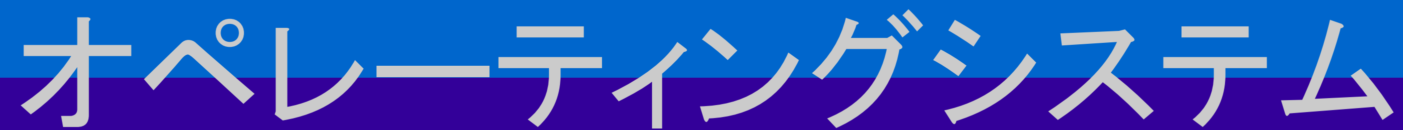


資料　第7分冊(H30)

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

（前回の資料の訂正：下線部分）

クリティカルセクション

- スタックのプッシュ, ポップ操作
  1. 分割して実行されてはならない
  2. 実行するプロセスは一度には1個だけでないといけない
  - プログラム中で2.の条件が必要な部分を、クリティカルセクションという（臨界領域、または際どい部分ということもある）（前回の資料では1.,2.の条件だった）
- **不可分操作**（atomic action）
  - 分割が許されない一連の実行操作
- **相互排除**（mutual exclusion）
  - 一つのプロセスしかクリティカルセクションには入れないようにする

演習問題 No.6 1.の解説

```
/* プロセスP1(生産者) */
i = 1; /* 送信データ */            (1)
wait(s1); /* writeセマフォ */     (2)
buffer = i;  /* 書き込み */        (3)
signal(s2); /* readセマフォ */ (4)
i = i+1; /* 送信データ */          (5)
wait(s1); /* writeセマフォ */     (6)
buffer = i;  /* 書き込み */        (7)
signal(s2); /* readセマフォ */ (8)
```

```
/* プロセスP2(消費者) */
wait(s2); /* readセマフォ */      ( 9)
j = buffer;  /* 読み出し */         (10)
signal(s1); /* writeセマフォ */ (11)
/* jを出力 */
wait(s2); /* readセマフォ */      (12)
j = buffer;  /* 読み出し */         (13)
signal(s1); /* writeセマフォ */ (14)
/* jを出力 */
```

s1, s2はセマフォ(初期値はs1=1, s2=0)であり
bufferはプロセス1とプロセス2で共有されている変数
レディキューの先頭にP1, P2の順につながれている
プログラムの実行はどのようになるか？

演習問題 No.6 1.1)の解答

```
/* プロセスP1(生産者) */
 i = 1;          (1)
wait(s1);        (2)
buffer = i;      (3)
signal(s2);      (4)
i = i+1;         (5)
wait(s1);        (6)
buffer = i;      (7)
signal(s2);      (8)
```

```
/* プロセスP2(消費者) */
wait(s2);        ( 9)
j = buffer;      (10)
signal(s1);      (11)
/* jを出力 */
wait(s2);        (12)
j = buffer;      (13)
signal(s1);      (14)
/* jを出力 */
```

実行	(1)	(2)	(3)	(4)	(5)	(6)	(9)	(10)	(11)	(12)	(7)	(8)	(13)	(14)
s1 = 1		0				block(P1)			restart(P1)					1
s2 = 0				1			0			block(P2)		restart(P2)		

演習問題 No.6 1.2)の解答（(1)のs1,s2をs1のみに変更）

```
/* プロセスP1（生産者）*/
 i = 1;         (1)
wait(s1);      (2)
buffer = i;    (3)
signal(s1);    (4)
i = i+1;       (5)
wait(s1);      (6)
buffer = i;    (7)
signal(s1);    (8)
```

```
/* プロセスP2（消費者）*/
wait(s1);      ( 9)
j = buffer;    (10)
signal(s1);    (11)
/* jを出力 */
wait(s1);      (12)
j = buffer;    (13)
signal(s1);    (14)
/* jを出力 */
```

実行	(1)	(2)	(3)	(4)	(5)	(6)	**(7)**	(8)	(9)	(10)	(11)	(12)	(13)	(14)
s1 =	1	0		1		0		1	0		1	0		1

P1（生産者）の送信データが上書きされ、P2（受信者）で受信できない

演習問題 No.6 1.3)の解答（(1)のP2のs1,s2を入れ替え）

```
/* P1（生産者）*/
 i = 1;          (1)
wait(s1);       (2)
buffer = i;     (3)
signal(s2);     (4)
i = i+1;        (5)
wait(s1);       (6)
buffer = i;     (7)
signal(s2);     (8)
```

```
/* P2（消費者）*/
wait(s1);       ( 9)
j = buffer;     (10)
signal(s2);     (11)
/* jを出力 */
wait(s1);       (12)
j = buffer;     (13)
signal(s2);     (14)
/* jを出力 */
```

実行	(1)	(2)	(3)	(4)	(5)	(6)	(9)
s1 = 1		0				block(P1)	block(P2)
s2 = 0				1			

P1,P2ともにブロックしてデッドロックとなる

演習問題 No.6 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)

演習問題 No.6 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**

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				**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**	Request 0 3 0

資源割り付け後の状態は安全か？

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**

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

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は同じ内容
  - プロセスIDは異なる

forkの使用例(rlogin)

```
If ((pid=fork())==-1) {
        perror("fork");
        exit(1);
}
If (pid==0) {
        while(ネットワークからデータ受信) {
                データを画面に表示
        }
} else {
        while(キーボードからデータ読み取り) {
                ネットワークに送信
        }
}
```

exec

- execv, execve, execl, execleなどの種類がある
- システムコールは一つ、他はライブラリ関数の形で提供されている
- 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する
  - 最初から実行

forkの使用例(リダイレクション)

```
If ((pid=fork())==-1) {
        perror("fork");
        exit(1);
}
If (pid==0) {
        fd=open(ファイル ,・・・・）;
        標準出力を　fdに切りかえる
        exec(プログラム名、引数、・・・);
        perror(“exec failed”);
        exit(1);
}
Wait(&status); /*終了待ち, セマフォのwait操作とは全く
別のものであることに注意 */
```

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(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
- 複数の子プロセスをfork()で生成した場合
  - 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

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
- このコマンドは子プロセスを複数生成した後では、その個数だけ繰り返し呼び出す必要がある

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の模索
  - モニタ、メッセージによる同期

基本同期命令 (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

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**（プロセスの実行が中断されない）
- 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には問題がある
  - 誤って禁止した場合に、システムを保護できない
- 応用
  - 利用したい資源に関連する割り込みだけ禁止

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（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りサポート）

UNIXでのセマフォの実装

- 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）

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決める
- 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);
```

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

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;に変えるだけ

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

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 VR2，3，4，4.1，4.2, 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）

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:　各種ユーティリティ、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

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)
  - 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

UNIXの標準化

- 米国電気電子学会(IEEE)を中心
  - POSIX
  - Portable Operating System Interface
- 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)

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
- 非UNIX OS上でのUNIX互換環境
  - EUNICE(VMS)
- Linux
  - Linus Torvalds　作 (カーネルのみ)
  - MINIXの非実用性に不満
  - 商用OSも始まる(RedHat, SuSEなど)

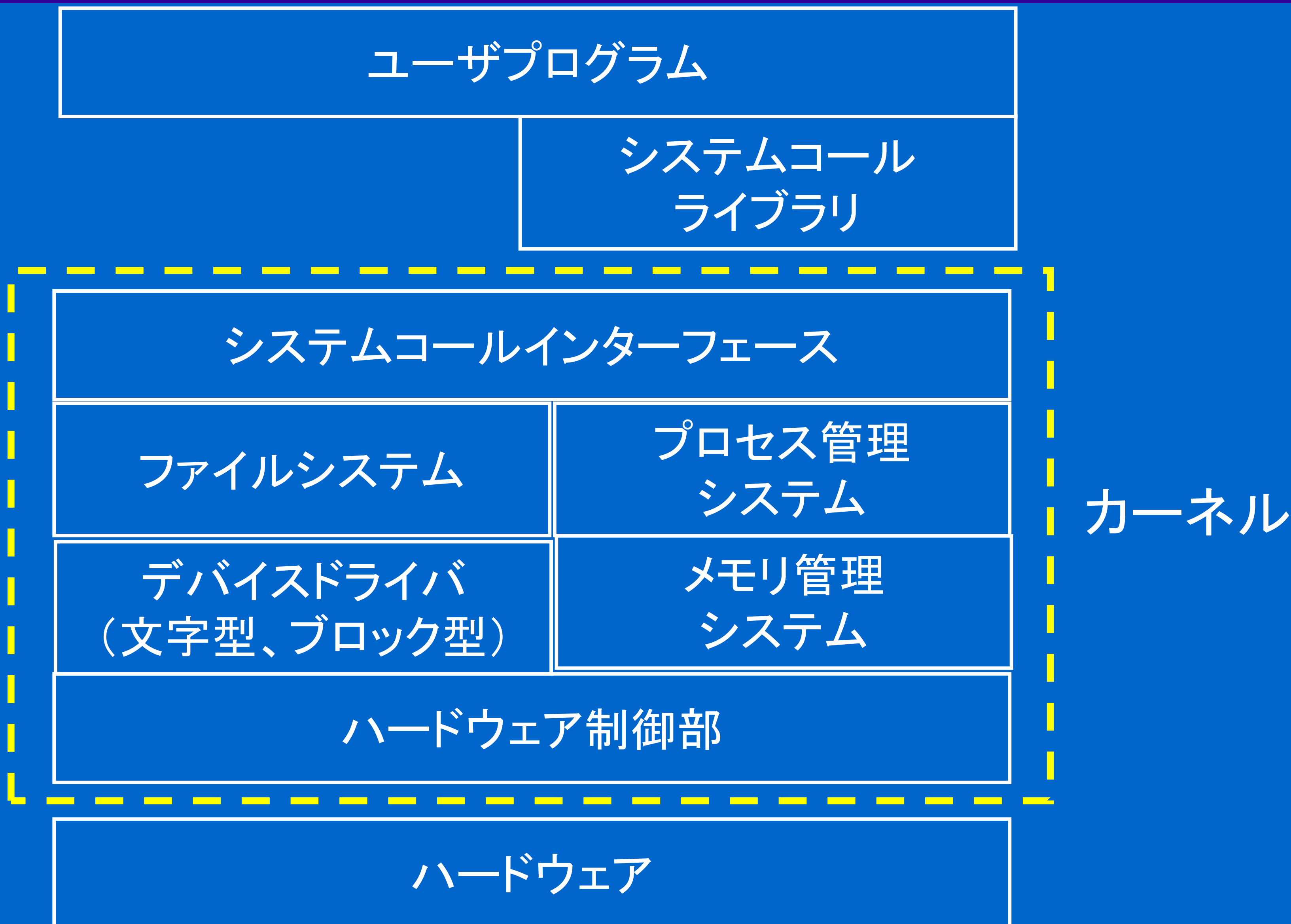
UNIXの特徴 (1)

- simplicity, elegance and ease of use
- 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」で資源を抽象化
  - 入出力装置
  - プロセス間通信
  - 通常のファイル

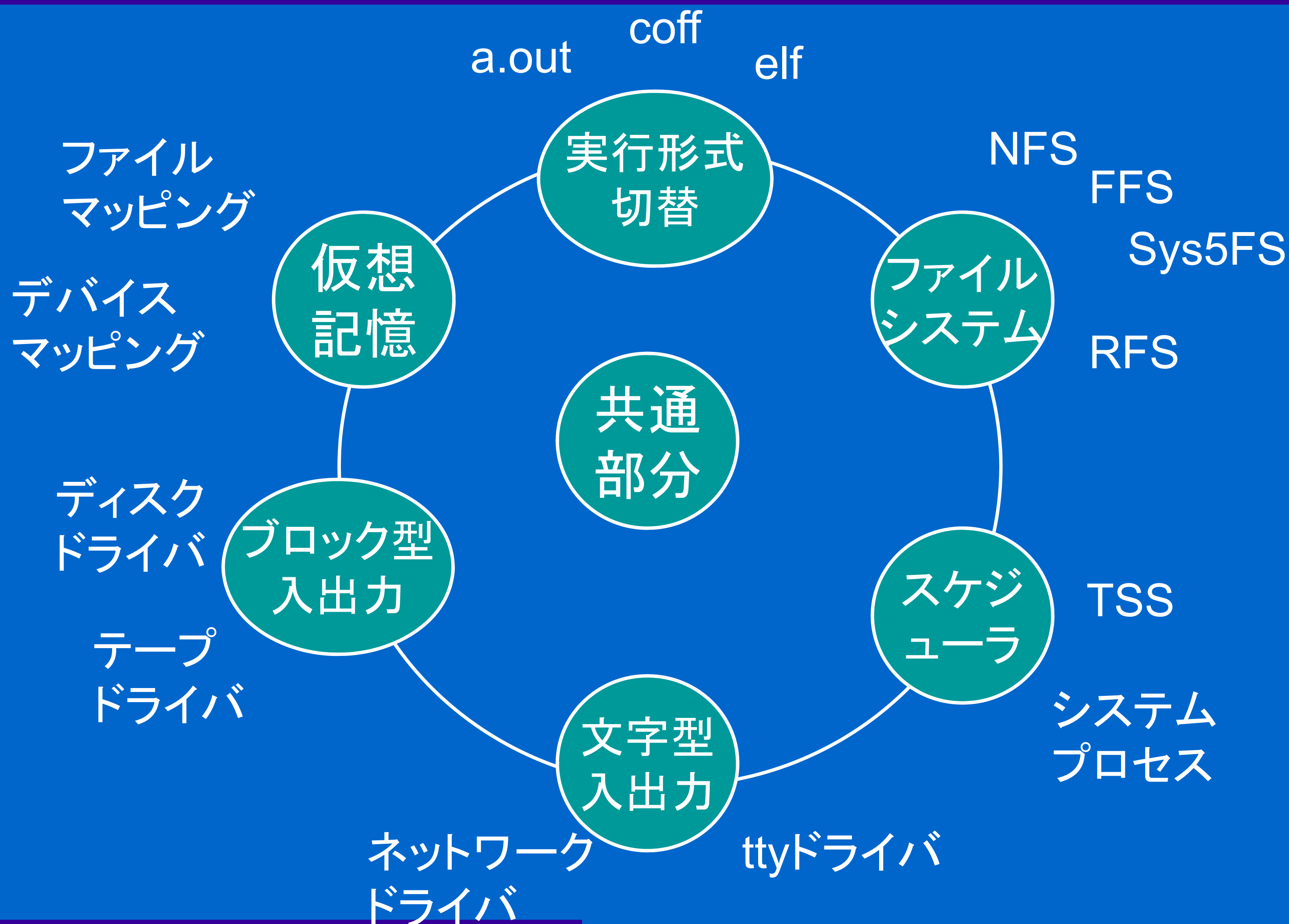
UNIXの特徴 (2)

- 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
- ツールキットアプローチ
  - 単機能のツール
  - 自由に組み合わせ可

カーネルの構成例(伝統的なUNIXカーネル)



最近のUNIXカーネルの構成例



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）

プロセス生成時の動作

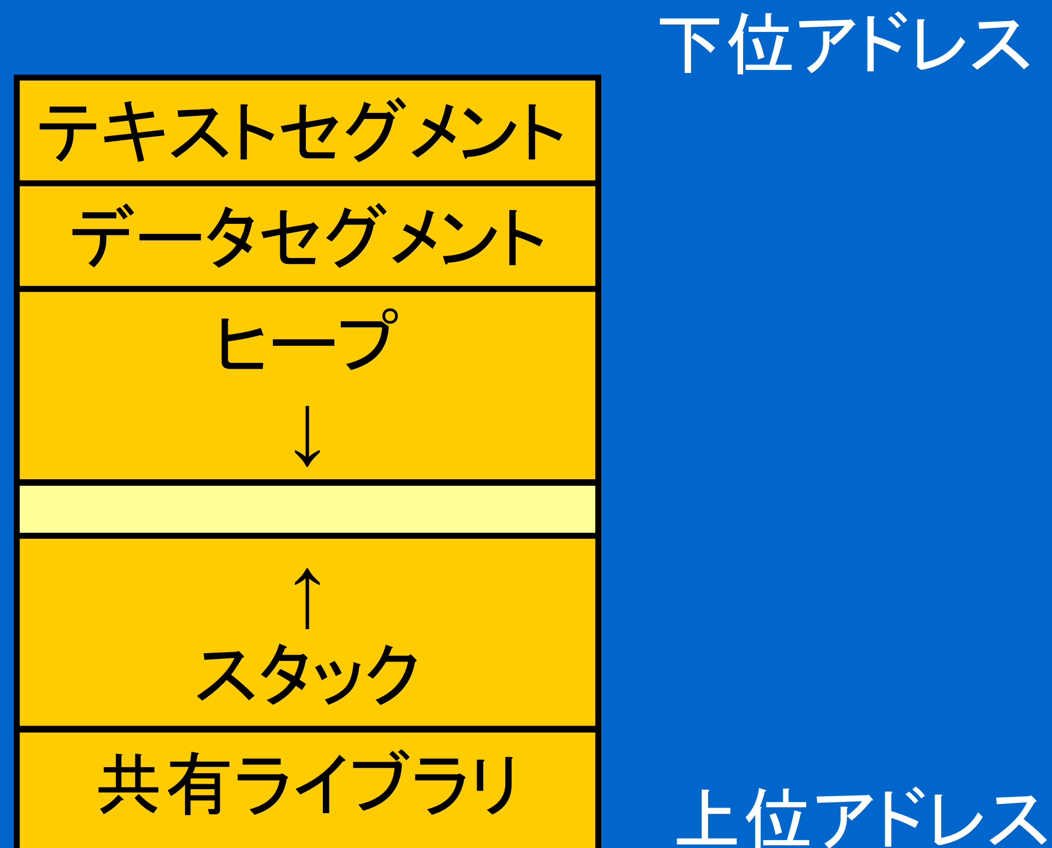
forkの実行

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。

（参考）UNIXのプロセス領域



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

- 優先度は整数を用いて表され、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1秒ごとにCPU時間の消費量に応じて再計算される

- 長時間CPUを割り付けられなかったプロセスの優先度は上げられ、最近プロセッサを割り付けられたプロセスの優先度は下げられる

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

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

Windows (1)

- IBMのパーソナルコンピュータ IBMPC
- 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 (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

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

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）

カーネルの構成法

ユーザモード
カーネルモード

ユーザプロセス

ファイルシステム
プロセス間通信
入出力管理
仮想記憶
プロセス管理

ハードウェア

単層カーネル方式

ユーザプロセス
ファイルサーバ
プロセス間通信サーバ
入出力管理サーバ
仮想記憶サーバ
プロセス管理サーバ

ユーザモード

マイクロカーネル

カーネルモード

ハードウェア

マイクロカーネル方式

Windows NT (3)

- 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**Windows NT 10.0**と表記される

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

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

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

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

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

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

WindowsNTシステムの構成

各種サブシステム

ユーザー
モード

NT Executive

NTマイクロカーネル

HAL

カーネル
モード

ハードウェア

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

NT Executive

- オブジェクトマネージャ
- LPC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
- Win32USER, **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 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

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(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
_execlpe、_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におけるスケジューリング

- 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）
- 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）

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

UNIXとWindows NT

	UNIX	Windows NT
位置付け	情報科学の研究基盤としてのOS（ソースの公開）	ビジネス利用のための商用OS
カーネルの構成	基本的に単層カーネル方式（研究用にマイクロカーネル方式を取るものもある）	マイクロカーネル方式によるモジュール化（効率化のため、一部のシステムサービスはカーネルで実行）
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（CPU割付け状況で調整） 計算性能を重視	スレッド単位で行われる優先度順とラウンドロビンを組み合わせたスケジューリング 応答性能を重視（ビジネス向けのOS）
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）