

オペレーティングシステム



資料 第 **7** 分冊(2021)

村田正幸 (murata@ist.osaka-u.ac.jp)
○松田秀雄(matsuda@ist.osaka-u.ac.jp)

2

第6回課題 2.の解説

	Allocation			Max			Need			Available		
資源型j	1	2	3	1	2	3	1	2	3	1	2	3
プロセス										3	3	2
P1	0	1	0	7	5	3	7	4	3			
P2	2	0	0	3	2	2	1	2	2			
P3	3	0	2	9	0	2	6	0	0			
P4	2	1	1	2	2	2	0	1	1			
P5	0	0	2	4	3	3	4	3	1			
P5の資源割り付け要求(0 3 0)												

課題2. について解説します。

課題2は、このスライドのように、資源が3種類、プロセスが5個あるときの、銀行家アルゴリズムによる資源割り付け要求の判定の課題です。

このスライドの設定で、P5が資源割り付け(0 3 0)を要求した場合を判定することになります。

第6回課題 2.の解答(1)

	Allocation			Max			Need			Available		
資源型j	1	2	3	1	2	3	1	2	3	1	2	3
プロセス										3	3	2
P1	0	1	0	7	5	3	7	4	3			
P2	2	0	0	3	2	2	1	2	2			
P3	3	0	2	9	0	2	6	0	0			
P4	2	1	1	2	2	2	0	1	1			
P5	0	0	2	4	3	3	4	3	1	Request 0 3 0		

Request(5,j) ≤ Need(5,j)かつ
Request(5,j) ≤ Available(j) (j=1,2,3)なので
Request(5,j)は割り付け可能

まず、P5による(0 3 0)の要求が割り付け可能かどうかを判定します。

要求された(0 3 0)は、P5の必要量であるNeedの範囲内であり、かつ、利用可能資源量であるAvailableの範囲内でもあるので、割り付け可能となります。

演習問題 No.6 2.の解答(2)

	Allocation			Max			Need			Available		
資源型j	1	2	3	1	2	3	1	2	3	1	2	3
プロセス										<u>3</u>	<u>0</u>	<u>2</u>
P1	0	1	0	7	5	3	7	4	3			
P2	2	0	0	3	2	2	1	2	2			
P3	3	0	2	9	0	2	6	0	0			
P4	2	1	1	2	2	2	0	1	1			
P5	<u>0</u>	<u>3</u>	<u>2</u>	4	3	3	<u>4</u>	<u>0</u>	<u>1</u>	Request 0 3 0		

資源割り付け後の状態は安全か？

次に、要求(0 3 0)の通り資源割り付けを実行した後の状態が安全かどうかを判定します。

割り付け後の資源の状態はこのスライドの通り、AvailableとNeedから資源2を3だけ減少させ、Allocationに加えることになります。

演習問題 No.6 2.の解答(3)

	Allocation			Max			Need			Available			Work		
資源型j	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
プロセス										3	0	2	3	0	2
P1	0	1	0	7	5	3	7	4	3						
P2	2	0	0	3	2	2	1	2	2						
P3	3	0	2	9	0	2	6	0	0						
P4	2	1	1	2	2	2	0	1	1						
P5	0	3	2	4	3	3	4	0	1						

資源割り付け可能なプロセスの系列が存在しないので安全ではない→資源割り付け要求は許可されない

この状況で、安全かどうか、すなわち、資源割り付け可能なプロセスの系列が存在するかどうかを判定します。

P1からP5までの必要量Needと、割り付け可能な資源量Availableを比較すると、P1からP5のいずれにおいても必要量を満たすものはありません。

つまり、資源割り付け可能なプロセスが存在しないことになります。

このことから、(0 3 0)の資源割り付け後には安全ではなくなるので、この資源割り付け要求は許可されないことになります。

プロセス管理の実装

- オペレーティングシステムごとにいろいろな方式がある
- プロセスの生成とプログラムの実行が別 (UNIX)
 - プロセスの作成: fork
 - プログラムの実行: exec
- プロセスの生成とプログラムの実行が一体化 (Windows NT系)
 - spawnなど

今回の講義の本題に入ります。

今回は、プロセス管理の実装、つまり、具体的なプロセス管理の方式について、オペレーティングシステムごとに説明していきます。

まず、プロセスの生成と、プログラムの実行との関係です。

UNIX系のOSでは、プロセスの生成はforkで、プログラムの実行はexecでいうように、それぞれのシステムコールが別々に用意されています。

一方、Windows NT系のOSでは、プロセスの生成とプログラムの実行が一体化しており、spawnというシステムコールで行われます。詳細は以降のスライドで説明します。

⋮

7

fork

- プロセスの複製を作る
- UNIXの成功により他のOSに広まった
- 利点
 - 動作が単純(実行したプロセスの複製を作るだけ)であり、引数が不要
 - プロセス生成時のプロセス領域の初期化が容易(親プロセスのプロセス領域の複製が取られるので、変更が必要な箇所だけ修正すればよい)

forkについて説明します。

プロセスの複製を作るシステムコールであり、UNIXの成功により他のOSに広まりました。

利点としては、動作が単純(実行したプロセスの複製を作るだけ)であり、引数が不要であることと、プロセス生成時のプロセス領域の初期化が容易である(親プロセスのプロセス領域の複製が取られるので、変更が必要な箇所だけ修正すればよい)ことです。

forkの仕様:

- 呼び出し形態
 - pid=fork() (引数無し)
- 戻り値:
 - エラー -1
 - 親プロセス 子プロセスのプロセスID
 - 子プロセス 0
- 親プロセスと子プロセスのプロセス領域は同じ内容
 - プロセスIDは異なる

ユーザプログラムではforkは関数として定義されています。

戻り値は、次の通りです。

エラー -1

親プロセス 子プロセスのプロセスID

子プロセス 0

実行の効果としては、親プロセスのプロセス領域のコピーが子プロセスに引き継がれるので、両者のプロセス領域は同じ内容を持ちます。

プロセスIDは異なり、別プロセスとして管理されます。

forkの使用例(リモートログイン)

```

:
:
:
if ((pid=fork())!=-1) {
    perror("fork");
    exit(1);
}
if (pid==0) { /* 子プロセスの処理 */
    while(ネットワークからデータ受信) {
        データを画面に表示
    }
} else { /* 親プロセスの処理 */
    while(キーボードからデータ読み取り) {
        ネットワークに送信
    }
}
}

```

forkを使用するプログラム例を示します。

これは、リモートログインの例で、最初のif文でforkの実行でエラーがないか、つまり子プロセスが正常に作られたかをチェックしています。

次のif文では戻り値が0、つまり子プロセスの処理で、リモートログインでネットワークから受信があるとそれを画面に表示します。

最後のelse文はそれ以外の戻り値、つまり親プロセスの処理で、キーボードから入力したデータをネットワークに送信します。

親プロセスと子プロセスに分けることで、ネットワークからのデータ受信と、キーボードからのデータ入力という独立した処理を独立に実行できます。

10

exec

- `execv`, `execve`, `execl`, `execle`などの種類がある
- システムコールとしては一つ、ライブラリ関数の形で細かな動作の指定をする
- 引数：
 - プログラムのファイル名
 - プログラムに与える引数、など
- 効果：
 - 現在のプロセスに指定したプログラムファイルをロードして実行（現在のプロセスで実行していたプログラムは消去）
- 現在のプロセスとは別プロセスでプログラムを実行するには、先に`fork`で子プロセスを生成し、子プロセスで`exec`を実行する

次に、UNIXでのプログラムの実行である`exec`システムコールについて説明します。

「プログラムの実行」となっていますが、実際には、これを実行したプロセスのコード領域を、引数で指定したプログラムファイルのコードに置き換え、コードの最初から実行します。

現在のプロセスとは別プロセスでプログラムを実行するには、まず`fork`で子プロセスを生成しておいて、子プロセスで`exec`を実行する必要があります。

forkとexecの使用例

```

:
:
If ((pid=fork())!=-1) {
    perror("fork");
    exit(1);
}
If (pid==0) {
    fd=open(ファイル,....);
    現在のプロセスの標準出力を fdに切りかえる
    exec(プログラム名、引数、....);
    perror("exec failed"); /* 常にエラーを返す(なぜか?) */
    exit(1);
}
wait(&status); /*終了待ち, セマフォのwait操作とは全く別の
関数であることに注意 */

```

forkとexecの使用例を示します。

forkで生成された子プロセスでは、新たにファイルをオープンし、プロセスの標準出力をそのファイルにすることで、出力をファイルに書き込むようにして、プログラムを切り替えるというプログラムになっています。

親プロセスはwait関数(セマフォのwait操作ではなく、子プロセスが終了するのを待つシステムコール)を実行することで、子プロセスが終わるのを待ちます。

プロセスの終了(1)

- `exit(int status)`
 - 親プロセスに実行の状態を表す値(status)を返す
- 内部で`exit`システムコールを実行
 - バッファつき入出力の`fclose`処理
 - `fopen`したファイルの後始末
 - `exit`システムコールを呼び出す

プロセスの終了処理の実装について説明します。

`exit`という関数が提供されており、子プロセスが`exec`を実行すると、引数の値を親プロセスに返します。

この値は実行の状態(プロセスが正常に終了したか、エラーがあったかなど)を表します。

`exec`関数は、内部で`exec`システムコールを実行し、`fclose`処理を実施してプロセスを終了させます。

プロセスの終了(2)

- 子プロセスの終了を待つ

```
int status;  
pid=wait(&status);
```
- 終了済みの子プロセスがある場合
 - その情報が返される
- `fork()`を複数回実行して子プロセスを生成
 - 子プロセスの数だけ`wait()`を呼び出す必要がある
- `status` 子プロセスの終了状態
 - どのような値を`exit`システムコールに渡したか

`exit`の引数の値がどのように、親プロセスに渡されるかを説明します。

前で説明したように、子プロセスは`fork`で生成されますが、`exit`の値は`fork`の戻り値として返されるわけではありません(`fork`の戻り値は親プロセスの場合は、子プロセスのプロセスID)。

実際には、親プロセスが`wait`関数を実行したとき、引数として渡されたポインタで指される先に、子プロセスの`exit`関数の引数値が書き込まれます。

`fork()`を複数回実行して子プロセスを生成したときは、生成した子プロセスの数だけ`wait()`を呼び出す必要があります。

プロセスの終了(3)

wait(&status)

- waitは子プロセスが終了するまで待ち、そのpidを返す
- 子プロセスはexitの引き数に終了ステータスを渡す
 - waitのstatusにその終了ステータスが入って戻る

wait()は子プロセスが終了するまで待ち、そのpid(プロセスID)を返します。
子プロセスはexitの引き数に終了ステータスを渡し、waitのstatusにその終了ステータスが入って戻ります。

セマフォについて

- 基本的な同期機構
- 多くのOSに備わる
- signal操作/wait操作を正しく使わなければならない
 - さもないと相互排除失敗orデッドロック
- signal操作とwait操作がプログラムの中に分散
 - 対応関係が正しいか検証が難しい
- より良い相互排除機構が提案されている
 - モニタ、メッセージによる同期

セマフォの実装について説明します。

セマフォは基本的な同期機構であり、多くのOSに備わっています、signal操作/wait操作を正しく使わなければならない、さもないと前回の課題にあったように、相互排除失敗やデッドロックを引き起こします。

signal操作とwait操作がプログラムの中に分散してしまうため、対応関係が正しいか検証が難しいという特徴があります。

このため、より良い相互排除機構が提案されています。例えば、モニタ、メッセージによる同期などです。

⋮

基本同期命令

16

(synchronization primitive)

- プロセス間で同期を取るための基本的な命令
- 不可分な操作である
 - 必ずしも1つの機械語命令ではない
- 相互排除の要件
 - 相互実行 (mutual execution) を禁止する
 - デッドロック (deadlock) を禁止する

セマフォのように、相互排除を行うための命令は、基本同期命令と呼ばれます。

これは、プロセス間で同期を取るための基本的な命令となっています。

命令の要件の一つは、不可分な操作であることです。ただし、必ずしも1つの機械語命令になっているわけではありません。

また、次の相互排除の要件を満たす必要があります。

相互実行 (mutual execution) を禁止することと、デッドロック (deadlock) を禁止することです。

単一プロセッサでのセマフォ

- もっとも簡単な基本同期命令
- 単一プロセッサであれば、**割り込みを禁止すれば、横取りは起こらない**(相互排除の実現が容易になる)
- 割り込み禁止は手間がかからない
 - 通常1命令
- ユーザプロセスで割り込みを禁止するのは問題あり
 - 誤って禁止した場合(特に意図せずに禁止して、割り込みを再び許可しなかった場合)不具合が発生する(**どのような不具合か?**)
- 改良
 - 利用したい資源に関連する割り込みだけ禁止

単一プロセッサでのセマフォは、もっとも簡単な基本同期命令です。

単一プロセッサであれば、割り込みを禁止すれば、横取りは起こりません。つまり、プロセスの実行が中断されて別のプロセスに切り替わるのを制限でき、相互排除の実現が容易になります。

割り込み禁止は手間がかからない操作であり、通常1命令で実行できます。

ただし、ユーザプロセスで割り込みを禁止することには問題があります。特に、意図せずに禁止してしまった場合に、禁止を解いて割り込みを再び許可する必要性を意識せずにいると、割り込みがずっと許可されなかった場合に不具合が発生します。

どのような不具合でしょうか？ヒントとしては、カーネルの呼び出しが必要なときに、割り込みがどのような役割を果たすかを考えてみてください。

改良としては、利用したい資源に関連する割り込みだけ禁止することがあげられます。

マルチプロセッサとセマフォ

- UNIXの相互排除
 - 相互排除操作をシステムコールで実現
- 初期のUNIXのマルチプロセッサ対応
 - システムコールを実行できるプロセッサを同時には1個だけに限定する(カーネルを実行できるプロセスを同時には1個だけにする)ことで、相互排除を実現
 - マルチプロセッサ対応UNIXでは、複数のプロセッサが同時にシステムコールを実行可能
 - カーネルの機能だけでは相互排除を実現できない(原則としてハードウェアによるサポートが必要)

マルチプロセッサシステムでのセマフォの実装について説明します。

セマフォを使った、UNIXの相互排除では、相互排除操作をシステムコールで実現しています。

初期のUNIXのマルチプロセッサ対応では、システムコールを実行できるプロセッサを同時には1個だけに限定する(カーネルを実行できるプロセスを同時には1個だけにする)ことで、相互排除を実現していました。

現在のマルチプロセッサ対応UNIXでは、複数のプロセッサが同時にシステムコールを実行可能となっています。

実際には、カーネルの機能だけでは相互排除を実現できないため、原則としてハードウェアにより相互排除をサポートすることが必要となります。

UNIXでのセマフォの使用例

- semget: セマフォの新たに作成または作成済みのセマフォを検索
- semctl: セマフォの制御(初期値の設定、値の読み出し、セマフォの削除など)
- semop: セマフォの操作(セマフォの**wait**操作, **signal**操作など)

UNIXでのセマフォの使用例について説明します。

UNIXでは、semget, semctl, semopなどのセマフォ関連の関数が用意されておりそれぞれセマフォの作成、制御、操作を行います。

セマフォの作成・検索(semget)

- 一般形 `int semget(key_t key, int nsems, int semflag);`
- `key`: 同じプロセスからforkで生成されたプロセス間で使用するセマフォか、任意のプロセス間で使用するセマフォかを定める
- 任意のプロセス間で使用するには、セマフォはファイルの一種としてパス名で指定され、各プロセスでsemgetを実行（最初に実行したプロセスが作成し、他は検索）
- `nsems`: セマフォの個数（同じIDで複数個のセマフォを作成できる）
- 同じプロセスからforkされたプロセス間で使用
`int semid=semget(IPC_PRIVATE, 1, 0666);`
- 任意のプロセス間で使用
`key_t key=ftok("/tmp/sem1",1);`
`int semid=semget(key, 1, 0666 | IPC_CREAT);`

semgetの仕様を説明します。

引数として、`key`, `nsems`, `semflag`をとります。

同じプロセスからforkされたプロセス間で使用するときは、単にsemgetの戻り値であるセマフォのIDを共有します。

一方、同じプロセスからforkしたプロセス以外の、任意のプロセス間で使用するには、`key`と呼ばれる引数をあらかじめ特定のファイルに対応付けて、ファイルのパス名をもとに共有します。

セマフォの制御 (semctl)

- 一般形
`int semctl(int semid, int semnum, int cmd, ...);`
- セマフォの初期値設定
 - **semgetではセマフォに初期値を設定できない** (semgetはすべてのプロセスで実行されるため初期値の設定はできない)
 - `semctl(semid, 0, SETVAL, 1);` (0番目のセマフォの初期値を1に設定)
- セマフォの消去
 - 指定されたセマフォ全体を削除 (semnumは無視される)
 - セマフォでブロックしているすべてのプロセスを実行可能にする
`semctl(semid, 0, IPC_RMID, 0);`
- セマフォの値の取得
`semctl(semid, 0, GETVAL, c_arg);` (c_argは値を入れる構造体)

セマフォの制御はsemctlで行います。

これで行える操作は、初期値の設定、セマフォの消去、セマフォの値の取得などです。

セマフォの操作 (semop)

一般形

```
int semop(int semid, struct sembuf *sops, unsigned nsops);
```

- sembuf型の構造体(個数がnsops個)に操作のパラメータを設定して呼び出す
- sembuf型の構造体中のsem_opの値の符号で動作が変わる
 - 値が負の場合 (wait操作):
 - セマフォの値が、sem_opの絶対値以上の時は、セマフォの値を絶対値の数だけ減らす
 - セマフォの値が、sem_opの絶対値未満の時は、絶対値の値以上になるまでブロックし、**値以上になったとき絶対値の数だけ減らす**
 - 値が正の場合 (signal操作)
 - その値をセマフォの値に加える

セマフォの操作はsemopにより行います。

引数の中のsem_opの値が負であればwait操作、正であればsignal操作となります。

なお、以前の講義で説明したwait操作とは違い、UNIXのsemopでは、wait操作でセマフォの値を減らせずにブロックした後に、別プロセスによるsignal操作でウェイクアップするのではなく、自らセマフォの値の増加を検出してウェイクアップします。また、ウェイクアップした後、セマフォの値を引数の絶対値分だけ減らします。

これに対して、signal操作では、ブロックしているプロセスがあってもウェイクアップせずに単純にセマフォの値を引数分だけ増やします。

2進セマフォの例

- wait操作

```
struct sembuf sb;  
sb.sem_num = 0;  
sb.sem_op = -1;  
sb.sem_flg = 0;  
semop(sid, &sb, 1);
```

- signal操作

上のsb.sem_op = -1;をsb.sem_op = 1;に換えるだけ

UNIXのsemopで、実際にwait操作とsignal操作を2進セマフォとして行う例を示します。

UNIXとWindows NTでのプロセスの実装

- UNIX
 - 情報科学の研究の共通基盤として開発
 - OSの新機能の試験環境として利用されてきた
 - UNIX互換OS (Linuxなど) が無償で利用できる
 - 後から追加した機能が多い (仮想記憶、スレッド、GUI など)
- Windows NT
 - 商用OSとして当初から多数の機能を盛り込む
 - GUIの機能をカーネル内に入れている (応答性重視)

UNIXとWindows NTでのプロセスの実装について説明します。

UNIXは、元々、情報科学の研究の共通基盤として開発され、OSの新機能の試験環境として利用されてきました。

このため、UNIX互換OS (Linuxなど) が無償で利用できるようになっています。研究用のOSであるため、後から追加した機能 (仮想記憶、スレッド、GUI など) が多いです。

これに対して、Windows NTは、商用OSとして当初から多数の機能を盛り込んでいます。

また、GUIの機能をカーネル内に入れているなど、応答性重視の設計となっています。

UNIXの概要

- 1969年誕生 当初はunics(multicsへの対抗)
- AT&Tベル研究所で開発(DEC社のPDP-7)
 - Ken Thompson, Dennis Ritchie
- メーカー以外で開発されたはじめての実用OS
 - 特定のメーカーのハードウェアに縛られない
 - ユーザの視点に立った設計
 - 使い勝手の良さから注目される
- 1973年にPDP-11に移植
 - C言語を用いて書きなおす
 - 移植性、可読性の向上

UNIXの概要はこのスライドに上げたとおりです。

1969年誕生し、初はunicsと表記していました。名前の由来は、当時開発中だった巨大なOSであるmulticsに対抗する意味があったと言われています。

AT&Tベル研究所でDEC社のPDP-7というコンピュータシステム上で回ハウされました。

開発者はKen Thompson, Dennis Ritchieです。

メーカー以外で開発されたはじめての実用OSだと言えます。

当時は、OSはコンピュータメーカーがそれぞれ独自に開発していましたが、UNIXは特定のメーカーのハードウェアに縛られないという優れた点がありました。

ユーザの視点に立った設計と使い勝手の良さから注目されるようになりました。

1973年にPDP-11というさらに性能の高いコンピュータシステムに移植されました。

OSをC言語を用いて書きなおしたため、Cコンパイラがあれば移植が可能となることから、移植性、可読性が向上しました。

商用UNIXの開発

- ベル研究所での研究
 - 7th→8th→Plan9→(ルーセント社へ)
- 32ビットUNIX
 - 32V・・・仮想記憶はない
- 商用UNIX
 - System III→System V Release 1
 - System V R2, 3, 4, 4.1, 4.2, 4.2MP
- Novell社 UnixWare
- The SCO GroupがUNIXの著作権を主張
 - Novell社から著作権を譲り受けて所有していると主張し、IBMなど自社計算機にLinuxを使用して販売しているメーカーを著作権侵害で提訴中(2007年8月に著作権譲渡が完全には行われていなかったという判決が下る)

このように、UNIXは元々は研究用のOSでしたが、使い勝手の良さから普及するにつれて、商用化の動きが出てきました。

ベル研究所で研究用に改良が進められてきたバージョンは、ルーセントという会社に継承されました。

一方で、これとは別に、商用UNIXの開発が始まり、System V(ファイブ)が開発されました。

UNIXの著作権は、ベル研究所の親会社であったAT&Tから、Novell社を経て、The SCO Groupという組織が譲り受けたと主張しました。

IBMなどのいくつかの会社は、自社計算機にLinuxを使用して販売していましたが、The SCO GroupはUNIXの著作権を主張し、著作権侵でIBMなどを提訴するという事件に発展しました。

この訴訟では、2007年8月に、著作権の移譲が完全には行われていなかったという判決が出たため、いったんは収まりましたが、正当な著作権を手に入れた場合は再度訴えられる危険性があります。

UCBでのUNIX開発

UCB(カリフォルニア大学バークレー校)

- Version6 UNIXを改良
 - 1BSD として配布
- Version7, 32V UNIXを基盤として
 - 3BSD: 各種ユティリティ、コンパイラなど
 - 4BSD: 完全なOS
 - 仮想記憶
 - 4.2BSD TCP/IPネットワーキング
 - 4.2BSDが急速に普及

研究用のUNIX開発で、ベル研究所と別の流れとして、UCB(カリフォルニア大学バークレー校)で開発されたBSDがあります。

BSDはバークレー・ソフトウェア・ディストリビューションの略で、ベル研究所のVersion 6, Version 7, 32V UNIXをベースに独自の改良を加えて、仮想記憶やTCP/IPのネットワーク機能を備えた完全なOSとなりました。

大学が作った完全かつ実用のOSとして定評がありました。

UCBの動き

- 4.xBSDのソースを精査・・・1990年代
 - AT&T UNIX由来のコードを含むものを識別
- AT&T由来以外のソースを公開
- 作業をさらに続ける
- USLとの訴訟・・・和解
- 配布自由な 4.4BSD-Liteを発表(1993)
 - AT&T UNIX由来のソースファイルが5個不足しているが、これを補えば、実際に動作するフリーのUNIXができる
 - NetBSD, FreeBSD, OpenBSDの基礎になる

UCBはその後、1990年代になって、4BSDのソースコードを精査し、AT&T(ベル研)由来以外の部分のソースコードを公開しました。

また、前述のUNIX訴訟においても、著作権を持っていたUSLとの訴訟で和解に至り、1993年に配布自由な4.4BSD-Liteを発表しました。

これは、まだ完全な公開ではなく、AT&T UNIX由来であるため公開できないソースファイルが5個不足していましたが、逆に言えば、その部分さえ独自に開発できれば実際に動作するフリーのUNIXができる道が開かれました。

これが、現在のフリーなBSD系UNIXである、NetBSD, FreeBSD, OpenBSDの基礎となっています。

UNIXの標準化

- 米国電気電子学会(IEEE)を中心
 - POSIX (Portable Operating System Interface) と呼ばれる
- UNIX系OSは皆POSIX準拠に
- UNIXの標準化
 - 一本化して The Open Groupが管理
 - UNIXの規格を制定(UNIX95, UNIX98, 他)

さらに、情報科学の分野での標準化で大きな力を持つ、IEEE(米国電気電子学会)を中心にUNIXの標準化が行われました。

この標準は、POSIX (Portable Operating System Interface) と呼ばれ、UNIX系OSは皆POSIX準拠となり、標準化に大きく貢献しました。

30

UNIXライクOS

- MINIX
 - A. S. Tanenbaum 教授 作
 - V7相当の機能、OS教育用
 - 内部構造は独自
- Linux
 - Linus Torvalds 作 (カーネルのみ)
 - 単層カーネル方式で設計されている
 - 商用OSも始まる(RedHat, SuSEなど)

前述のUNIXの著作権訴訟問題から、UNIXという名前は使わないが、実際にはUNIXと互換性があり、同じように使えるOSが登場しました。

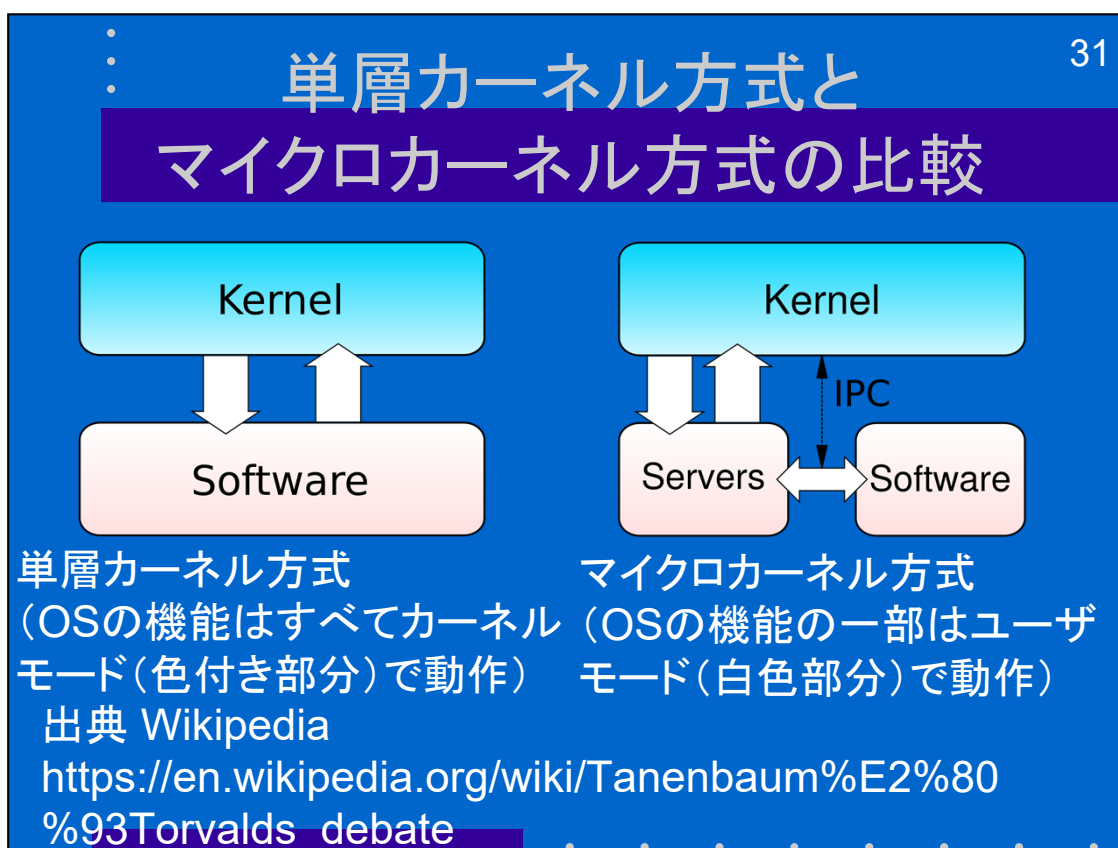
これらは、UNIXライクOSと呼ばれ、代表的なものにMINIXとLinuxがあります。

MINIXはOSの教科書の執筆者として有名な、Tanenbaum 教授が開発した教育用のOSです。

V7 UNIX相当の機能を持っていますが、OSの中身は独自に作成しているため、UNIXの著作権の問題がありません。

さらに有名なLinuxもUNIXライクOSの一つです。Linus Torvalds が独自に作成したカーネルをベースにしており、単層カーネル方式で設計されています。

Linuxは広く普及し、RedHatやSuSEといった商用Linuxも登場するようになりました。



ここで、以前にも説明した、単層カーネル方式とマイクロカーネル方式を復習しておきます。

英語版のWikipediaでは両者をこのスライドにあるような図で説明しています。それによると、単層カーネル方式では、OSの機能はすべてカーネルモード(色のついた部分)で動作するのに対して、マイクロカーネル方式ではOSの機能の一部はユーザモード(Serversと書かれた白い部分)で動作します。OSの実装と、カーネルの構成方式との関係については、後で説明します。

UNIXの特徴 (1)

- 単純性
 - 構造が単純なだけでなく、可能なかぎり小さくすること
- 移植性
 - 実行効率よりも移植のしやすさを選ぶ(アセンブリ言語よりもできる限り高級言語(C言語)でカーネルを記述)
- ツールキットアプローチ
 - 全体が統合された環境を目指すよりは、単一の機能しかない小さなツールを組み合わせで複合的な機能を作り上げる

UNIXの特徴には、まず、このスライドであげたようなものがあります。

単純性として、構造が単純なだけでなく、可能なかぎり小さくすることです。

移植性として、実行効率よりも移植のしやすさを選ぶこと、つまり、実行効率の良いアセンブリ言語よりも、移植性の高い高級言語(UNIXの場合はC言語)でカーネルを記述したことです。

ツールキットアプローチとして、全体が統合された環境を目指すよりは、単一の機能しかない小さなツールを組み合わせで複合的な機能を作り上げることです。

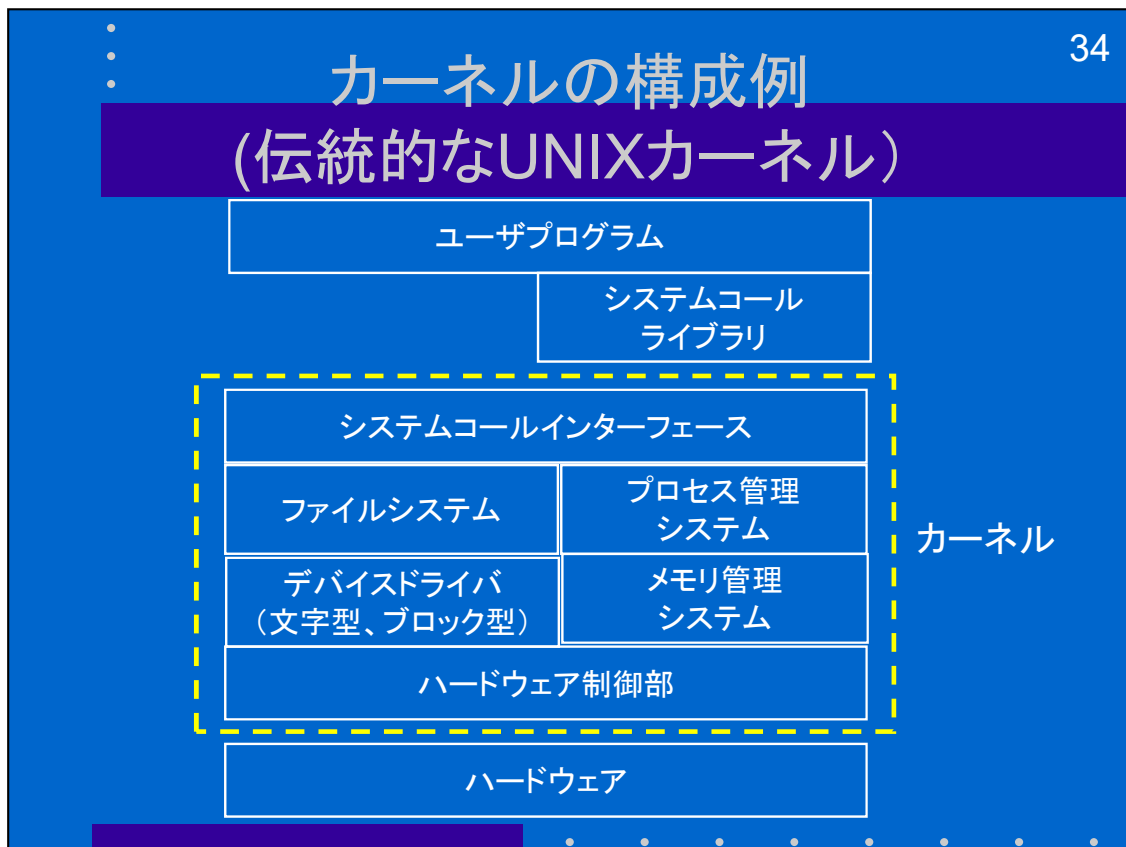
UNIXの特徴 (2)

- シンプルなファイルシステム
 - バイトストリーム
- 階層ファイルシステム
 - 個々のディスク上のファイルシステムを接合
- 「ファイル」でハードウェア資源を隠ぺい
 - 入出力装置、プロセス間通信、通常のファイルなど、あらゆる資源を単一の方法で扱える
 - ファイル入出力、プロセス間通信、システムコールを使い分ける必要がない
- 豊富なプログラミング言語、ツール
 - ツール=コマンド=ユーティリティプログラム

さらに、入出力などあらゆる資源を単一の方法で扱えるようになっています。

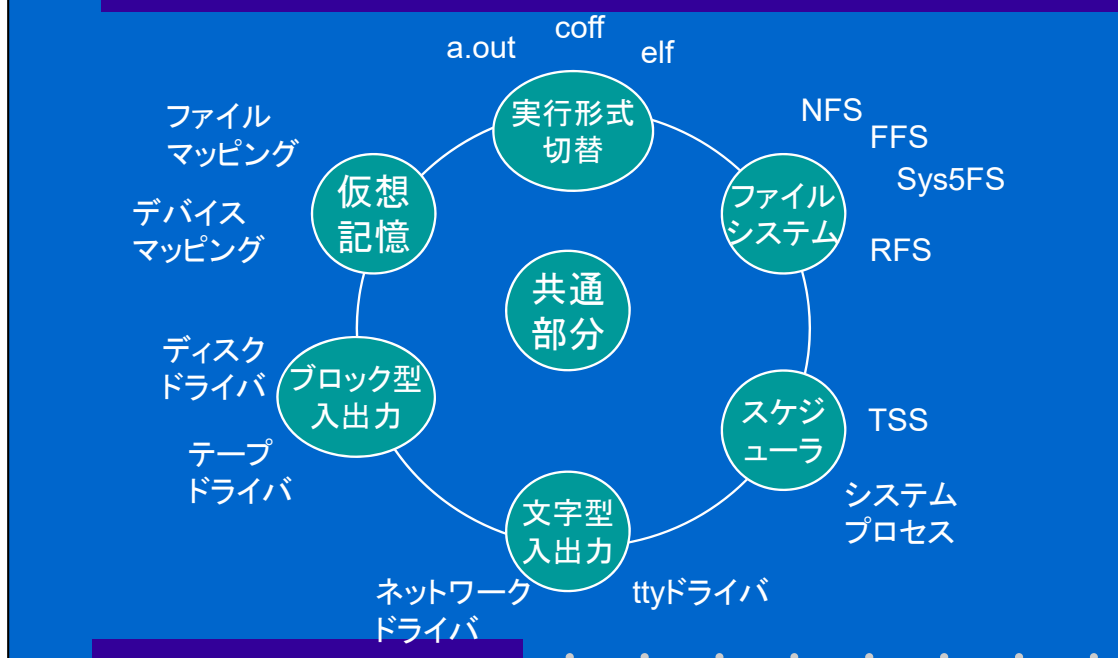
また、豊富なプログラミング言語をサポートし、多数のツールを備えています。

これらのツールは、自由に組み合わせて使えるのも大きな特徴です。



以前に示した、このスライドのようなカーネルの構成は、伝統的なUNIXカーネルの内部構成を示していました。

最近のUNIXカーネルの構成例



最近のUNIXカーネルの内部構成は、はるかに複雑になっており、伝統的なカーネル構成にある積み上げ型の構成ではなく、多数のモジュールが相互につながっている構成になっています。

UNIXでのプロセスの実装

- プロセスの作成とプログラムの実行を分離
 - プロセスの作成: fork
 - プログラムの実行: exec
- 利点
 - シンプルなシステムコールで済む(自分の複製を作って、それがプログラムを実行する)
 - プロセス生成時の初期化が楽になる(親プロセスの領域が複製されているので、変更箇所のみ変えればよい)

UNIXでのプロセスの実装は、先に説明したように、forkとexecで行われます。先には、プログラム例として動作を示しましたが、ここではさらにカーネル内の動作に踏み込んで説明します。

プロセス生成時の動作

forkの実行

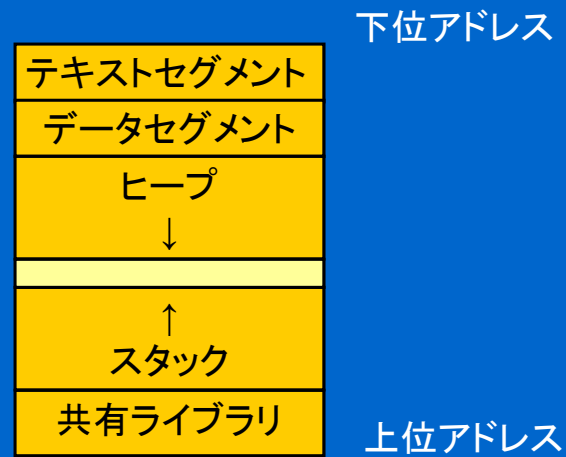
1. 子プロセスのためのプロセステーブルの空きエントリを確保し、親のプロセステーブルのエントリの内容をコピーする(子のプロセスIDと親のプロセスIDをセットし直す)
2. スタックとデータセグメントの領域(ユーザテーブルを含む)を確保し、その内容を親からコピーする
3. プロセステーブルのエントリのポインタを新しい領域を指すように変更する。テキストセグメントは共有し、スタックとデータセグメントは独立した領域となる。

execの実行

- テキストとデータのセグメントの領域を新たに確保し、プロセステーブルのエントリのポインタを変更する。

カーネルでのforkとexecの実行は、このスライドで示す手順で行われます。
実際には、プロセス領域に関連して多くの処理が行われていることがわかります。

(参考)UNIXのプロセス領域



前のスライドにある、UNIXのプロセス領域はこのスライドに示すような配置になっています。

UNIXでのプロセスのスケジューリング

プロセスの優先度に基づいて行われる

- 優先度は整数で表され、小さい値の方が優先度が高い(ユーザモードで実行しているプロセスの優先度は正の値、カーネルモードでは負の値)
- 優先度ごとに実行可能なプロセスのキューが用意されている(図5.10)
- 優先度の最も高いプロセスのキューの先頭にあるものが順に実行される
- 実行プロセスは、入出力待ちなどでブロックされるか、あるいはタイムスライス(100ミリ秒程度)が経過するまで連続して実行できる

すべてのプロセスの優先度はプロセッサ時間の消費量に応じて再計算される

- プロセッサを長時間割り付けられなかったプロセスの優先度は値が小さく(高く)なり、最近プロセッサを割り付けられたプロセスの優先度は値が大きく(低く)なる

UNIXでのプロセスのスケジューリングは、実際にはこのスライドで示す手順で処理されます。

(参考)タイムスライスの決め方

- **タイムスライスの時間を短くすると、**
 - プロセスの切替えが頻繁に起こり、他のプロセスが実行し続けることによる**待ち時間**が減る(**応答時間**が小さくなる)
 - 短くしすぎると、プロセススイッチばかりに時間が取られて、プロセスの実行効率が悪くなる
- **タイムスライスの時間を長くすると、**
 - プロセススイッチがまれにしか起こらなくなり、プロセスの実行効率が上がる
 - 一度プロセスが中断されると、次に実行が再開されるまでに長時間待たされることになる(**待ち時間**、**応答時間**が大きくなる)
- **プロセスによって適切なタイムスライスの値が異なる**
 - 対話的な処理では短くして応答時間を削減(応答性能重視)
 - 計算主体の処理では大きくして実行効率を上げる
 - UNIXでは、10ms程度の基準時間(timer tick)ごとにプロセスの実行状況を監視して、タイムスライスの値を調整している

以前にも同じ説明がありましたが、UNIXでのタイムスライスはこのスライドで説明するような手順で決まります。

41

Windows NT

- マイクロソフトが開発した32ビットOS
 - ユーザーインターフェースは Windows 3.1, Windows 95等と同じ
 - マルチユーザ・マルチタスク
 - 互換性、信頼性、移植性など
- Windows NT3.5, NT4.0, 2000, XP, Vista, 7, 8, 10
- Windows 9x系列からの移行
 - 95, 98, Meを、XPで統合へ

別のOSとして、Windows NTを取り上げます。

Windows NTはマイクロソフトが開発した32ビットOSです。

ユーザーインターフェースは、それ以前のWindows 3.1, Windows 95等と同じであり、

マルチユーザ・マルチタスクで、互換性、信頼性、移植性を持つ本格的なOSです。

Windows NT3.5, NT4.0, 2000, XP, Vista, 7, 8, 10という系列で発展しており、マイクロソフトのそれまでのWindows 9x系列から移行し、95, 98, Meの系列を、XPで統合しました。

Windows (1)

- IBMのパーソナルコンピュータ IBMPC向け
- OSはマイクロソフトが担当
 - MS-DOS 1.0
 - マイクロソフトにとっての初めてのOS製品
- MS-DOS 2.0
 - UNIX を意識した拡張(ファイルシステムのディレクトリ構造など)

Windows NT以前のマイクロソフトのOSについても説明しておきます。

マイクロソフトのOSは、元はIBMのパーソナルコンピュータ IBMPC向けであり、IBMPCのOSをマイクロソフトが担当して開発しました。

MS-DOS 1.0はマイクロソフトにとっての初めてのOS製品であり、MS-DOS 2.0になるとUNIX を意識した拡張(ファイルシステムのディレクトリ構造など)が行われました。

Windows (2)

- プロセッサの進歩につれWindowsも進歩
 - 実用レベルのものはなかなか出なかった
- IBMと共同でOS/2を開発
 - 80286(16ビットCPU)以上で動作
 - その後、協力関係を破棄
- Windows 386
 - 80386(32ビット) 以上で動作
 - かなり使い物になるようになってきた

プロセッサの進歩につれWindowsも進歩しましたが、実用レベルのものはなかなか出ませんでした。

IBMと共同でOS/2を開発し、80286(16ビットCPU)以上で動作しましたが、その後、協力関係を破棄し立ち消えとなりました。

Windows 386を開発し、80386(32ビット) 以上で動作して、かなり使い物になるようになってきました。

Windows(3)

- Windows 3.0 → Windows 3.1 → Windows 95 → Windows 98 → Windows Me
 - MS-DOSの上に32bit APIとGUIを載せたもの
 - マルチタスク機能貧弱、シングルユーザ
- 独自のOS開発を決意
 - 1988年
 - David CutlerをDECから引き抜く
 - VMSの開発者
- Windows NT (New Technology)と名付ける

さらに、Windows 3.0 → Windows 3.1 → Windows 95 → Windows 98 → Windows Meという、MS-DOSの上に32bit APIとGUIを載せたOSを開発しましたが、いずれもマルチタスク機能貧弱でしたし、シングルユーザでしか使えませんでした。

そこで、マイクロソフトは独自のOS開発を決意し、1988年にDavid Cutler(VMSという本格的なOSの開発者)をDECから引き抜いて開発に着手させました、こうして生まれたOSに対して、Windows NT (New Technology)という名前がつけられました。

Windows NT

- マイクロカーネル方式
- 1993年 Windows NT 3.1 発表
 - Windows 3.1と同じGUI
- 1994年 Windows NT 3.5 発表
- 1996年 Windows NT 4.0 発表
 - Windows 95と同じGUI
 - GUIルーチンや画面描画などいくつかの機能をカーネルに移動(純粋なマイクロカーネル方式ではない)

Windows NTが、それまでのUNIXなどと比べた特徴としては、マイクロカーネル方式を採用したことです。

1993年 Windows NT 3.1 が発表され、Windows 3.1と同じGUIを持っていました。

1994年 Windows NT 3.5 が発表され、1996年 Windows NT 4.0 発表されました。

Windows NT 4.0は、Windows 95と同じGUIを持っていました。

3.5と比べると、GUIルーチンや画面描画などいくつかの機能が3.5までは全部カーネルの外に出されていたものを、カーネル内に移動させました。

つまり、4.0からは純粋なマイクロカーネル方式ではなくなっています。

Windows NTの開発物語

G パスカル ザカリー著、山岡洋一訳、「闘うプログラマービル・ゲイツの野望を担った男達」

日経BP社、2009

原書の題名: **Show-stopper!**: Breakneck Race to Create Windows NT and the Next Generation at Microsoft, Little, Brown & Company, 1994

show-stopperの辞書の意味

[ショーが一時中断されるほどの]拍手喝さいの名演技
[名演奏・せりふ]、人目を引く物[人]

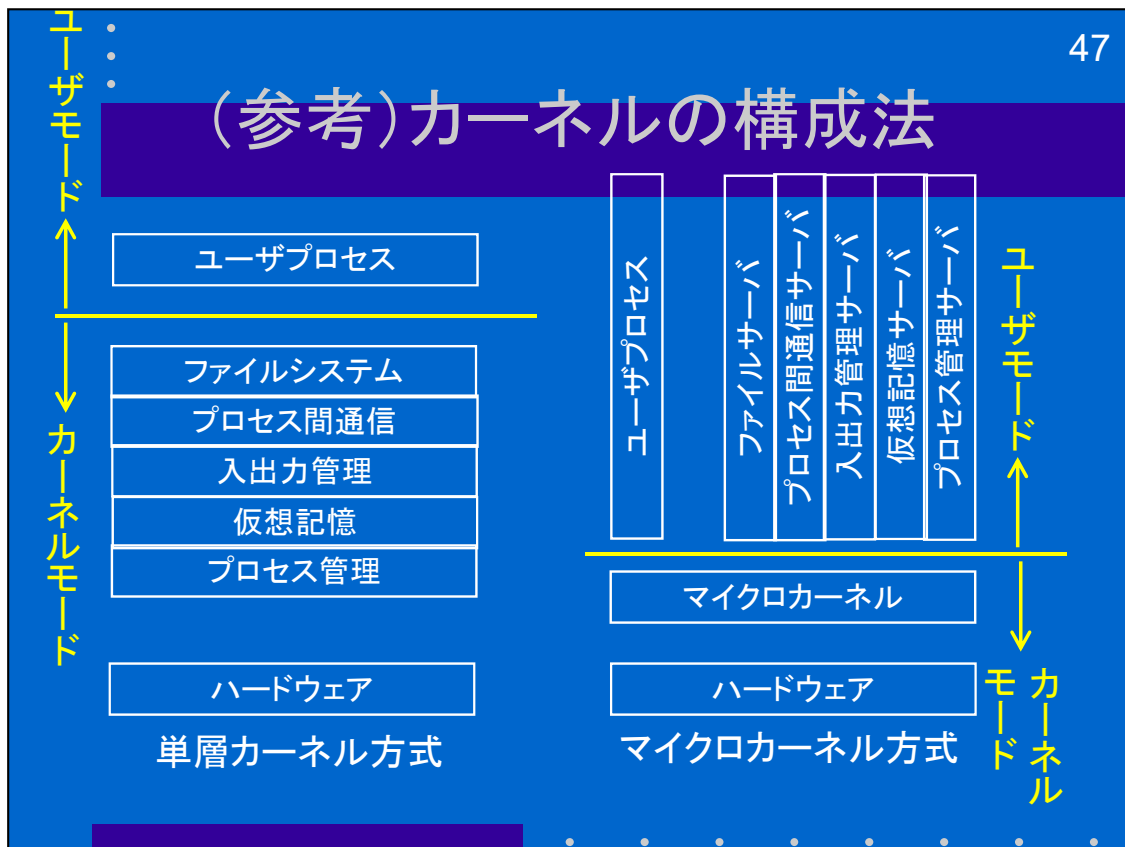
〈俗〉コンピューターの動作を停止させるバグ、[ソフトウェアなどの]致命的問題◆演劇・音楽などでの用法と正反対に「最悪」「まるで駄目」を意味する。

ここでWindows NTの開発物語が書かれた本を紹介します。

海外で出版された原書を翻訳した本ですが、原書のタイトルと比べると、日本語訳の本のタイトルはかなり攻撃的に意識していると考えられます。

なお、原書のタイトルもなかなか興味深いです。show-stopperという単語は、一般には良い意味で使われますが、コンピュータ業界では逆にかなりの問題を含む意味で使われます。

この本ではどちらの意味で使われているか知りたい人は、この本を購入して読むことをお勧めします。



以前に述べたように、マイクロカーネル方式では、割り込み処理等の基本的なカーネルの機能以外はすべてシステムサーバプロセスに移して、ユーザモードで動作させるのが特徴です。

48

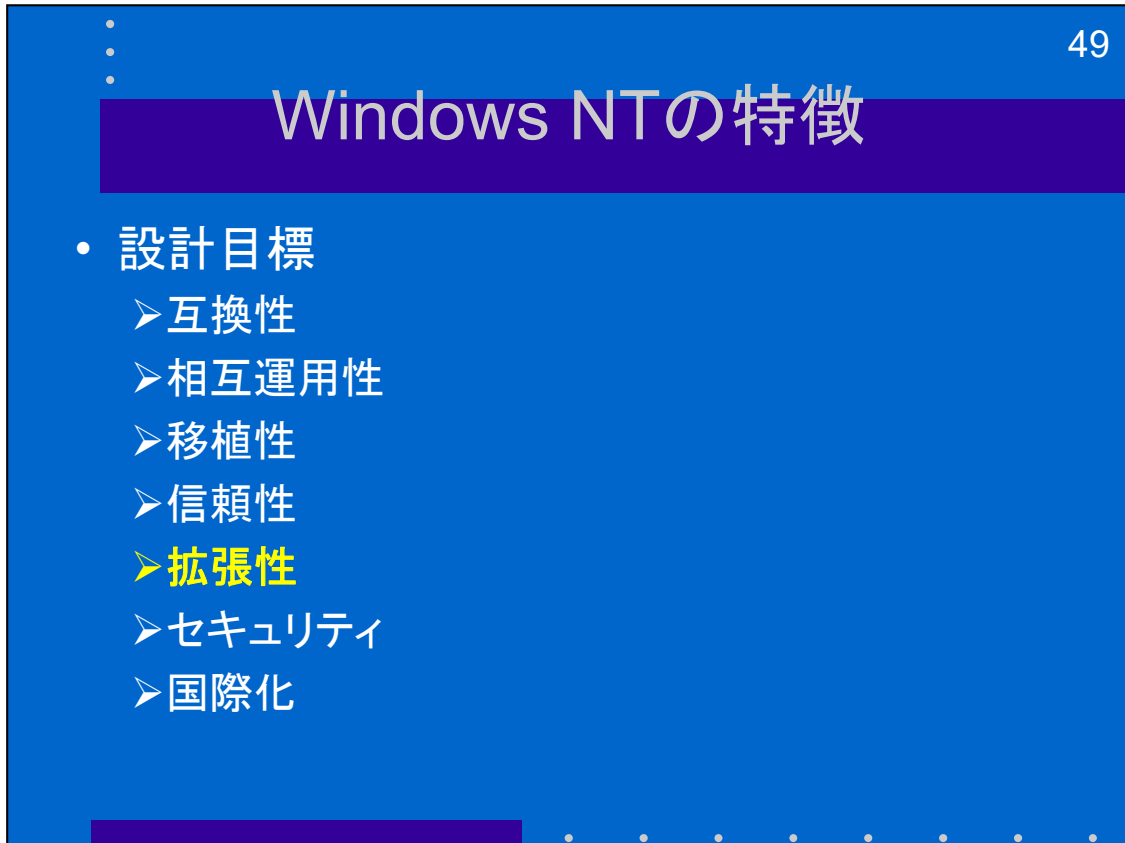
Windowsの内部バージョン

- Windows 2000
 - 内部バージョン(カーネルなどの基本部分)は**Windows NT 5.0**
- Windows XP
 - 内部バージョンは**Windows NT 5.1**
- Windows Vista
 - 内部バージョンは**Windows NT 6.0**
- Windows 7
 - 内部バージョンは**Windows NT 6.1**
- Windows 8 → 8.1
 - 内部バージョンは**Windows NT 6.2 → 6.3**
- Windows 10
 - 内部バージョンは当初は**Windows NT 6.4**だったが、現在は**10.0**と表示される

Windows NTと聞くと、以前のOSの名前であり、今のWindowsとは関係ないと思うかも知れません。

しかし、実は、Windows NT以降のWindowsで始まる名前を持つOSは、内部バージョンと呼ばれるカーネルなどの基本部分は、実はWindows NTなのです。このスライドにあるように、Windows 2000から始まって、最新のWindows 10まで、内部バージョンはすべてWindows NTとなっています。

(自分が今使っているWindowsの内部バージョンを知りたいければ、コマンドプロンプトを実行してみてください。内部バージョンが表示されます。)



49

Windows NTの特徴

- 設計目標
 - 互換性
 - 相互運用性
 - 移植性
 - 信頼性
 - **拡張性**
 - セキュリティ
 - 国際化

Windows NTの特徴としては、まず設計目標にここにあげている多くの特性をあげています。

⋮

50

互換性

- NT 3.5
 - Windows 3.1と同じGUI
- NT 4.0
 - Windows 95と同じGUI
- プログラムが移植しやすい環境
 - MS-DOS, OS/2, POSIX
 - 環境サブシステム
 - サーバはユーザプロセス

まず、互換性ですが、Windows NTはそれ以前のマイクロソフトのOSと同じGUIを備えるとともに、プログラムを移植しやすいような環境を提供しています。

相互運用性

- 高機能API (RPC, winsock)
- 各種プロトコル
 - TCP/IP, NetBEUI, IPX/SPX, Appletalk
- さまざまな環境へ接続可能
- 現実には相互運用性は低い
 - 特に異なるOSのサーバとの接続が困難

相互運用性については、高機能APIを備え、多様な通信プロトコルに対応していることとされています。

しかし、現実には相互運用性はそれほど高くはなく、特に異なるOSのサーバとの接続が困難という欠点が指摘されています。

移植性

- HAL (Hardware Abstraction Layer)
 - ハードウェアの違いを吸収
- 高級言語で記述
- 複数のCPUファミリーに対応
 - IA32, MIPS Rx000, IBM PowerPC, DEC Alpha
- 現在では（基本的に）IA32
 - Server Editionで、ItaniumやAMD64をサポート
 - x64 Editionで、AMD64, Intel EM64Tをサポート

移植性についても、ハードウェアの使いを吸収するHAL (Hardware Abstraction Layer)を備えています。

HALは高級言語で記述されており、多様なCPUファミリーに対応していることになっています。

しかし、現在では、インテルCPUの基本アーキテクチャであるIA32にのみ対応していることが多く、別のCPUファミリーへの対応が課題となっています。

信頼性

- カーネルモードとユーザーモードの分離
 - ユーザアプリケーションはカーネルに干渉できない
- ユーザプロセス相互の分離
- カーネルモードのプログラム
 - モジュールに分割
 - 階層構成

信頼性については、マイクロカーネル方式を採用して、カーネルモードとユーザーモードを分離し、ユーザアプリケーションはカーネルに干渉できない点があげられます。

また、ユーザプロセスを相互に分離する、カーネルモードのプログラムはモジュールに分割され、階層構成を取るなど、信頼性を上げるための工夫がされています。

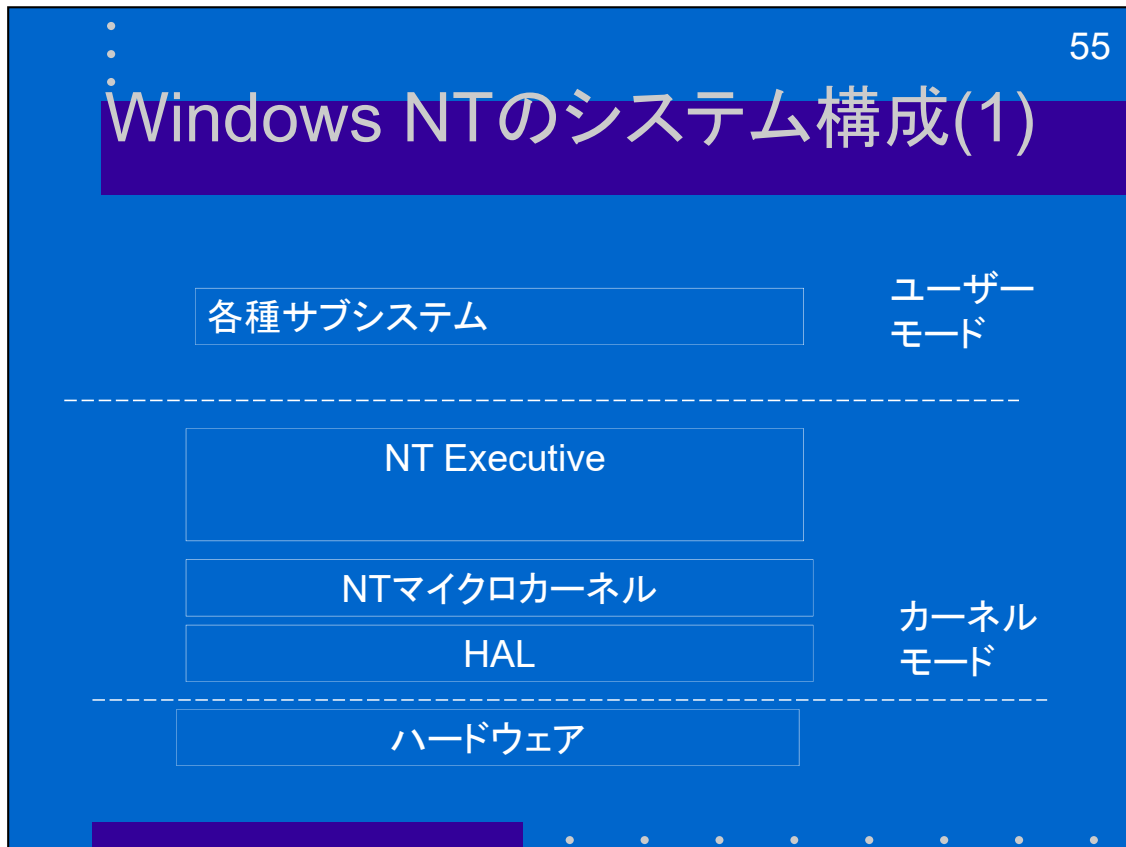
拡張性

- モジュール化
 - 機能追加が容易
 - カーネルモードで機能の追加
 - ユーザモードで機能の追加
- モジュールを追加ロードすればよい
 - APIが増える/変わる

拡張性については、モジュール化が進められたため、機能追加が容易となっています。

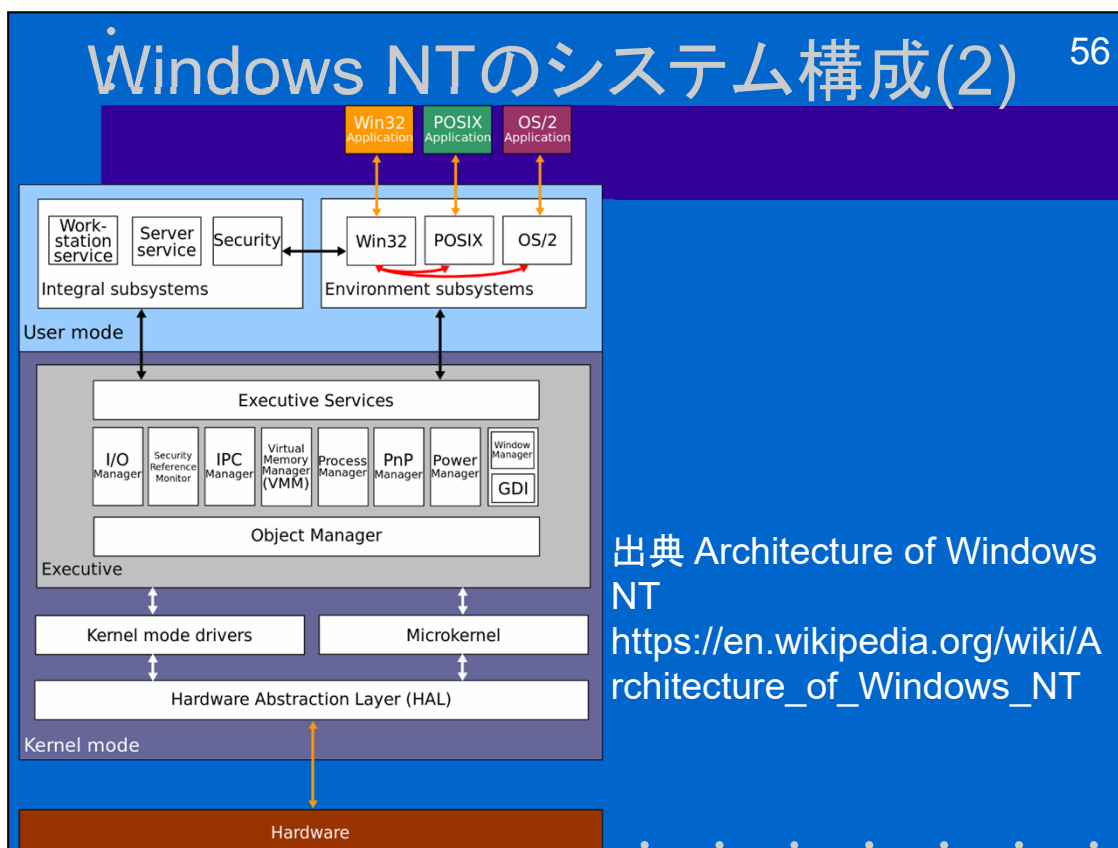
具体的には、モジュールを追加ロードすればよいことになっています。

実際には、モジュールを追加すると、APIが増える/変わる可能性があります。



Windows NTのシステム構成です。

ハードウェアを隠ぺいするHAL(Hardware Abstraction Layer)、NTマイクロカーネル、マイクロカーネル以外でカーネルモードで動作する機能进行处理するNT Executiveがカーネルモードで動作し、それ以外の機能をユーザモードで処理する各種サブシステムがあります。



前のスライドの構成をより詳細にみたのがこの図です。
 それぞれについては、以降のスライドで説明します。

Windows NTシステムの構成

- カーネルモード
 - NT Executiveが動作する
- ユーザモード
 - サブシステムが動作する
- NT Executive
 - NTシステムサービス
 - 機能モジュール群
- サーバとクライアント(ワークステーション)のモデルがある

Windows NTでHALとNTマイクロカーネル以外の機能は、NT Executiveとサブシステムで分担されます。

詳細は以降のスライドで説明します。

NT Executive

- オブジェクトマネージャ
- LPC(Local Procedure Call)機能
- セキュリティ参照モニタ
- プロセスマネージャ
- 仮想メモリマネージャ
- I/Oシステム
- **GDI (Graphical Device Interface)**
 - NT3.5まではユーザモードで動作していたが、NT4.0からは性能向上のためカーネルモードで動作させるようになった

NT Executiveについての説明です。

このスライドにあげられているような機能がNT Executiveで処理されます。

特徴としては、GUIの機能の一部を担当するGDI (Graphical Device Interface)です。

NT 3.5まではユーザモードでサブシステムとして動作していましたが、NT 4.0からはNT Executiveの中でカーネルモードで動作するようになりました。

これは、ユーザモードで動作させると、カーネルとの間でプロセス間通信が必要となり性能が低下するため、応答性を重視する設計思想からカーネルモードで動作させるように変更となりました。

NT マイクロカーネル

- スレッドのスケジューリングとディスパッチ
- 割込み処理とディスパッチ
- 例外処理とディスパッチ
- マルチプロセッサの同期
- カーネルオブジェクトの提供
 - ディスパッチャオブジェクト
 - コントロールオブジェクト
- マルチプロセッサ(マルチコア)に対応

NT マイクロカーネルの機能の一覧です。

カーネルスレッドの機能を提供しており、マルチプロセッサ(マルチコア)に対応しています。

⋮

Windows NTでのプロセスの生成

- 新しいプロセスはspawnとexecの2種類の関数で生成
- exec関数: 新しいプロセスの領域は、呼出し元のプロセスの領域を上書きし、呼出し元のプロセスは消滅する
- spawn関数: 新しいプロセスと、呼出し元のプロセスの両方がメモリ中に存在することが可能
 - `_P_OVERLAY`: 親プロセスを子プロセスでオーバーレイし、親プロセスを破壊する (exec 関数の呼び出しと同じ結果)
 - `_P_WAIT`: 新しいプロセスが終了するまで、呼出し元のスレッドを一時停止する (同期_spawn 関数)
 - `_P_NOWAIT` または `_P_NOWAITO`: 子プロセスと並行して親プロセスを実行する (非同期_spawn 関数)
 - `_P_DETACH`: 親プロセスの実行を継続し、子プロセスはバックグラウンドで実行する (キーボードからアクセス不能)

http://www.microsoft.com/JAPAN/developer/library/vccore/_crt_process_and_environment_control.htm

Windows NTでのプロセスの生成について説明します。

UNIXのfork, execに対して、spawnとexecの2種類の関数が用意されています。

spawn関数は、UNIXのfork関数よりも多くの機能を持っています。

Windows NTでのプロセスの生成(2)

	ファイルサーチに PATH変数を使うか	引数の渡し方	環境設定
_execl、_spawnl	×	引数並び	親プロセスから継承
_execle、_spawnle	×	引数並び	環境テーブルのポインタを引数で渡す
_execlp、_spawnlp	○	引数並び	親プロセスから継承
_execlep、 _spawnlpe	○	引数並び	環境テーブルのポインタを引数で渡す
_execv、_spawnv	×	配列	親プロセスから継承
_execve、 _spawnve	×	配列	環境テーブルのポインタを引数で渡す
_execvp、 _spawnvp	○	配列	親プロセスから継承
_execvpe、 _spawnvpe	○	配列	環境テーブルのポインタを引数で渡す

Windows NTでのプロセスの生成に関連した機能を表にまとめるとこのようになります。

多様な機能が提供されています。

Windows NTにおけるスケジューリング

- 多重フィードバック(優先度順とラウンドロビンの組合せ)
- 優先度の値は0から31までの整数値(値が大きいほど優先度が高い)
- 対話的な(ユーザーとのやりとりを多く行っている)入出力処理の多いプロセスやスレッドの優先度は高くし、計算処理の多いプロセスやスレッドの優先度は低くする(応答性能を重視)
- プロセスとスレッドの優先度がある
 - スレッドの優先度はプロセスの優先度のクラスによって変わる
- 一つのプロセスにおける複数のスレッドを別々のプロセッサ上で実行可能(SMP: Symmetric Multi Processingに対応)

Windows NTでのスケジューリングについて説明します。

優先度順とラウンドロビンの組み合わせた多重フィードバックであり、プロセスとスレッドにそれぞれ優先度があります。

応答性能を重視しており、対話的な(ユーザーとのやりとりを多く行っている)入出力処理の多いプロセスの優先度は高くし、計算処理の多いプロセスの優先度は低くします。

63

macOS

- 2016年にOS X(ten)から名称を変更
- Mach(BSD系のUNIXの一種)をベースにしたOSとして全く新たに作り直した

UNIX → Mach → OPENSTEP → (Mac) OS X → macOS

- macOSのカーネルは、Machを基にした**マイクロカーネル**構成
 - ただし一部のシステムサービスはカーネルに取り込んでいる

macOSについても説明しておきます。

2016年にOS X(ten)から名称を変更しており、Mach(BSD系のUNIXの一種)をベースにしたOSとなっています。

ベースにしたMachはマイクロカーネル方式のOSであったため、macOSもマイクロカーネル方式となっています。

ただし、NT 4.0以降のWindows NTと同様に、一部のシステムサービスはカーネルに取り込まれており、純粋なマイクロカーネル方式ではなくなっています。

UNIXとWindows NTの設計思想

- UNIX
 - 単純性
 - 移植性
 - ツールキットアプローチ
- Windows NT
 - 信頼性(安定性)
 - 拡張性
 - 互換性

UNIXとWindows NTの設計思想についてまとめておきます。

UNIX は最初に開発されてから長い時間が経っており、その設計思想については多くの説がある。

しかし、単純性(構造が単純だけでなく、可能な限り小さくすること)、移植性(実行効率

よりも移植しやすさを選ぶ)、ツールキットアプローチ(単一の機能しかない小さなツールを組み

合わせて複合的な機能を作り上げていく)はほぼ共通していると言われています。

一方、Windows NT の設計思想は、主にビジネス用途に向けた信頼性(安定性)、拡張性(後から機能の追加がしやすい)、互換性(以前のWindows 95 などと同じGUI を持つ)と言われています。

UNIXとWindows NT

	UNIX	Windows NT
カーネルの構成	基本的に 単層カーネル方式 (設計思想のうちの単純性を反映)	マイクロカーネル方式 によるモジュール化(設計思想のうちの信頼性(安定性)、拡張性を反映) 応答性能をあげるために一部のシステムサービスをカーネルモードで実行
スケジューリング	以前は プロセス単位 (今は スレッド単位が多い) 優先度順スケジューリング(多重フィードバックもある)	スレッド単位 の多重フィードバック(対話的なスレッドの優先度を上げる)
スレッドへの対応	対応はまちまち	初めからOSの機能として提供 (ユーザスレッドの数だけカーネルスレッドを作成)

先にあげた設計思想から来る違いですが、UNIXでは、カーネルの構成が基本的に単層カーネル方式となっています。これは、UNIXの設計思想のうちの単純性を反映していると考えられます。

一方、Windows NTではマイクロカーネル方式を基本としていますが、これはWindows NTの設計思想である信頼性(安定性)、拡張性を反映していると考えられます。

なお、Windows NTでは、GUIでのユーザインタフェースを重視する点から、当初よりスレッド(特にカーネルスレッド)を導入しています。応答性能をあげるために一部のシステムサービスをカーネルモードで実行するようになっています。

また、スケジューリングの方式では、Windows NTは応答性能を重視しており、対話的(ユーザとの入出力処理の多い)スレッドの優先度を、計算処理の多いスレッドよりも上げるとことを行います。

比較するとUNIXでは、元々はプロセス単位のスケジューリングとなっており、スレッドへの対応は実装ごとにまちまちとなっています。