファイルを開くことが可能である。この違いは、アクセス制御やファイルシステム設計における根本的な構造的意義を含んでおり、曖昧な表現によってその理解が阻害されている。

1.3 本稿の技術的立場とUnix v6ベースの検証対象

本稿では、このような「xで読めるのか」という直感的疑問に対して、UNIX v6のカーネル実装に基づく構造的検証を行う。Unix v6は、現代のLinuxやBSDにも継承された基本的なファイルシステム構造（i-nodeベースのパス解決、アクセス制御、デバイス独立性）を備えており、概念の純粋形を確認するには格好の対象である。検証の中心には、パス解決を担当するnamei()関数の動作、i-node取得に関与するiget()、ディスクブロック読み込みを担うreadi()、およびそれらに伴うアクセス権チェック

の位置と粒度を置く。これにより、r権限とx権限の意味的・構造的差異を、動作原理と責務境界の観点から明示的に区別し、その背後にある設計思想を浮き彫りにする。

第2章　Unix v6におけるディレクトリの構造

2.1 ディレクトリのi-nodeとデータブロックの配置

Unix v6では、すべてのファイルとディレクトリがi-nodeによって管理される。i-nodeにはファイル種別、アクセス権限、オーナ情報、更新時刻などのメタデータと、データブロックのポインタ群が含まれる。ディレクトリもその例外ではなく、ファイル種別フィールドで「ディレクトリ」としてマークされ、対応するデータブロック群には「ディレクトリエントリ」の構造体が並ぶ。これらのエントリは、ファイル名と対応するi-node番号から構成されており、パス名の解決やファイル検索時の基盤となる。ディレクトリの内容は、通常のテキストとは異なり、人間が読むためのものではなく、システム内部でのi-node探索に特化して設計されている。

2.2 ディレクトリエントリ構造：14バイト固定名とi-node番号

Unix v6におけるディレクトリエントリは、16バイトの固定長構造で構成される。その内訳は、2バイトのi-node番号と14バイトのファイル名である。i-node番号は、ディスク上のi-node配列のインデックスであり、実体情報に直結するキーである。ファイル名はASCII文字列で、NULL終端ではなく、空白でパディングされる形式をとる。名前の最大長が14バイトに制限されているのは、この構造の単純さと処理速度を重視した当時の実装思想による。現代のファイルシステムではより柔軟な構造が採用されているが、UNIX v6ではこの固定長の設計が、namei()による高速な線形探索とディスクアクセスの最小化に貢献している。

2.3 namei()によるパス解決処理とdir読み出しの流れ

namei()は、与えられたパス（例：/usr/bin/ls）をディレクトリエントリ単位で順に分解し、各パス要素に対応するi-nodeを取得していく関数である。この関数は、対象ディレクトリのi-nodeをもとに、対応するデータブロックを読み出し、各エントリをスキャンすることで次のi-nodeを取得する。この際、i-node自体の読み出しにはiget()、ブロックのマッピングにはbmap()、ディスクブロックの読み出しにはreadi()が連携して動作する。重要なのは、namei()は各ディレクトリを単なる検索対象として扱っており、一覧取得は行わないという点である。これにより、namei()はx権限のみで動作可能であり、r権限が不要であるというアクセス制御上の合理性が生じる。ディレクトリの中身を知るのではなく、名前からi-nodeを解決するという「通過」の役割がx権限に対応するものである。

第3章　検索としてのx権限——namei()の動作分析

3.1 namei()の処理ステップとx権限の確認

namei()は、Unix v6においてパス解決の中核を担う関数である。呼び出されると、渡されたパス名（たとえば /home/user/file.txt）をスラッシュで分割し、左から順に処理していく。各ディレクトリ名に対して、その親ディレクトリのデータブロックを読み込み、該当するエントリ（14バイト名＋i-node番号）を探索する。その過程において、namei()はディレクトリの「中身」を一覧表示するのではなく、特定の名前を探して次のi-nodeを取得するための探索を行っている。したがって、必要なのは「検索する」権限、すなわちx権限であり、r権限は不要である。この違いは、アクセス制御の設計上重要であり、最小権限原則に照らして妥当であるといえる。

3.2 u.u_dirp, u.u_dent, u.u_pdir の更新と読み出し単位

namei()はグローバルなプロセス状態構造 user（u領域）を用いて処理を管理する。たとえば、u.u_dirpは現在処理中のパス名へのポインタを示し、u.u_dentは直前に一致したディレクトリエントリのコピーを保持する。また、u.u_pdirは直前の親ディレクトリのi-nodeを保持するポインタである。これらのフィールドは、検索中に逐次更新され、パスの解決状態を逐次追跡するために用いられる。ブロック単位での読み出しにおいては、1ブロック内の全ディレクトリエントリを線形探索するが、該当名を見つけるとすぐに処理が中断される。したがって、これは「部分的読み出し」にあたり、全件を読み込む必要はない。この点でも、r権限が不要であるという論理的整合性が確認できる。

3.3 iget(), bmap(), readi() による部分的ブロックアクセス

namei()の内部でi-nodeやデータブロックへのアクセスを実現しているのが、iget()、bmap()、およびreadi()の三関数である。iget()はi-node番号をもとに対象のi-node構造体をメモリ上に取得し、bmap()はそのi-nodeから物理ブロック番号を導出する。最後にreadi()がそのブロックを読み出してバッファに格納する。これらの関数は、特定のエントリだけを対象としたブロックの部分読み出しにも対応しており、ディレクトリ全体を一覧する必要はない。これは、i-nodeベースの間接参照によって構造的に可能となっている設計であり、検索対象の名前だけを頼りに、対象ファイルやディレクトリへ至る「道」を探索するという意味で、x権限の機能的定義に完全に合致する。したがって、xは「読める」のではなく「通過して検索できる」という意味であり、readi()の挙動もそれに沿って制限された範囲で動作している。

第4章　列挙としてのr権限――readi()とgetdirentの違い

4.1 lsコマンドの構造とreadi()の役割

lsコマンドは、指定されたパスがディレクトリである場合、その内容を一覧表示する処理を行う。そのためには、ディレクトリのデータブロック全体を読み取り、各ディレクトリエントリを順に取得する必要がある。この読み取りは、Unix v6においてはreadi()関数によって行われる。readi()はi-nodeを通じて取得したファイルのデータブロックにアクセスし、ファイルタイプに応じた読み出し処理を行うが、ディレクトリの場合にはエントリ列（14バイト名＋i-node番号）の繰り返しとして構造化されたバイト列を返す。この操作は、単一の名前検索とは異なり、すべてのエントリを反復的に処理する必要があるため、部分的な読み込みでは成立しない。すなわち、これは「列挙」であり、構造的にr（読み取り）権限を必要とする。

4.2 ディレクトリ全件列挙と権限チェックの位置

ディレクトリに対する全件列挙は、ディレクトリファイルそのものを通常のファイルとして開き、read系システムコールでバッファへと読み出すことで実現される。このとき、カーネルはopen()やread()の段階で、呼び出し元のプロセスに対してr権限があるかをチェックする。この権限チェックは、i-nodeのモードフィールドとプロセスのユーザ情報に基づいて行われ、許可されない場合は即座にアクセス拒否エラー（EACCES）が返される。これは、パスの探索段階（namei()）でのx権限チェックとは完全に別個であり、ファイルの中身を「読む」ことに対する明確な制御である。結果として、lsを実行してもr権限がなければ内容の表示はできず、「Permission denied」と表示されることになる。

4.3 r権限がない状態での挙動とアクセス拒否の理由

x権限のみを持つユーザが、あるディレクトリにcdで移動することは可能であるが、lsでその中身を見ることはできない。これは、cdはパス探索（namei()）に基づく動作であり、ディレクトリの中身の列挙を伴わないからである。一方で、lsは中身の列挙を必要とするため、r権限が求められる。実際に、rを外したディレクトリに対してlsを試みると、アクセスエラーが返される。これはUnixのアクセス制御が「読み取り」と「検索」を厳密に分離していることを示しており、両者を混同する設計であれば、このような選択的拒否は不可能である。結果として、アクセスに対する責務の分離、利用目的に応じた権限最小化、意図的な構造化が、Unixのファイルシステムにおいて設計理念として貫かれていることが理解される。

第5章　検索と列挙の境界──アクセス権限の設計的含意

5.1 namei()がrを要求しない合理的根拠

namei()は、パス名から目的のi-node番号を検索する処理を担う関数であり、その動作はディレクトリ内のエントリを1つずつ読み取り、名前が一致するかを検証する反復処理である。この際に必要とされるのは、ディレクトリエントリの探索が可能であるかどうか、すなわち「検索権限（x）」である。検索の過程では、エントリを列挙する必要はなく、目的の名前に対して逐次照合を行い、最初に一致した時点で処理が完了するため、全件を読み取る必要はない。ゆえに、読み取り権限（r）を必要としないという設計が合理的である。これは、ファイルシステムが必要最小限の情報にのみアクセスを許す原則に基づいており、オーバーアクセスを防ぐ観点からも重要である。

5.2 xのみでアクセス可能な構造の意図とセキュリティ

Unixにおけるx権限のみのディレクトリは、検索は許されるが列挙は許されないというセキュリティ構造を実現する。この設計により、たとえば公開ディレクトリ内の特定ファイルの存在を知っていれば、そのファイルへは直接アクセスできるが、存在を知らない他のファイルについては列挙できないという制限を課すことができる。これは、「知っている者にはアクセスを許し、知らない者には隠蔽する」という選択的情報公開の実装として機能する。たとえば、ウェブサーバのログディレクトリや一部のユーザホームディレクトリなどで、意図的にrを落としxのみを付与する運用が行われるのはこのためである。結果として、x権限は「探索可能性の最小単位」として機能し、r権限とは明確な責務分離が構築されている。

第6章　構造的総括──アクセス制御の分離と設計原則

6.1 検索と列挙の分離は「最小権限」設計の原型である

Unixファイルシステムにおけるディレクトリへのアクセス設計は、「検索（x）」と「列挙（r）」という二つの操作を厳密に分離している。この分離は、アクセス目的に応じた最小限の許可のみを付与する「最小権限の原則（Principle of Least Privilege）」の実装例として見ることができる。ユーザやプロセスは、検索だけが必要な場合にはxのみを、列挙まで必要な場合にはrを要求すべきであり、両者を混在させないことでセキュリティと可用性のバランスが保たれる。この設計は現代的なアクセス制御モデル（たとえばCapabilityベースやRBAC）にも共通する思想であり、Unix v6時点ですでにそれを先取りしていたことは注目に値する。

6.2 namei()の処理単位から見る責務境界と参照粒度

namei()の動作は、各ディレクトリエントリの照合とi-node番号の取得という最小粒度の検索に特化している。これに対し、readi()による読み出しはディレクトリ全体の構造を解釈し、複数のエントリを一括して処理することを前提としている。このように、ファイルシステムのカーネル実装において、検索処理と読み取り処理が異なる責務単位で構成されていることは、ソフトウェアアーキテクチャの視点からも明確な関心点である。参照の粒度を明示的に分離することで、システムの透明性とセキュリティが高まり、特権の誤用や過剰露出を避ける構造が生まれる。結果として、Unixのアクセス制御は単なるフラグの設定ではなく、参照構造そのものに根ざした高度な設計思想に基づいていることが理解できる。