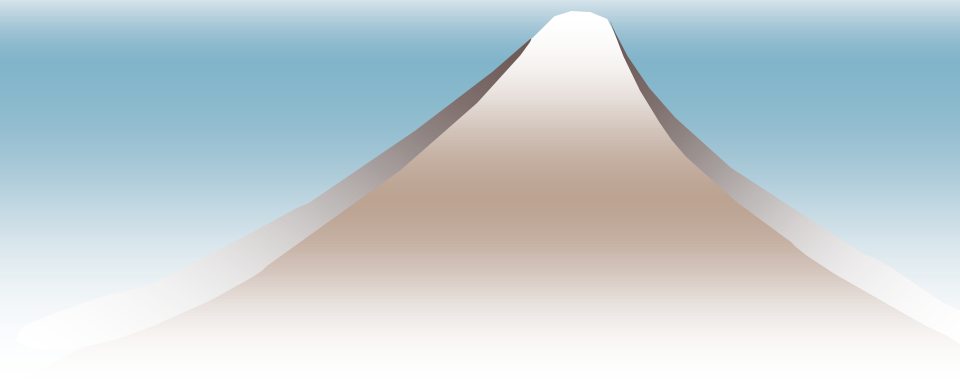


保護と安全性



「安全な」OS?

- ◆ そのOSでアプリケーションを動かしている限り危険なことは起こらないOS?
 - 無理な注文
- ◆ (3歩譲って)「情報漏えい, データの破壊, システムの機能不全が起きない」に限定すれば?
 - `rm -rf` で全部のファイルが消えた!
 - これが絶対できないOSがほしいのか?
 - 操作者の知識や酔い具合を判定したいのか?

◆ 他の人が自分のファイルを全部消した

- 誰でもできてしまったら明らかに安全なシステムとはいえない
- でも自分が放置した端末を勝手に使われたのならユーザの責任
- パスワードを盗まれたら？
- 単純なパスワードを推測されたら？
 - 誕生日パスワードはクレジットカード損害が補償されない

(控えめな)目標

- ◆ 安全と危険を一般的に線引きするのは不可能
 - 目的・操作方法・一般常識・想定ユーザなどあらゆることに依存する
- ◆ ⇒ ある操作の許可・不許可が決まる仕組みを理解する
 - アプリを安全にするための、OSとアプリとの責任分担を理解する

操作

◆「操作」≈

- ファイルに対する操作(読み・書き・実行)
- プロセスに対する操作(e.g., 強制終了)
- プロセス間通信(ネットワークへの接続)

◆ 守りたいデータのありか

- メモリ
- HDDなどの2次記憶装置
- ネットワーク

プロセス間の分離

ファイルシステムAPI

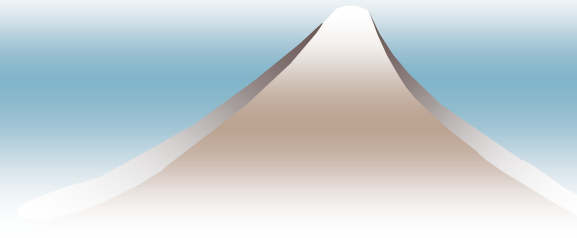
プロセス間通信API

復習

- ◆ 他のプロセスのメモリを読み書きすることはできない(例外: mmapなどを用いたプロセス間共有メモリ)
- ◆ HDDやネットワークなどの外部装置にアクセスするにはシステムコールを用いなければならない
- ◆ この前提で,
 - OSが提供するセキュリティ ≈ システムコールの実行許可・不許可の認定(authorization)

以降の概要

- ◆ ファイルシステム操作のauthorization
 - ファイルの許可属性(permission)
 - プロセスのUID
- ◆ ネットワーク操作のauthorization
- ◆ ネットワーク経由アプリケーションのセキュリティ



ファイルシステム操作の authorization

- ◆ 決定すべきこと: どのような場合に, ファイルをアクセスするシステムコールが成功するか(**アクセス制御; access control**)
 - ファイルの読み書き
 - Unix: $fd = \text{open}(f, a);$
 - Win: $handle = \text{CreateFile}(f, a, \dots);$
 - プログラムの実行
 - Unix: $\text{execv}(f, \dots);$
 - Win: $handle = \text{CreateProcess}(f, \dots);$

ファイルへのアクセス可否を決定するパラメータ

◆ アクセスを行ったプロセス P

- ユーザID: 操作を発行したプロセスは「誰か」≈「誰の権限で実行されているか」

◆ アクセスされているファイル F

- アクセス許可: 誰に対して, 何が許可されているか
- 所有者

◆ アクセスの種類 A

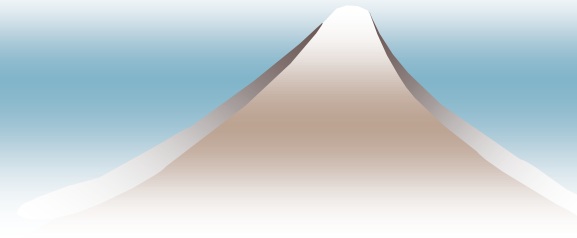
- 読み, 書き, 実行, アクセス許可変更, 所有者変更
 - open, execv, chmod, chown
- 

「誰」の意味

- ◆ OSが認識する主体=ユーザID
- ◆ 「Xはこのファイルを読める」の意味：
(ユーザ ID) **Xの権限で実行されているプロセス**
は, このファイルを読める
 - このXを「プロセスのユーザID (UID)」と呼ぶ
 - つまりプロセスのユーザID (誰の権限で実行しているか)がどう決まるかを理解すればよい

以降の話

- ◆ Unixを例に挙げて説明する
- ◆ Windowsも概念的には類似. だがAPIはもっと複雑



Unixにおける ファイルのアクセス許可

◆ アクセス許可

- read, write, execの3種類

◆ アクセス許可を出す対象

- 所有者, グループ, その他全員
- 「グループ」: ユーザIDの集合(詳細省略)

	Read	Write	Exec
所有者	OK	OK	NG
グループ	OK	NG	NG
その他	OK	NG	NG

ファイルのアクセス許可を 見る・変更するシステムコール

◆ 見る

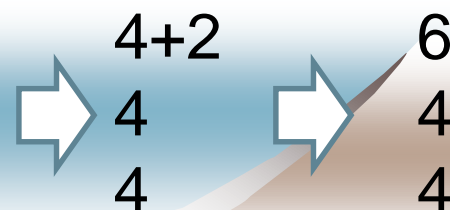
```
struct stat s;  
stat(path, &s);
```

◆ 変更する

```
chmod(path, mode);
```

◆ 許可情報は3桁の整数に符号化

	4	2	1
	Read	Write	Exec
所有者	OK	OK	NG
グループ	OK	NG	NG
その他	OK	NG	NG



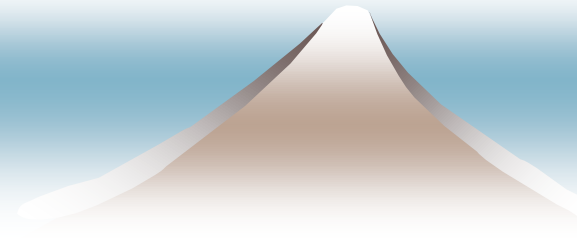
ファイルのアクセス許可を 見る・変更するコマンド

- ◆ 見る:

`ls -l path`

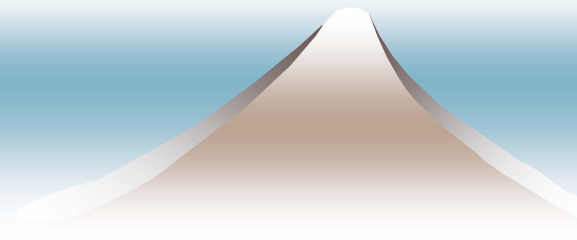
- ◆ 変更する

`chmod mode path`



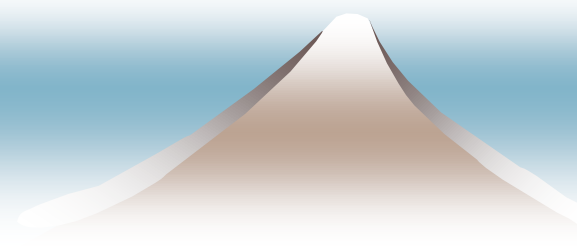
アクセス許可・所有者変更

- ◆ アクセス許可情報(read, write, exec)によって, open/execの成否が決まる
- ◆ ではそのアクセス許可を変更できるのは?
 - 答え: ファイルの**所有者**
- ◆ ではそのファイルの所有者はどう決まる?
 - 答え: そのファイルを作成したプロセスのユーザID
 - 特例: Unixでは特権ユーザ(root)がそれを変更できる (chown システムコール・コマンド)



ファイルに対する操作(まとめ)

- ◆ 読み書き (open) : ファイルに対するアクセス許可により, 成否が決まる
- ◆ アクセス許可変更 (chmod) : 所有者とrootのみ成功
- ◆ 所有者変更 (chown) : rootのみ成功

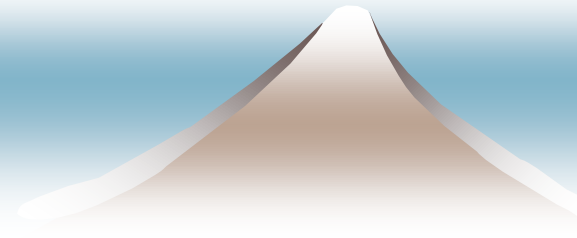


プロセスのユーザID

- ◆ そのプロセスがどのユーザIDの権限で実行されているかを示す, **プロセスの属性**
- ◆ Unixでは3種類ある
 - **実**ユーザID (real user id; uid)
 - **実効**ユーザID (effective user id; euid)
 - **保存**ユーザID (saved user id)
 - アクセス権の検査は**実効ユーザID**に対して行われる
 - 他のユーザIDは何のため? (→後で)

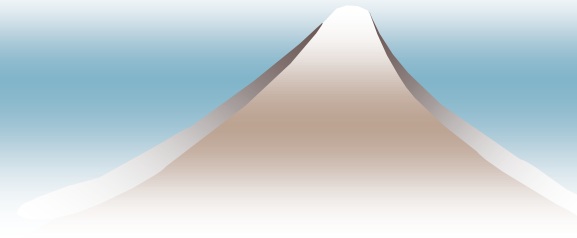
プロセスのユーザIDを 見るコマンド

- ◆ ps
- ◆ top



プロセスの実効ユーザIDは「どう」決まるのか? (1)

- ◆ Aが自分のファイルのアクセス許可を「自分だけ」(600)に設定した
- ◆ AはEmacsでAのファイルを開けるが, 他のユーザ(B)は開けない
- ◆ それはA (B)が起動したEmacsの実効ユーザIDがA (B)だから
- ◆ でもなぜそうなのか?



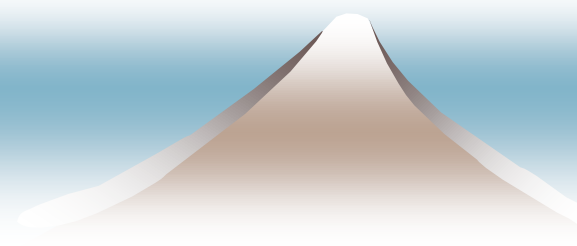
実効ユーザIDはどう決まる? (2)

- ◆ AはAのユーザIDとパスワードでログインしたからに違いない
- ◆ でもプロセスを起動するたびにパスワードを渡して実効ユーザIDを決めているようには見えない(cf. fork, exec)
- ◆ そもそも実効ユーザIDを変更するのに「必ず」パスワードが必要というのは不便
- ◆ そもそもパスワードが特別安全な認証手段というわけでもない

◆ 答え:

- 「実効ユーザID, 実ユーザID」を変更できるシステムコールがある

◆ なりすまし (impersonation) API



成りすまし(Impersonation)

◆ システムコール:

- `seteuid(new_euid);`
- `setreuid(new_uid, new_euid);`

◆ コマンド:

- `su`

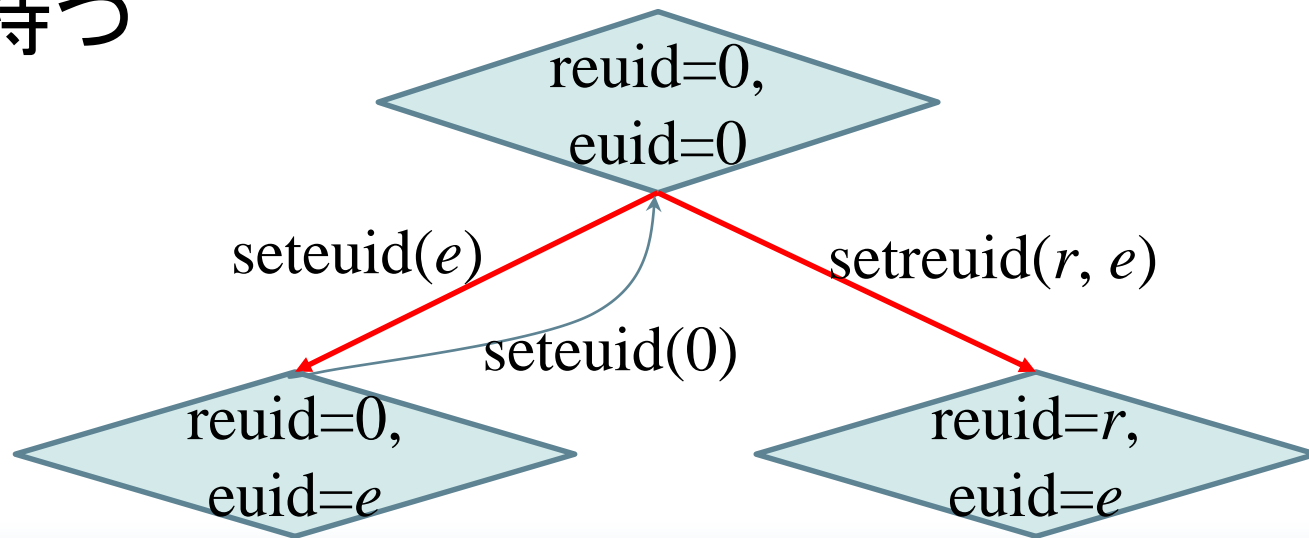
◆ 「抜け道」のようだが、任意のユーザ権限で実行するプロセスを作るのに必要

- 「login」を受けつけるプログラム(e.g., sshデーモン)
- 各種サーバ: メールサーバ, Webサーバ, etc.

◆ 認証はアプリの責任!

seteuid, setreuidの動作

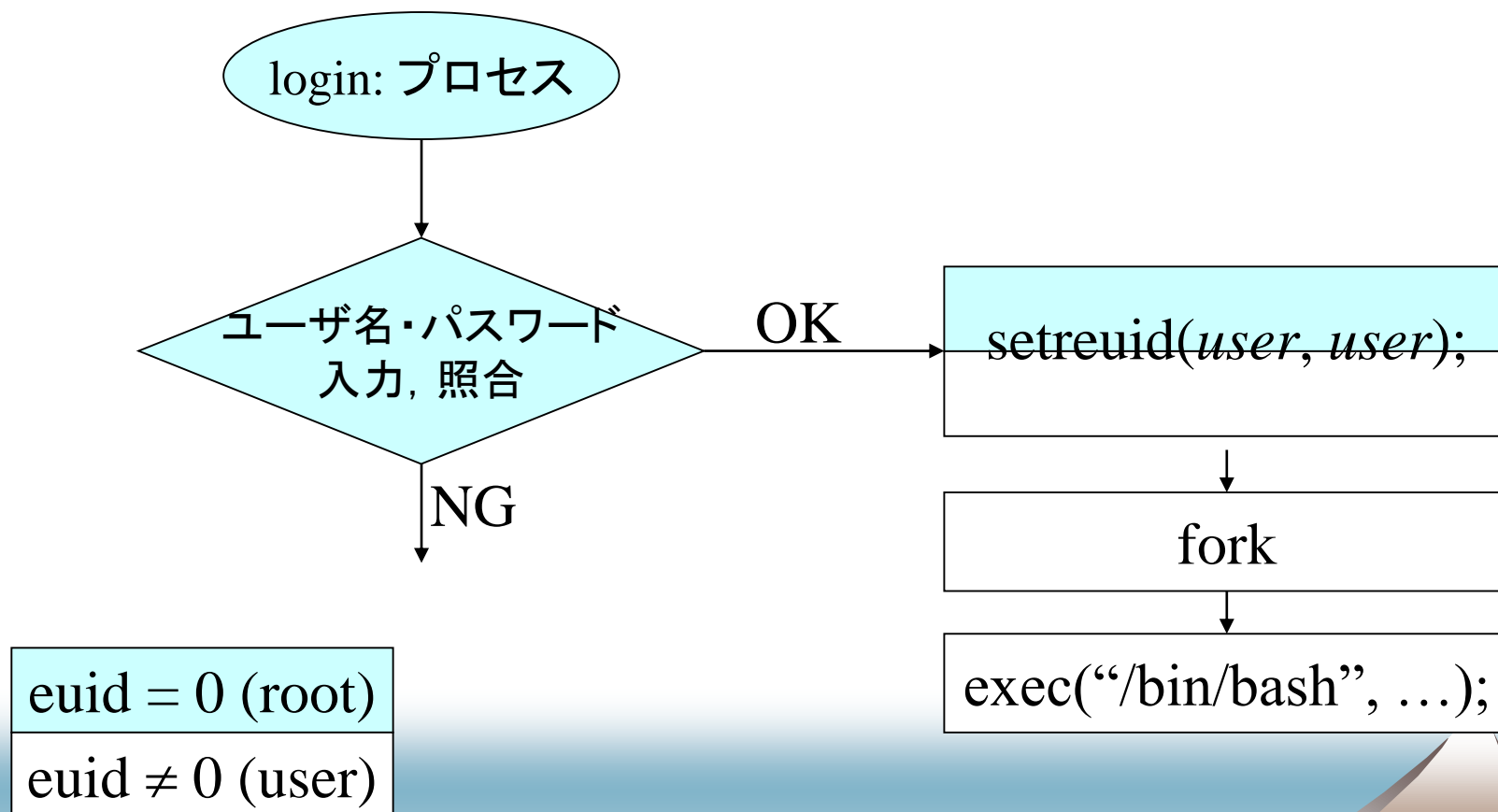
- ◆ 各プロセスは二つの属性 実効uid (effective uid; euid), 実 uid (real uid; reuid) を持つ



誰が成りすましを実行できるのか？

- ◆ もちろん, 誰でもできたらsecurityはゼロ
- ◆ 基本的なUnixルール:
 - $\text{euid} = 0$ (特権ユーザ; root)は無条件に可能
 - つまり「誰にでもなりすませる」
 - 通常ユーザは, 現在の euid , ruid のどちらかにのみそれぞれを変更できる
 - 例: 一時的に一般ユーザに降格($\text{euid} \neq 0$, $\text{ruid} = 0$), その後またrootに戻る

まとめ: 私はなぜ私か?



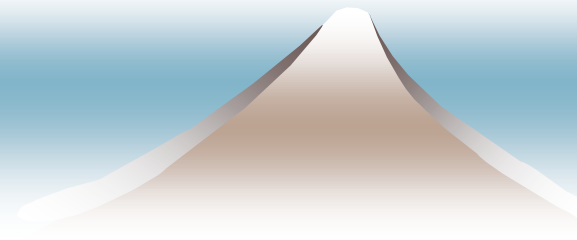
非rootからrootになる手段はないのか？

◆ 答え: ある

- もしないのなら, “su”コマンドはありえない

◆ 実行可能ファイルの属性: **setuid bit**

- 実行(exec)された際に,
“プロセスの実行ユーザID = そのファイルの所有者”
となる
- このプログラムを実行する際は誰でも自分になって良い, という許可
- 時に必要. だがsecurity holeのもと



ネットワークアプリケーションとセキュリティ

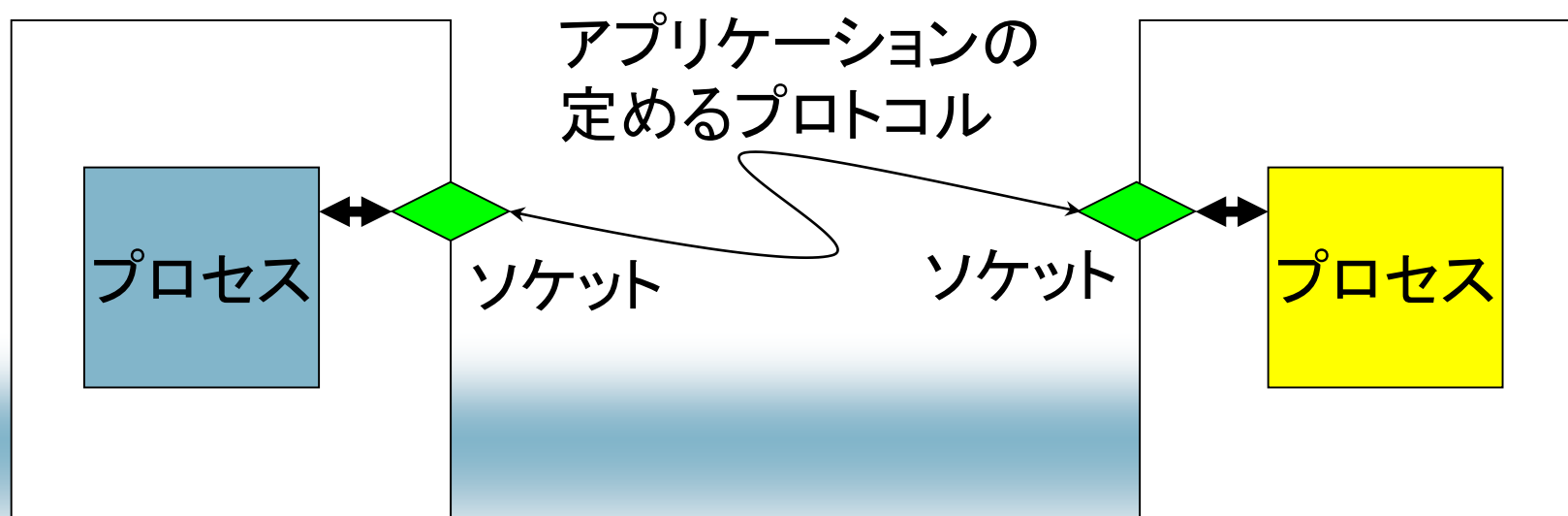


代表的ネットワーク アプリケーション

- ◆ Web
 - ◆ メール
 - ◆ リモートログイン
 - SSH, RSH, etc.
 - ◆ ネットワークファイル共有
 - CIFS/Samba, NFS, etc.
 - ◆ 遠隔端末
 - X Window, Windows Remote Desktop
- ネットワークアプリケーション ≈
ネットワーク上の他のマシンを利用して機能を実現するアプリケーション

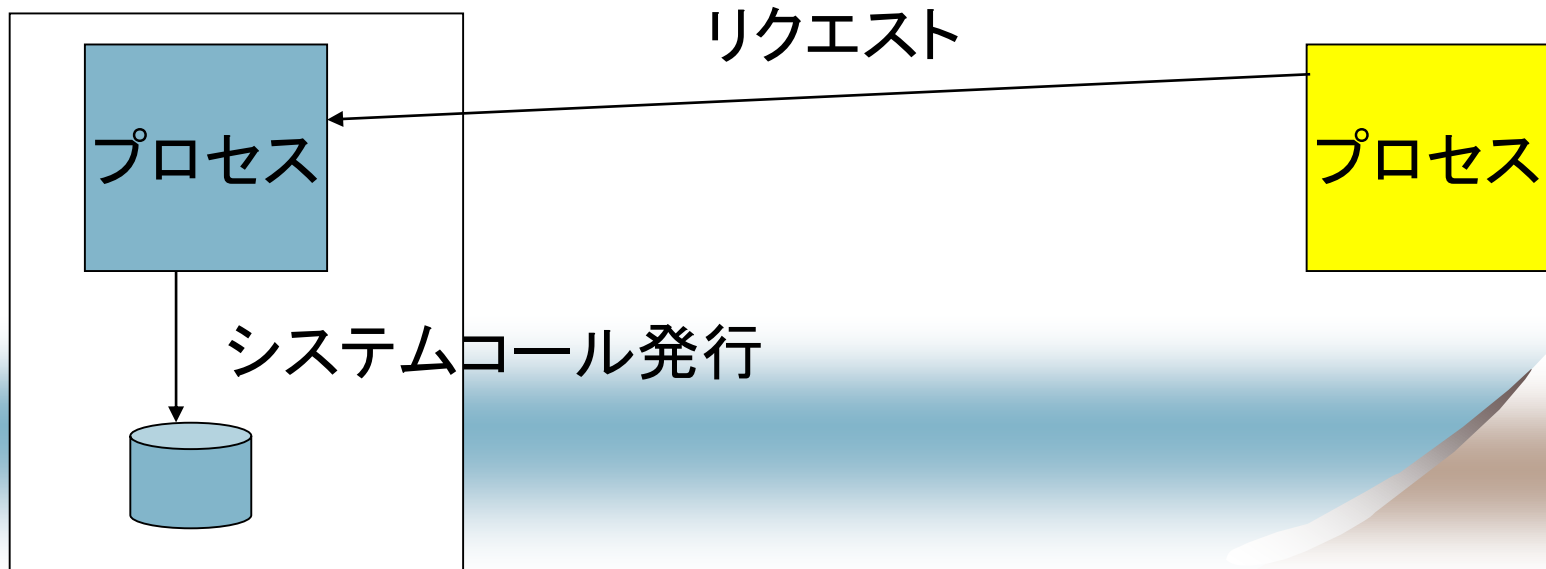
ネットワークアプリケーションの 基本的構成

- ◆ クライアント: 機能を利用するプロセス(必要に応じて立ち上がる)
- ◆ サーバ: クライアントからの呼び出しに備えて常時立ち上がっている(デーモン)プロセス



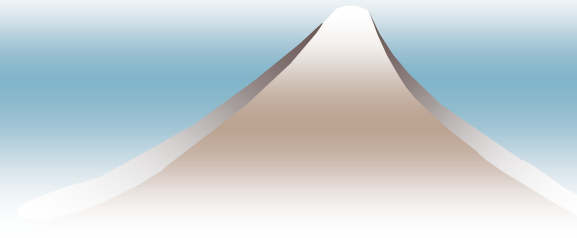
ネットワークアプリケーションに固有の難しさ

- ◆ アクセスをする(論理的な)主体は遠隔にいるクライアント
- ◆ 実際にOSにリクエストを出すのはサーバ上にあるプロセス



ネットワークAPI : ソケット

- ◆ さまざまなプロセス間通信プロトコルに共通のAPI
 - インターネット(IP, UDP, TCP)
 - いくつかのLANプロトコル(AppleTalk, etc.)
 - 1 Unixコンピュータ内 (Unix domain)



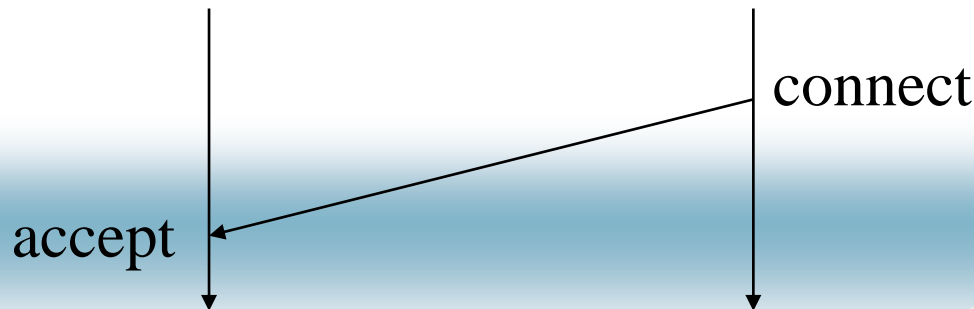
ソケットAPI

◆ サーバ

```
s = socket(...);  
bind(s, addr, port);  
listen(s, n);  
new_s = accept(s);  
send/recv(new_s, ...);
```

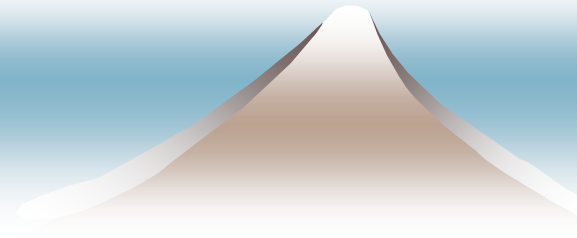
◆ クライアント:

```
s = socket(...);  
connect(s, addr, port, ...);  
send/recv(s, ...);
```



まとめ知識

- ◆ `ps -ef`
 - すべてのプロセスを表示
- ◆ `netstat -a`
 - 現在使われているソケットの状態を表示
 - 待機中(LISTEN)
 - 接続中(ESTABLISHED)

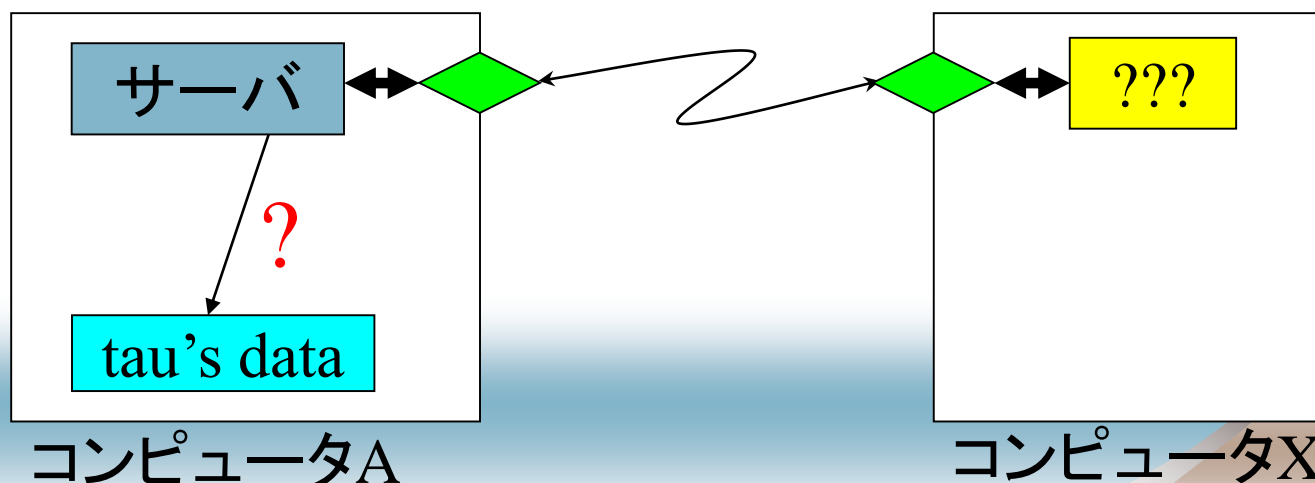


インターネットとセキュリティ (1)

- ◆ 「誰が」ソケットに対してアクセスできるのか?
 - 通常, acceptしているソケットには誰でもconnectできる
 - 相手のIPアドレスやポートに基づいてconnectの許可・不許可をする仕組みはある
 - iptables
 - ルータ機器に備わったフィルタリング
 - しかし, 相手プロセスのユーザIDに基づいたアクセス制御は組み込まれていない
 - 全世界のユーザを管理・把握することはできないので, 無理もない

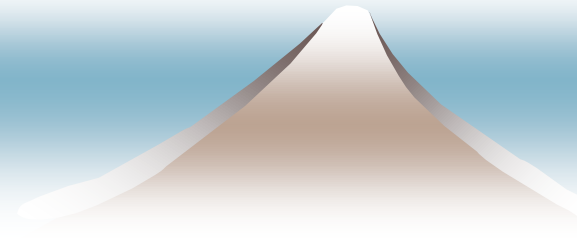
インターネットとセキュリティ (2)

- ◆ 現在のOSにはインターネット越しのユーザに対する保護・アクセス制御の概念はない
- ◆ アクセス制御(相手が誰なのかの判別; 認証)はアプリケーションの役目



ネットワークアプリケーションの アクセス制御の実例

- ◆ アプリケーションごとに異なる, アクセス制御の方針
- ◆ それを実現するための, アプリケーションの構成

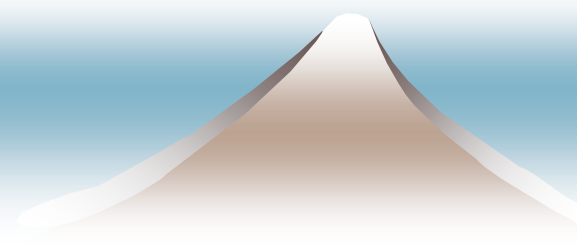


例1: 遠隔ログイン

◆ 例: SSH

◆ 要件

- 任意のクライアントから接続を受け付ける
- クライアントがログイン後のユーザ名を主張する
 - `ssh tau@server.eidos.ic.i.u-tokyo.ac.jp`
- 認証が成功したら主張されたローカルユーザXと同一の権限を与える

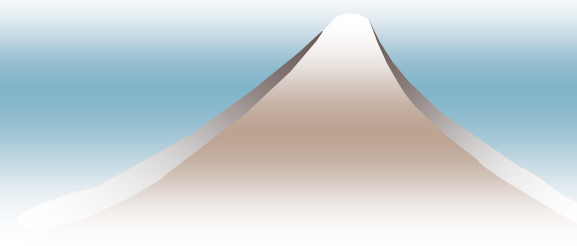


遠隔ユーザの認証

◆ パスワード認証

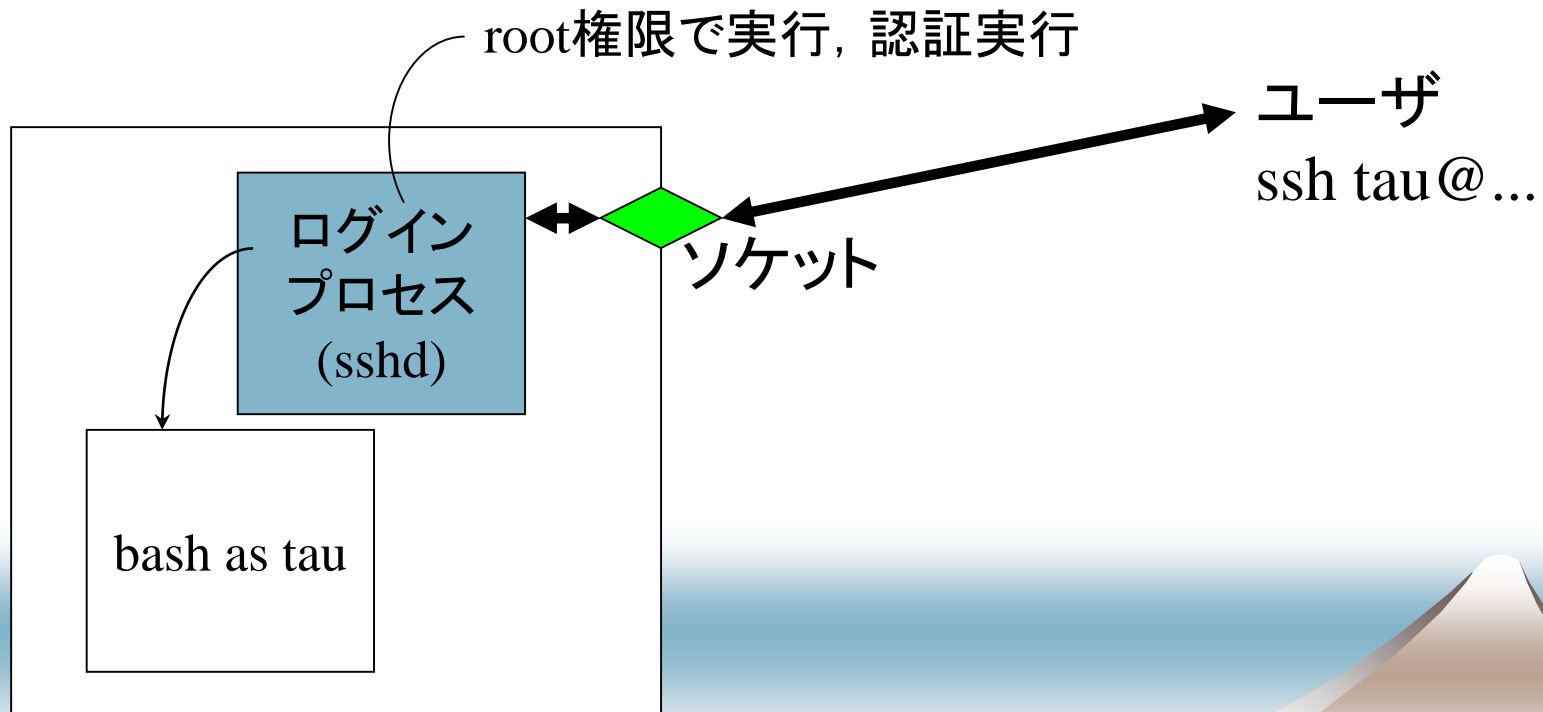
- クライアントがサーバへ、ユーザXのパスワードを送信
- SSH, RSH

◆ 公開鍵認証

- サーバに保管してある公開鍵と、クライアントに保管してある秘密鍵が、対応する鍵の対であることを検証する
 - SSH, PGP
- 

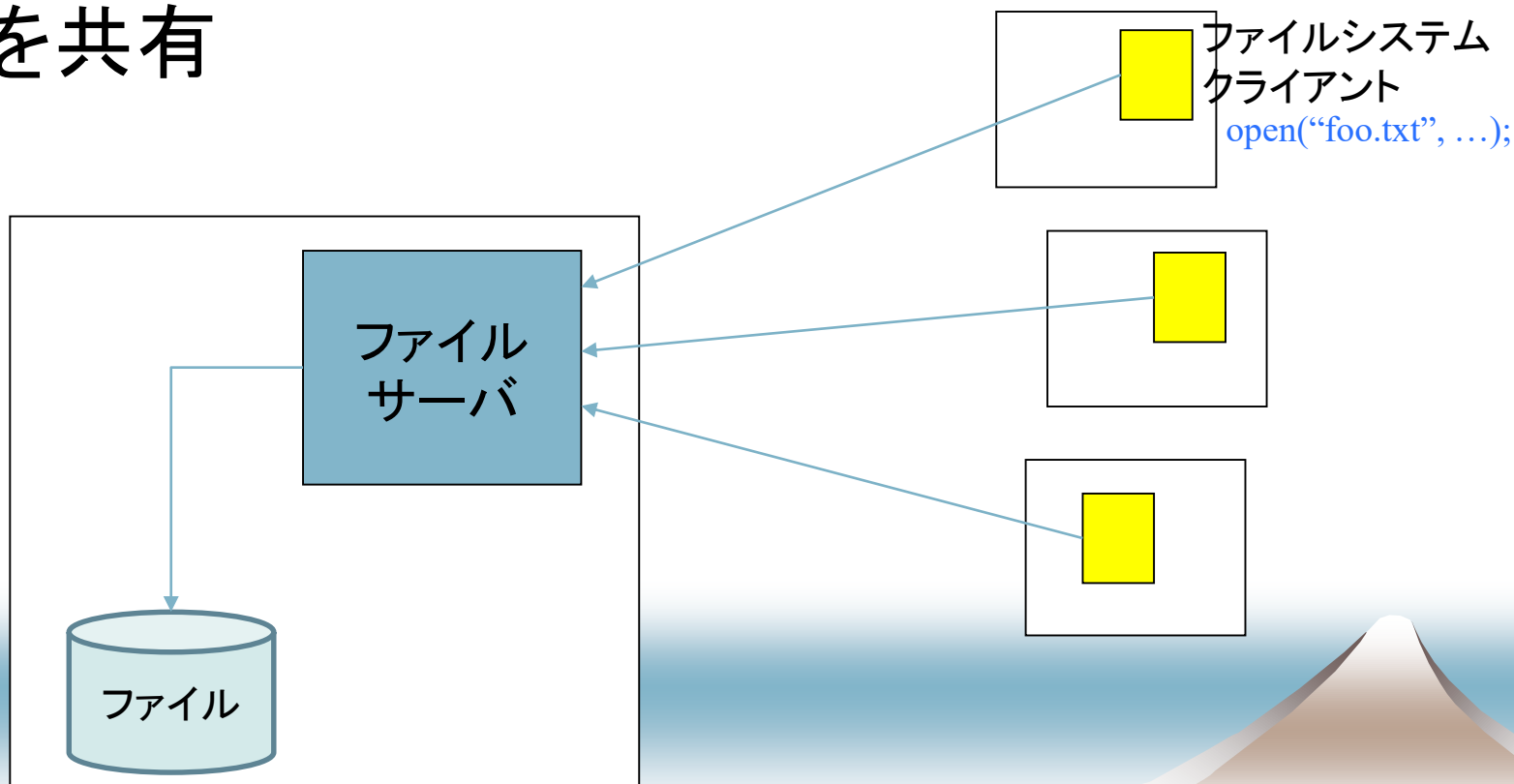
認証後の処理

- ◆ 認証成功後, 要求されたユーザに成りすま
す(setuid)



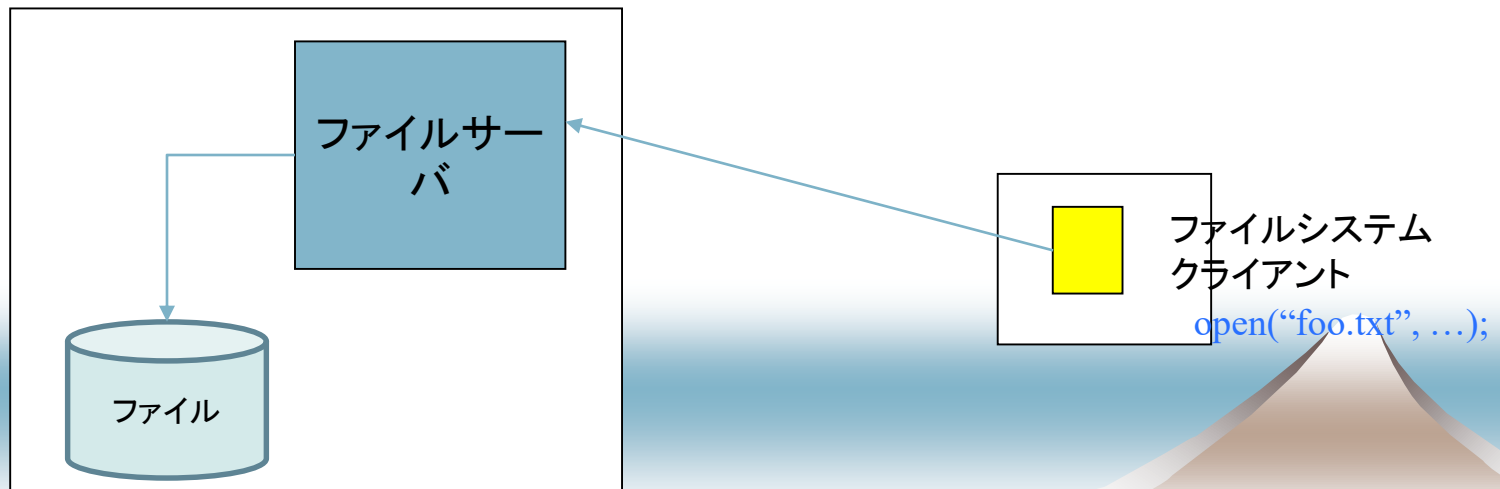
例2: ネットワークファイルシステム

- ◆ 例: CIFS (Samba), NFS
- ◆ 多数のマシン, 多数のユーザ間でファイルを共有



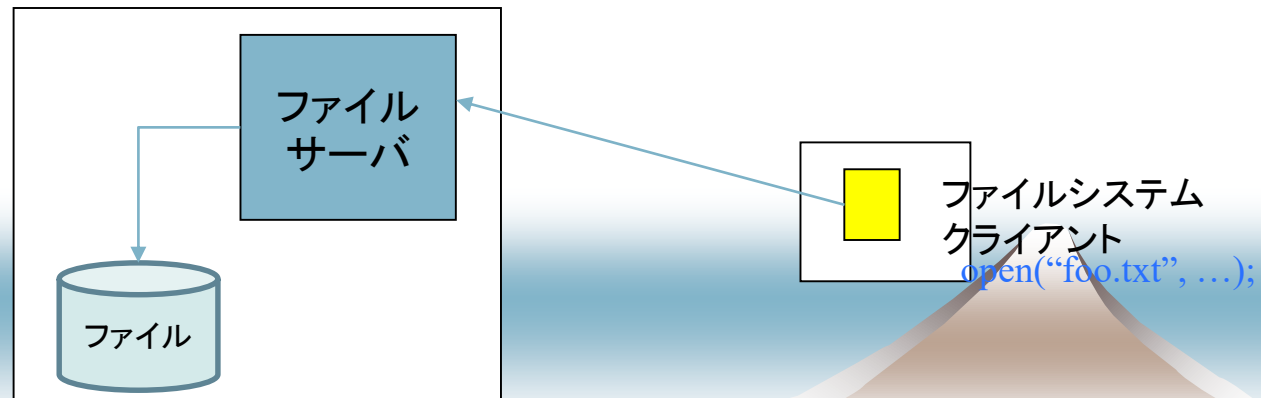
ファイルシステム:認証方法

- ◆ ファイルサーバが、どうやって、クライアントプロセスのuidを確かめるか？



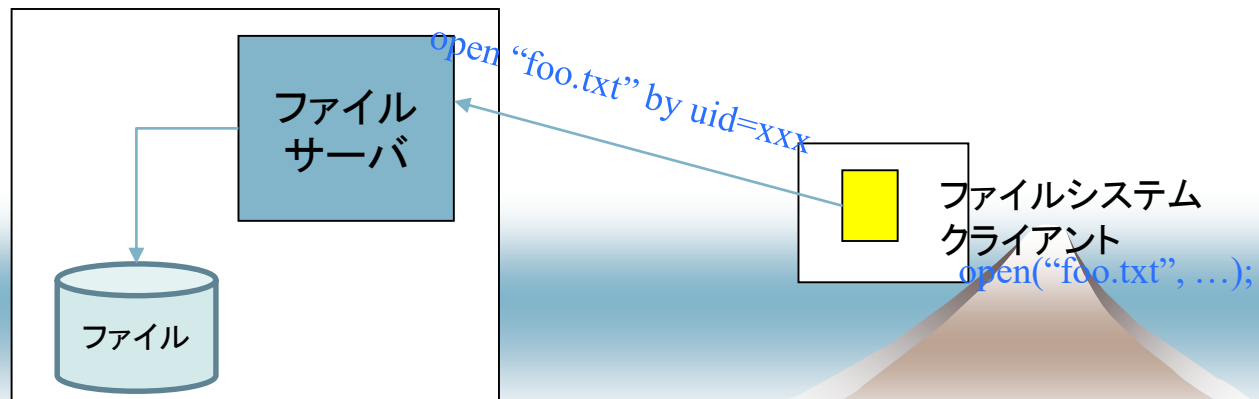
認証方法1 (CIFS)

- ◆ クライアントごとにsshと似た認証をする
 - パスワードなど入力が必要だと, 多数のクライアントでの共有には手間が多すぎ
 - パスワードなしで認証しても, クライアントごとに1プロセスを消費するのは多クライアント・ユーザでは問題



認証方法2 (NFS)

- ◆ クライアント「マシン」をまとめて認証
 - ファイルサーバ: 特定マシン(IPアドレス, ポート)からの依頼は信用する
 - ポートは, 特権ポート(rootでしか使えないポート; 通常1024以下)を使う



ネットワークセキュリティ(まとめ)

- ◆ 多くの部分がOSの守備範囲外
- ◆ 遠隔ユーザの認証, それに基づくアクセス制御を**正しく**実行するのは大部分が(OSではなく)サーバアプリケーションの役目
- ◆ Unixはrootに,
 - ほとんどのファイルへのアクセス権限
 - 他のユーザになりすます権限を与え, あとはアプリケーションに任せる