

オペレーティングシステム 第1章 オペレーティングシステムとは

<https://github.com/tctsigemura/OSTextBook>

オペレーティングシステムとは

1 / 16

オペレーティングシステムとは

Windows, macOS, Linux, FreeBSD, Android, iOS などのこと。

1. カーネル (OSの本体)
2. ライブライ (プログラムが使用するサブルーチン, DLL)
3. ユーザインターフェース (GUI, CLI)
4. ユーティリティソフトウェア (ファイル操作, 時計, シェル, システム管理 ...)
5. プログラム開発環境 (エディタ, コンパイラ, アセンブラー, リンカ, インタプリタ)

● **広義**では上に列挙した全てのこと。

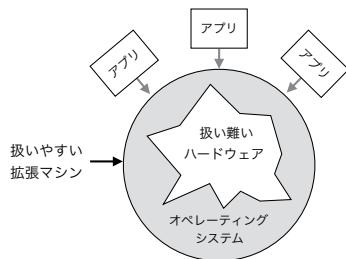
● **狭義**では「カーネル」だけを指す。

この科目では、主に「カーネル」の仕組みを学ぶ。

オペレーティングシステムとは

2 / 16

拡張マシンとしてのオペレーティングシステム



- 抽象化 (ファイル, プロセス)
- 拡張マシン (システムコール)

オペレーティングシステムとは

3 / 16

ハードウェア管理プログラムとしての オペレーティングシステム



- 仮想化
- 時分割多重による仮想化
- 空間分割多重による仮想化

オペレーティングシステムとは

4 / 16

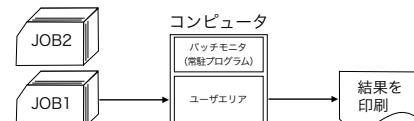
第1世代

- 真空管
- オペレーティングシステムなし
- 巨大 TeC 状態？

オペレーティングシステムとは

5 / 16

第2世代 (バッチ処理, その1)

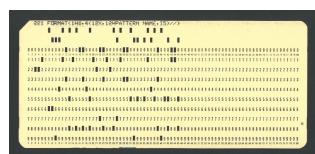


- メインフレーム
- バッチ, バッチ処理
- バッチモニタ
- JOB 制御言語 (JCL : Job Control Language)
- 実行モード
- システムコール
- 記憶保護

オペレーティングシステムとは

6 / 16

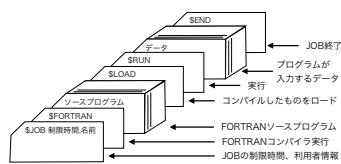
第2世代 (バッチ処理、その2)



- 紙カード
- JOBの構成

オペレーティングシステムとは

7 / 16



第3世代 (メインフレーム)



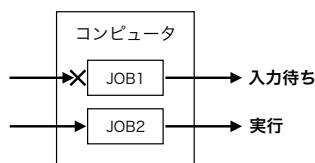
写真: フォルクスワーゲンで使われる System/360
Wikimedia / Bundesarchiv, B 145 Bild-F038812-0014 / Schaack, Lothar / CC-BY-SA 3.0 de

- IBM System/360、シリーズ化
- 仮想記憶

オペレーティングシステムとは

8 / 16

第3世代 (マルチプログラミング)



- OS/360
- MULTICS (MULTIplexed Information and Computing Service)
- UNIX (ユニックス)
- Dynabook

オペレーティングシステムとは

9 / 16

第3世代 (タイムシェアリング : TSS)

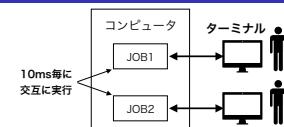


写真: <http://commons.wikimedia.org/wiki/File:Televideo925Terminal.jpg> (パブリックドメイン)

- TSS (Time Sharing System)
- ターミナル

オペレーティングシステムとは

10 / 16

次の世代に向けて (Dynabook)

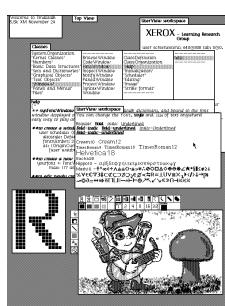


写真: ウィキメディア / SUMMIT ST / Alto や NoteTaker で動作したアラン・ケイ達の暫定 Dynabook 環境 (Smalltalk-76, 同-78 の頃) / CC-BY-SA 4.0

- GUI, マウス, パーソナルコンピュータ

オペレーティングシステムとは

11 / 16

IBM PC



写真: IBM PC
Wikimedia / Bundesarchiv, B 145 Bild-F077948-0006 / Engelbert Reineke / CC-BY-SA 3.0 de

- 16bit, 32bit, 64bit
- IBM PC (MS-DOS, Windows)
- 互換機

オペレーティングシステムとは

12 / 16

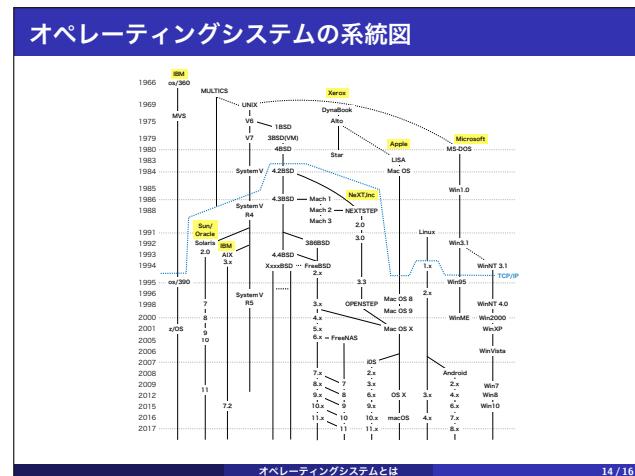
Apple Macintosh



写真：初代 Macintosh
ウィキメディア / w:User:Grm wr / File:Macintosh 128k transparency.png / GFDL

- 16bit, 32bit, 64bit
- Macintosh (Mac OS, Mac OS X, OS X, macOS)

オペレーティングシステムとは 13 / 16



練習問題（1）

次の言葉の意味を説明しなさい。

- オペレーティングシステム
- カーネル
- 拡張マシン
- 抽象化
- 仮想化
- 時分割多重
- 空間分割多重
- プロセス
- バッチモニタ
- 実行モード
- 記憶保護
- 仮想記憶
- マルチプログラミング
- タイムシェアリング

オペレーティングシステムとは 15 / 16

練習問題（2）

- 抽象化の例をいくつか挙げなさい。
- 仮想化の例をいくつか挙げなさい。
- 自分が使用しているパソコンのOSは何？
- 自分が使用しているスマートフォンのOSは何？

オペレーティングシステムとは 16 / 16

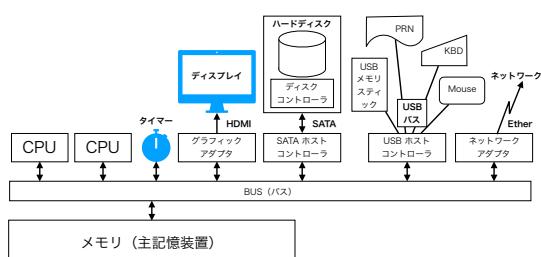
オペレーティングシステム 第2章 前提知識

<https://github.com/tctsigemura/OSTextBook>

前提知識

1 / 20

ハードウェア構成

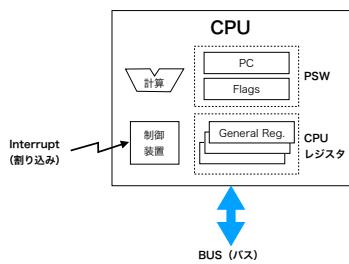


- SMP (Symmetric Multiprocessing)
- CPU, メモリ, タイマー, アダプタ, コントローラ, バス
- DMA (Direct Memory Access), I/O 完了割込み

前提知識

2 / 20

CPUの構成

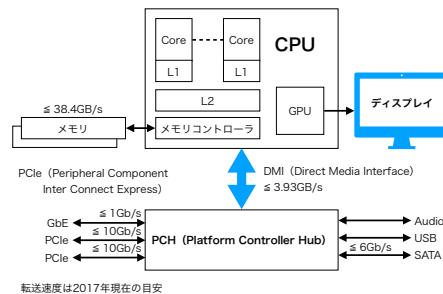


- PSW (Program Status Word)
- CPU レジスタ
- 割り込み (Interrupt)

前提知識

3 / 20

デスクトップ・パーソナルコンピュータ



- CPU
- コア (Core)

前提知識

4 / 20

デスクトップ・パーソナルコンピュータの内部

PC内部の例

マザーボード
CPU, メモリ, PCH,
PCIe, USB, GbE(Net),
HDMI, SATA
電源装置

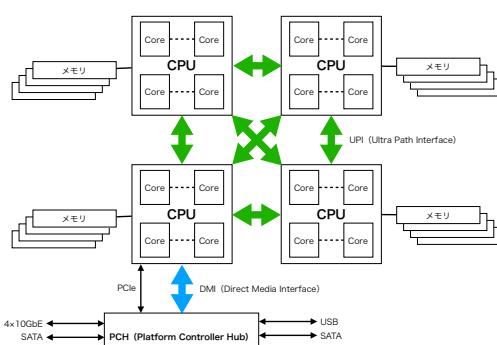
ストレージ
HDD(SATA),
DVD(SATA),
カードリーダ(USB)



前提知識

5 / 20

サーバコンピュータ



前提知識

6 / 20

割込み

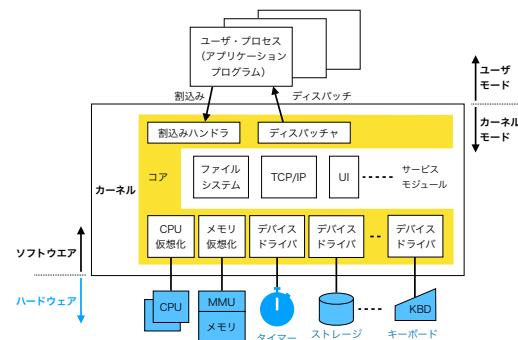
ユーザ・プロセスからカーネルに切り換わる唯一の方法

1. I/O完了・タイマー
ホストコントローラ、ネットワークアダプタ、タイマー等の命令完了など
2. システムコール
ユーザ/プロセスが *SVC (Supervisor Call)* 命令を実行
3. 保護違反
特権違反、メモリ保護違反
4. ソフトウェアのエラー
オーバーフロー、ゼロ除算など
5. ハードウェアのエラー
故障、電源異常

前提知識

7 / 20

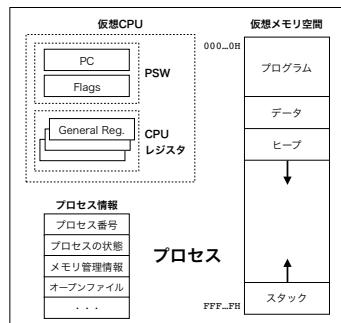
オペレーティングシステムの構造



前提知識

8 / 20

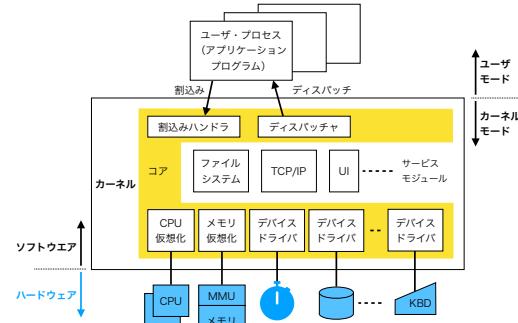
プロセスの構造



前提知識

9 / 20

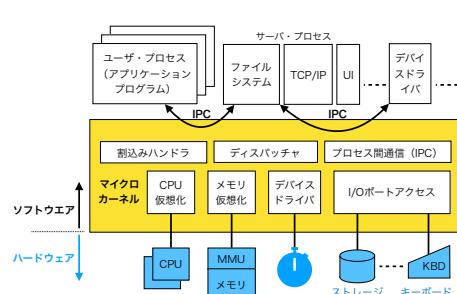
単層カーネル (モノリシック・カーネル)



前提知識

10 / 20

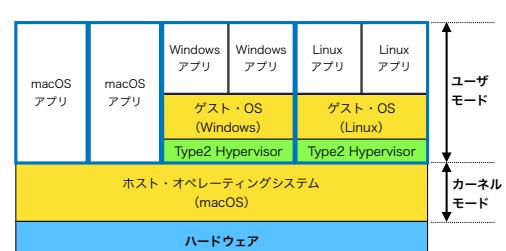
マイクロカーネル (micro-kernel)



前提知識

11 / 20

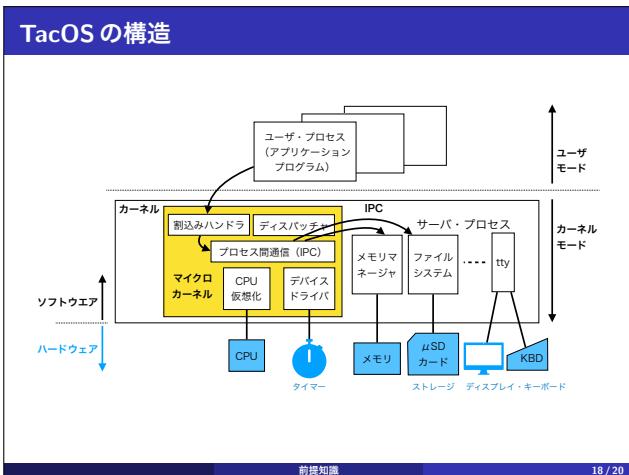
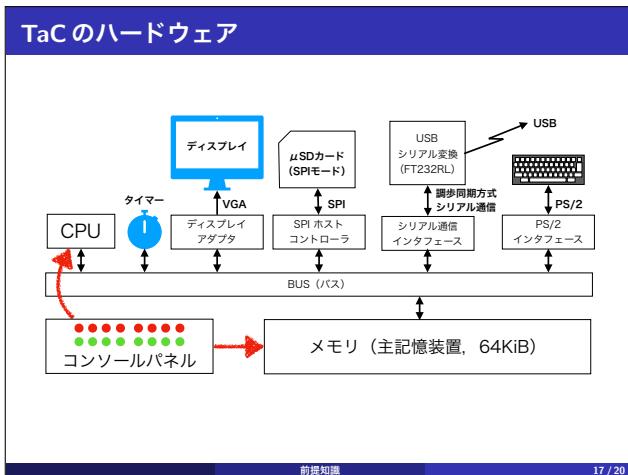
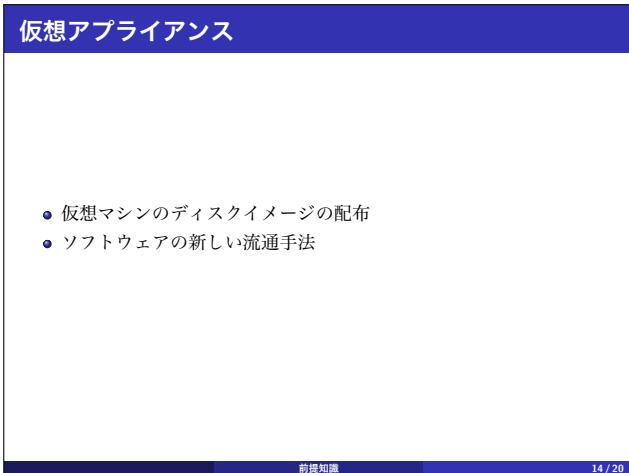
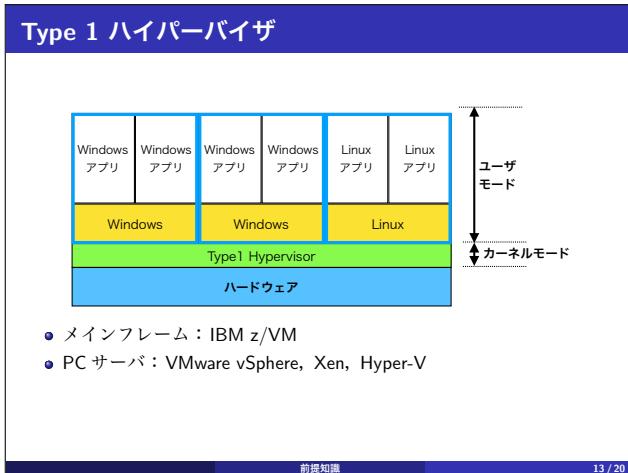
Type 2 ハイパーバイザ



- ホスト・オペレーティングシステム
- ゲスト・オペレーティングシステム
- VMware Workstation, VirtualBox

前提知識

12 / 20



練習問題（1）

次の言葉の意味を説明しなさい。

- CPU・ホストコントローラ・バス
- DMA
- SMP（対象型マルチプロセッシング）
- PSW・CPU レジスタ、割込み、SVC 命令
- ディスパッチャ
- サービスマジュール
- デバイスドライバ
- カーネルのコア
- コンテキスト
- 仮想 CPU
- 仮想メモリ空間
- 単層カーネル（モノリシック・カーネル）・マイクロカーネル
- IPC（プロセス間通信）
- Type 1 ハイパーテーバイザ・Type 2 ハイパーテーバイザ

前提知識

19 / 20

練習問題（2）

自分が使用しているPCのハードウェア構成を調べなさい。

- CPU の種類（名称、メーカー、クロック、コア数（CPU数））
- メモリの大きさ
- 二次記憶装置（ストレージ）の種類（ハードディスク？、SSD？）
- 二次記憶装置（ストレージ）の大きさ
- グラフィックアダプタの種類
- キーボードやマウスの接続方式（USB？、Bluetooth？）

前提知識

20 / 20

**オペレーティングシステム
第3章 CPU の仮想化**

<https://github.com/tctsigemura/OSTextBook>

CPU の仮想化 1 / 36

時分割多重による CPU の仮想化

- 時分割多重：CPU が実行するプロセスを次々切換える。
- コンテキストスイッチ：CPU が実行するプロセスを切換えること。
- ディスパッチ：プロセスに CPU を割り付ける。(実行開始)
- ディスパッチャ：ディスパッチするプログラムのこと。

CPU の仮想化 2 / 36

CPU の構造 (参考)

- コンテキスト = PSW + CPU レジスタ
- コンテキストを保存・ロードして次のプロセスに
- コンテキストスイッチ

CPU の仮想化 3 / 36

プロセスの構造 (参考)

- 仮想 CPU にコンテキストを保存

CPU の仮想化 4 / 36

プロセスの状態遷移

- 基本的な三つの状態
- 六つの状態遷移

CPU の仮想化 5 / 36

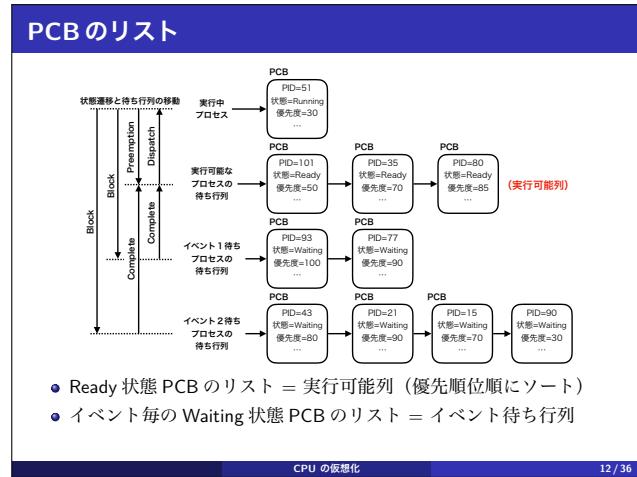
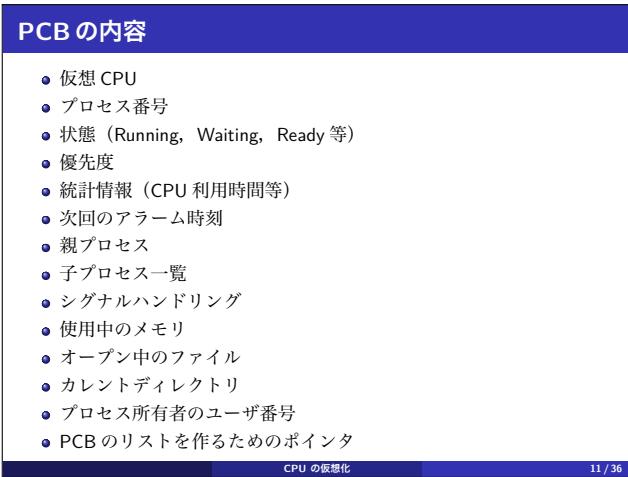
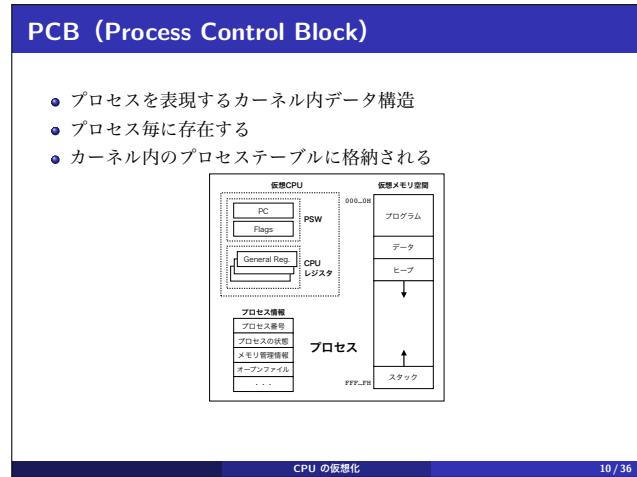
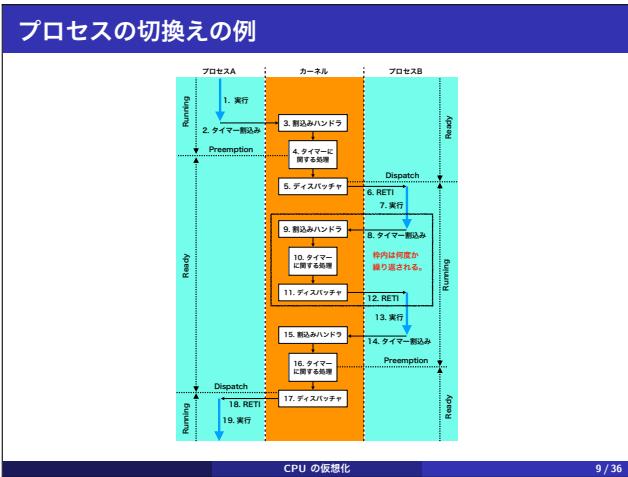
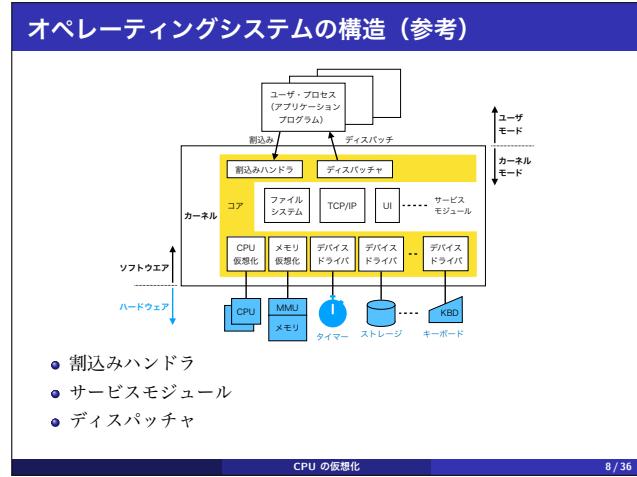
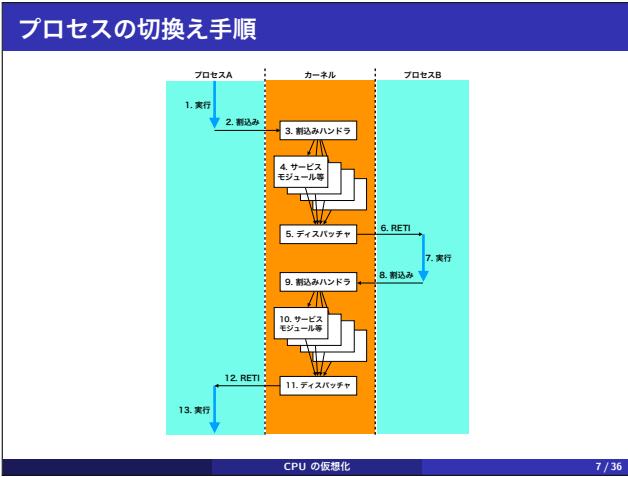
プロセス切換えの原因

- イベント

プロセス自ら「システムコールを発行する」Blockする
他のプロセスから「干渉を受け」Blockする
他のプロセスから「干渉を受け」Preemptionする
他のプロセスから「干渉を受け」Completeする
I/O完了やタイマ完了のイベントによりCompleteする
- タイムスライシング

クォンタムタイムを使い切ったプロセスはPreemptionする

CPU の仮想化 6 / 36



スレッド (Thread)

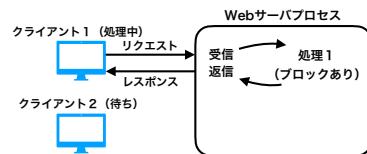
- CPU の仮想化によりマルチプログラミングが可能になった。
- プロセスが並行 (Concurrent) に実行できる。
- プロセスは一つの仮想 CPU を持っている。
- プロセスはコンピュータを仮想化したもの。
 - CPU が一つしかないコンピュータを仮想化している。
 - CPU を複数持つ SMP を仮想化するには不十分。
- 一つのプロセスが複数の仮想 CPU をもつモデルを導入する。
- プロセスが処理の流れスレッドを複数持つことができる。

CPU の仮想化

13 / 36

スレッドの役割 (1)

マルチプログラミングを用いない Web サーバ



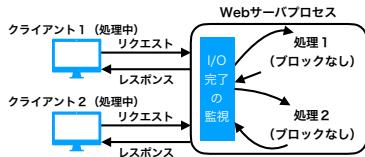
- 処理は順番に処理される。
- 前の処理が終わるまでクライアントは待たされる。

CPU の仮想化

14 / 36

スレッドの役割 (2)

マルチプログラミングを用いない Web サーバ



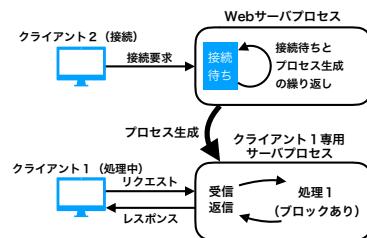
- 工夫すると並列して処理することも可能。
- しかし、プログラミングが難しい。
- しかし、SMP が活かせない。

CPU の仮想化

15 / 36

スレッドの役割 (3)

マルチプログラミングを用いる Web サーバ (プロセス版)



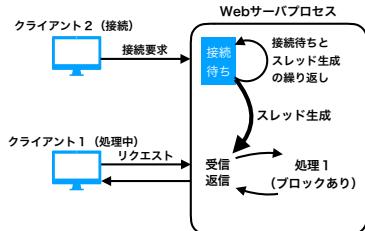
- クライアント毎にプロセスを起動 (fork()) する。
- プログラミングは難しい。
- しかし、処理が重いし、プロセス間の情報共有の効率が悪い。

CPU の仮想化

16 / 36

スレッドの役割 (4)

マルチプログラミングを用いる Web サーバ (スレッド版)

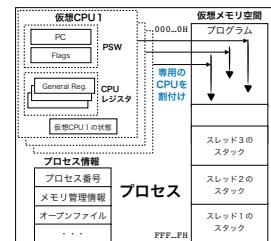


- クライアント毎にスレッドを起動する。
- プロセスの起動より 10~100 倍速い。
- スレッド間は情報共有しやすい。
- プログラミングは少し難しい。

CPU の仮想化

17 / 36

スレッドの形式 (1) -カーネルスレッド-

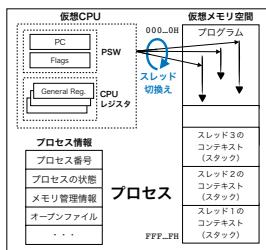


- プロセスが複数の仮想 CPU を持つ。

CPU の仮想化

18 / 36

スレッドの形式 (2) -ユーザスレッド-



- ユーザプログラム（ライブラリ）の工夫でスレッドを実現する。
- 並行（Parallel）実行はできない。
- どれかのスレッドがブロックすると全スレッドが停止する。

CPU の仮想化

19 / 36

スレッドの形式 (3) -スレッドモデル-

上記の2方式を組み合わせた3種類のスレッドモデルがある。

● One-to-One Model

一つのスレッドを一つのカーネルスレッドで実行する。

● Many-to-One Model

複数のスレッドを一つのカーネルスレッドで実行する。

● Many-to-Many Model

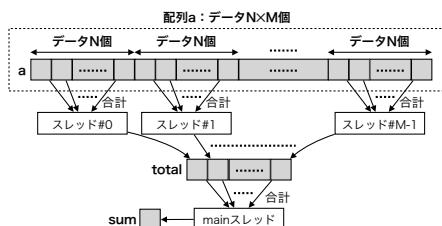
複数のスレッドを複数のカーネルスレッドで実行する。

CPU の仮想化

20 / 36

スレッドの使用例 (1)

M個のスレッドで手分けをして合計を計算する様子
(複数のカーネルスレッド(CPU)で手分けすることで短時間で処理が終わるはず)



CPU の仮想化

21 / 36

スレッドの使用例 (2)

M個のスレッドで合計を計算するプログラム

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
4 #define N 1000
5 #define M 10
6 pthread_t tid[M];
7 pthread_attr_t attr[M];
8 int a[M*N];
9 int total[M];
10 typedef struct { int no, min, max; } Args;
11
12 void *thread(void *arg) {
    // 自スレッドの担当部分のデータの合計を求める
    Args *args = arg;
    // 番目のスレッド
    int sum = 0;
    for (int i=args->min; i<args->max; i++) {
        sum += a[i];
    }
    total[args->no]=sum;
    return NULL;
}
  
```

CPU の仮想化

22 / 36

スレッドの使用例 (3)

```

22 int main() { // mainスレッドの実行はここから始まる
23     // 想定的なデータを生成する
24     for (int i=0; i<N*M; i++) // 配列 a を初期化
25         a[i] = i+1;
26     // 各スレッドを起動する
27     for (int m=0; m<M; m++) { // 各スレッドについて
        Args *p = malloc(sizeof(Args)); // 引数領域を確保
        p->no = m; // m番目のスレッド
        p->min = m*N; // 担当範囲下限
        p->max = N*(m+1); // 担当範囲上限
        pthread_attr_init(&attr[m]); // アトリビュート初期化
        pthread_create(&tid[m], &attr[m], thread, p); // スレッドを生成しスタート
    }
28     // 各スレッドの終了を待ち、求めた小計を合算する
29     int sum = 0;
30     for (int m=0; m<M; m++) { // 各スレッドについて
        pthread_join(tid[m], NULL); // 終了を待ち
        sum += total[m]; // 小計を合算する
    }
31     printf("1+2+ ... +%d=%d\n", N*M, sum);
32     return 0;
33 }
  
```

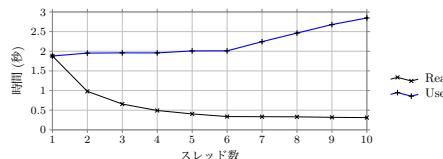
CPU の仮想化

23 / 36

スレッドの使用例 (4)

実行時間の計測結果

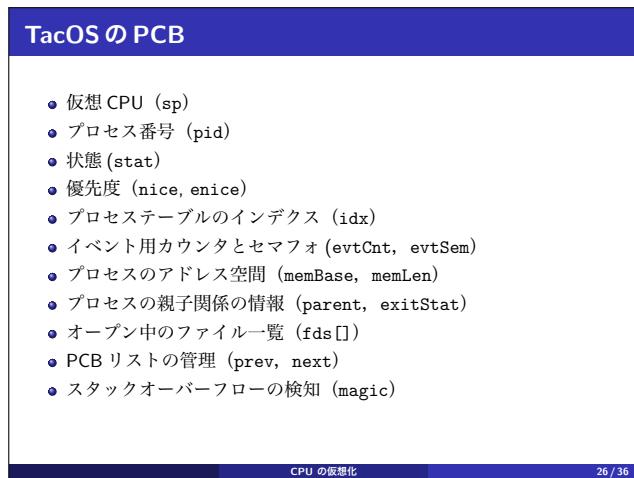
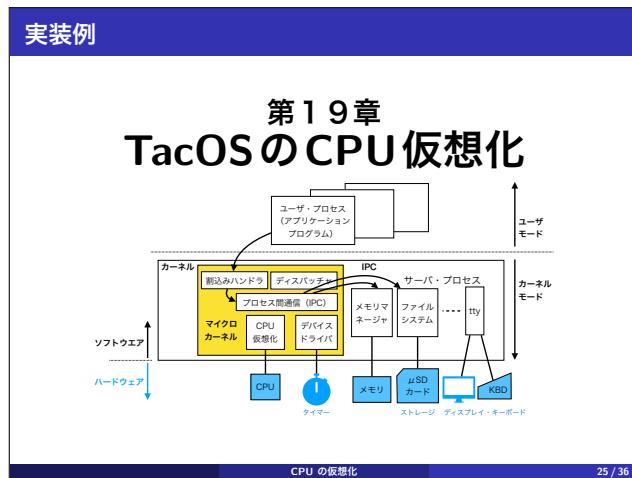
スレッド数 (N) × データ件数 (N*M)	スレッド数 (N) × データ件数 (N*M)									
	1	2	3	4	5	6	7	8	9	10
N	10,000	5,000	3,333	2,500	2,000	1,666	1,428	1,250	1,111	1,000
N*M	10,000	10,000	9,999	10,000	9,998	9,996	10,000	9,999	10,000	9,999



6コアのMac Proで計測
(Hyper-Threadingのお陰で6コアと12コアの中間的な振舞)

CPU の仮想化

24 / 36



TacOS の PCB (前半)

```
// ----- プロセス関連 -----
#define PRC_MAX 10 // プロセスは最大 10 個
#define P_KERN_STKSIZ 200 // プロセス毎のカーネルスタックのサイズ
#define P_LOW_PRI 30000 // プロセスの最低優先度
#define P_RUN 1 // プロセスは実行可能または実行中
#define P_WAIT 2 // プロセスは待ち状態
#define P_ZOMBIE 3 // プロセスは実行終了
#define P_MAGIC 0xabcd // スタックオーバーフロー検知に使用
#define P_FILE_MAX 4 // プロセスがオープンできるファイルの最大数

// プロセスコントロールブロック(PCB)
// 優先度は値が小さいほど優先度が高い
1 struct PCB { // PCB を表す構造体
2     int sp; // コンテキスト(他の CPU レジスタと PSW は
3             // プロセスのカーネルスタックに置く)
4     int pid; // プロセス番号
5     int stat; // プロセスの状態
6     int nice; // プロセスの本来優先度
7     int enice; // プロセスの実質優先度(将来用)
8     int idx; // この PCB のプロセステーブル上のインデクス

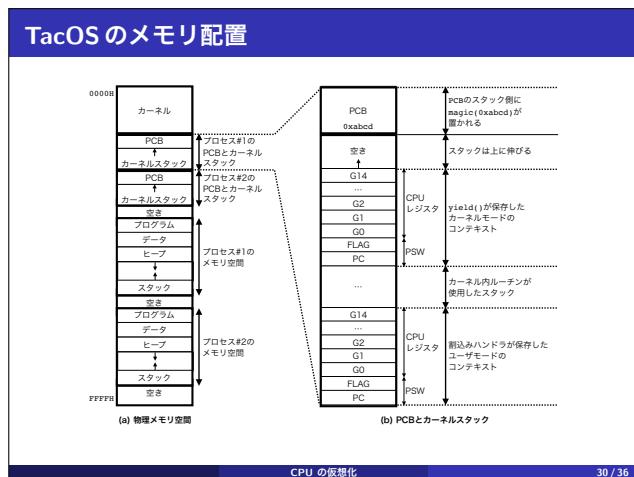
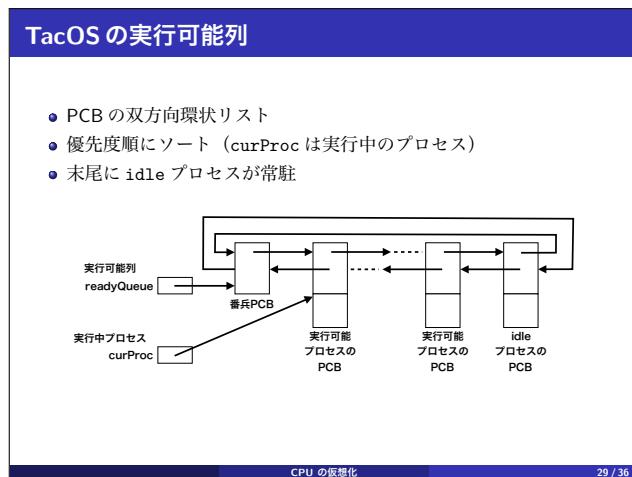
```

CPU の仮想化 27 / 36

TacOS の PCB (後半)

```
10 // プロセスのイベント用セマフォ
11 int evtCnt; // カウンタ(>0:sleep中, ==-1:wait中, ==0:未使用)
12 int evtSem; // イベント用セマフォの番号
13
14 // プロセスのアドレス空間(text, data, bss, ...)
15 char[] memBase; // プロセスのメモリ領域のアドレス
16 int memLen; // プロセスのメモリ領域の長さ
17
18 // プロセスの親子関係の情報
19 PCB parent; // 親プロセスへのポインタ
20 int exitStat; // プロセスの終了ステータス
21
22 // オープン中のファイル一覧
23 int[] fds; // オープン中のファイル一覧
24
25 // プロセスは重連結環状リストで管理
26 PCB prev; // PCB リスト(前へのポインタ)
27 PCB next; // PCB リスト(次のポインタ)
28 int magic; // スタックオーバーフローを検知
29 }
```

CPU の仮想化 28 / 36



TacOS のタイマー管理プログラム

```

3 // タイマー割り込みハンドラ(10ms 毎に割り込みによって起動される)
4 interrupt tmrIntr() {
5     boolean disp = false; // ディスパッチの必要性
6
7     // 起きないといけないプロセスを起こしてまわる
8     for (int i=0; i<RC_MAX; i=i+1) {
9         PCB p = procTbl[i];
10        if (p!=null && p.evtCnt>0) { // タイマー稼働中なら
11            int cnt = p.evtCnt - TICK; // 残り時間を計算
12            if (cnt<=0) { // 時間が来たら
13                cnt = 0; // タイマーを停止し
14                disp = iSemV(p.evtSem) || disp; // プロセスを起こす
15            }
16            p.evtCnt = cnt;
17        }
18    }
19
20    if (disp) yield(); // 必要ならディスパッチ
21 }

```

CPU の仮想化

31 / 36

参考(普通の関数)

C--言語の void 型関数

```

1 void func();
2
3 void sFunc() {
4     for (int i=0; i<10; i=i+1) {
5         func(); // 関数呼び出し
6     }
7 }

```

```

1 .sFunc PUSH FP
2     LD FP,SP
3     PUSH G3
4     LD G3,#0
5     .L1
6     CMP G3,#10
7     JGE .L2
8     CALL .func
9     LD G0,G3
10    ADD G0,#1
11    LD G3,G0
12    JMP .L1
13
14    POP G3
15    POP FP
16    RET

```

CPU の仮想化

32 / 36

参考(interrupt 関数)

C--言語の interrupt 型関数

```

1 void func();
2
3 interrupt iFunc() {
4     for (int i=0; i<10; i=i+1) {
5         func(); // 関数呼び出し
6     }
7 }

```

```

1 .iFunc PUSH G0
2     PUSH G1
3     PUSH G2
4     PUSH FP
5     LD FP,SP
6     PUSH G3
7     LD G3,#0
8     .L1
9     CMP G3,#10
10    JGE .L2
11    CALL .func
12    LD G0,G3
13    ADD G0,#1
14    LD G3,G0
15    JMP .L1
16
17    POP G3
18    POP FP
19    POP G2
20    POP G1
21    POP G0
22    RETI

```

CPU の仮想化

33 / 36

TacOS のコンテキスト保存プログラム (yield())

```

1 yield
2 ;--- G13(SP)以外の CPU レジスタと FLAG をカーネルスタックに退避 ---
3 push g0 ; FLAG の保存場所を準備する
4 push g0 ; G0 を保存
5 ld g0.flag ; FLAG を上で準備した位置に保存
6 st g0,2,sp ;
7 push g1 ; G1 を保存
8 push g2 ; G2 を保存
...
17 push g11 ; G11 を保存
18 push fp ; フレームポイント(G12)を保存
19 push usp ; ユーザモードスタックポイント(G14)を保存
20 ;
21 ;----- G13(SP)を PCB に保存 -----
22 ld g1,_curProc ; G1 <- curProc
23 st sp,0,g1 ; [G1+0] は PCB の sp フィールド
24 ;
25 ;----- [curProc の magic フィールド]をチェック -----
26 ld g0,30,g1 ; [G1+30] は PCB の magic フィールド
27 cmp g0,#0xabcd ; P_MAGIC と比較、一致しなければ
28 jnz .stkOverflow ; カーネルスタックがオーバーフローしている

```

CPU の仮想化

34 / 36

TacOS のコンテスト復旧プログラム (dispatch())

_dispatch は _yield (28 行) の直後にある。(つながっている!)

```

1 _dispatch
2 ;----- 次に実行するプロセスの G13(SP)を復元 -----
3 ld g0,_readyQueue ; 実行可能列の番兵のアドレス
4 ld g0,28,g0 ; [G0+28] は PCB の next フィールド(先頭の PCB)
5 st g0,_curProc ; 現在のプロセス(curProc)に設定する
6 ld sp,0,g0 ; PCB から SP を取り出す
7 ;
8 ;----- G13(SP)以外の CPU レジスタを復元 -----
9 pop usp ; ユーザモードスタックポイント(G14)を復元
10 pop fp ; フレームポイント(G12)を復元
11 pop g11 ; G11 を復元
12 pop g10 ; G10 を復元
13 pop g9 ; G9 を復元
...
21 pop g1 ; G1 を復元
22 pop g0 ; G0 を復元
23 ;
24 ;----- PSW(FLAG と PC)を復元 -----
25 reti ; RETI 命令で一度に POP して復元する

```

CPU の仮想化

35 / 36

練習問題

● 次の言葉の意味を説明しなさい。

- 時分割多重
- コンテキストスイッチ
- Dispatch (ディスパッチ)
- Preemption (ブリエンプション)
- プロセスの状態
- プロセスの状態遷移
- RETI 命令
- PCB
- 待ち行列
- 実行可能列
- スレッド
- カーネルスレッド
- ユーザスレッド
- One-to-One Model
- Many-to-One Model
- Many-to-Many Model

● POSIX スレッドについて調査しなさい。

CPU の仮想化

36 / 36

オペレーティングシステム 第4章 スケジューリング

<https://github.com/tctsigemura/OSTextBook>

スケジューリング 1 / 20

評価基準

- スループット (Throughput)
- ターンアラウンド時間 (Turnaround time)
- レスポンス時間 (Response time)
- 締め切り (Deadline)
- その他 (公平性, 省エネ, 預測性など)

スケジューリング 2 / 20

システムごとの目標

コンピュータの種類	重視する性能
メインフレーム (パッチ処理)	スループット, ターンアラウンド時間
ネットワークサーバ	レスポンス時間, スループット
デスクトップパソコン	レスポンス時間
モバイルデバイス	レスポンス時間, 省エネルギー
組込み制御	締め切り

スケジューリング 3 / 20

CPUバウンドプロセス

動画圧縮の例

The diagram illustrates the execution of a CPU-bound process (CPU burst) and an I/O-bound process (I/O burst) over time. The CPU burst is shown as a long green bar, and the I/O burst is shown as a blue bar labeled '読み' (read). A red bar labeled '書き' (write) represents the compression processing. The timeline shows the alternation between CPU bursts and I/O bursts, with a '圧縮処理' (compression processing) label indicating the time spent on compression.

スケジューリング 4 / 20

I/Oバウンドプロセス

スプレッドシートの例

The diagram illustrates the execution of an I/O-bound process. It shows two phases: '入力待ち' (input wait) and '再計算' (re-computation). The CPU burst is shown as a green bar, and the I/O burst is shown as a blue bar labeled 'I/Oバースト (I/O burst)'. The timeline shows the alternation between these phases.

スケジューリング 5 / 20

FCFSスケジューリング (1)

FCFS (First-Come, First-Served)

- プリエンプションしないスケジューリング方式

プロセス	到着時刻	CPUバースト時間 (ms)
P_1	0	100
P_2	0	20
P_3	0	10

The Gantt chart shows three processes, P_1 , P_2 , and P_3 , executing sequentially. The timeline starts at 0 ms. P_1 runs for 100 ms, P_2 for 20 ms, and P_3 for 10 ms. The total execution time is 130 ms.

- P_1 , P_2 , P_3 の順に実行
- 平均ターンアラウンド時間 ($((100 + 120 + 130)/3 = 117 \text{ ms})$)
- 最悪の平均ターンアラウンド時間を選択することもある。

スケジューリング 6 / 20

FCFSスケジューリング (2)

- 平均ターンアラウンド時間は 到着順により大きく変化する。

プロセス 到着時刻 CPU バースト時間 (ms)

P_1	0	100
P_2	0	20
P_3	0	10



- P_2, P_3, P_1 の順に実行

平均ターンアラウンド時間 $((20 + 30 + 130)/3 = 60 \text{ ms})$

スケジューリング

7 / 20

SJFスケジューリング

SJF (Shortest-Job-First)

- プリエンプションしないスケジューリング方式

プロセス 到着時刻 CPU バースト時間 (ms)

P_1	0	100
P_2	0	20
P_3	0	10



平均ターンアラウンド時間 $((10 + 30 + 130)/3 = 57 \text{ ms})$

スケジューリング

8 / 20

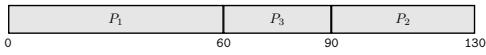
SRTFスケジューリング (1)

SRTF (Shortest-Remaining-Time-First)

比較のための SJF スケジューリングの例

プロセス 到着時刻 CPU バースト時間 (ms)

P_1	0	60
P_2	10	40
P_3	60	30



- SJF はプリエンプションなし

平均ターンアラウンド時間
 $((60 - 0) + (90 - 10) + (130 - 60))/3 = 70 \text{ ms}$

スケジューリング

9 / 20

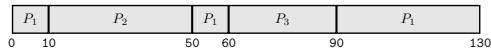
SRTFスケジューリング (2)

SRTF (Shortest-Remaining-Time-First)

前の SJF と同じプロセスを SRTF でスケジューリング

プロセス 到着時刻 CPU バースト時間 (ms)

P_1	0	60
P_2	10	40
P_3	60	30



- SRTF はプリエンプションあり

平均ターンアラウンド時間
 $((130 - 0) + (50 - 10) + (90 - 60))/3 = 67 \text{ ms}$

スケジューリング

10 / 20

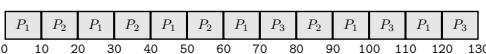
RRスケジューリング (1)

RR (Round-Robin)

クォンタムタイムまでプリエンプションしない。

プロセス 到着時刻 CPU バースト時間 (ms)

P_1	0	60
P_2	10	40
P_3	60	30



- クォンタムタイム = 10ms

平均ターンアラウンド時間
 $((120 - 0) + (90 - 10) + (130 - 60))/3 = 90 \text{ ms}$

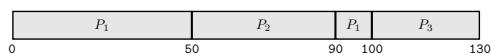
スケジューリング

11 / 20

RRスケジューリング (2)

プロセス 到着時刻 CPU バースト時間 (ms)

P_1	0	60
P_2	10	40
P_3	60	30



- クォンタムタイム = 50ms

平均ターンアラウンド時間
 $((100 - 0) + (90 - 10) + (130 - 60))/3 = 83 \text{ ms}$

スケジューリング

12 / 20

優先度順スケジューリング

Priority

- 実行可能列を優先度順でソートしておく。

状態遷移と待ち行列の説明
Block → Dispatch → Complete

実行中のプロセス → 実行可能なプロセスの待ち行列 → イベント1待ちプロセスの待ち行列 → イベント2待ちプロセスの待ち行列

(実行可能列)

● 静的優先度／動的優先度
● スターベーション (starvation)：飢餓
● エージング (aging)：老化、熟成

スケジューリング 13 / 20

FBスケジューリング

FB (Multilevel Feedback Queue)

優先度別実行可能列

● エージング

スケジューリング 14 / 20

実装例

第19章 (6節) TacOSのスケジューラ

ユーザ モード
ユーザ プロセス (アプリケーション プログラム)

カーネル モード
マイクロ カーネル
割込みハンドラ
ディスパッチャ
プロセス間通信 (IPC)
CPU バーチャライゼーション
ディバイス ドライバ
メモリ ネッジ
ファイル システム
tty

ハードウェア
CPU
タイマー
メモリ
SD カード
ストレージ
ディスプレイ・キーボード
KBD

スケジューリング 15 / 20

TacOSのスケジューラ

```

1 public void schProc(PCB proc) {
2     int r = setPri(D1|KERN);
3     int enice = proc.enice;
4     PCB head = readyQueue.next;
5     while (head.enice<=enice)
6         head = head.next;
7     insProc(head,proc);
8     setPri(r);
9 }
  
```

// 割り込み禁止、カーネル
// 実行可能列から
// 優先度がより低い
// プロセスを探す
// 見つけたプロセスの
// 直前に挿入する
// 割り込み状態を復元する

スケジューリング 16 / 20

TacOSの実行可能列 (参考)

- yield()
- dispatch()
- 実行可能列

実行可能列 readyQueue

実行中プロセス curProc

番兵PCB 実行可能プロセスの PCB 実行可能プロセスの PCB idle プロセスの PCB

スケジューリング 17 / 20

練習問題

練習問題

スケジューリング 18 / 20

練習問題（1）

- 次の言葉の意味を説明しなさい。
 - スループット
 - ターンアラウンド時間・レスポンス時間
 - ハードリアルタイム・ソフトリアルタイム
 - CPU バウンドプロセス・I/O バウンドプロセス
 - FCFS スケジューリング・SJF スケジューリング
 - SRTF スケジューリング・RR スケジューリング
 - 優先度順スケジューリング・FB ケジューリング
 - クォンタム時間
 - スタペーション
 - エージング

スケジューリング

19 / 20

練習問題（2）

- 次の三つのプロセスの実行順をガントチャートで示しなさい。また、平均ターンアラウンド時間を計算しなさい。

プロセス名	到着時刻 (ms)	CPUバースト時間 (ms)
P_1	0	70
P_2	10	50
P_3	20	30

- FCFS でスケジューリングした場合
- SJF でスケジューリングした場合
- SRTF でスケジューリングした場合
- RR (但しクォンタム時間は 20ms) でスケジューリングした場合
- RR (但しクォンタム時間は 40ms) でスケジューリングした場合
- RR (但しクォンタム時間は 60ms) でスケジューリングした場合

スケジューリング

20 / 20

オペレーティングシステム 第5章 プロセス同期

<https://github.com/tctsigemura/OSTextBook>

1 / 40

共有資源

- スレッド間の共有変数
- プロセス間の共有メモリ
- カーネル内のデータ構造
- ファイル
- 入出力装置
- その他

プロセス同期

2 / 40

競合 (Race Condition, Competition)

```
// スレッド間の共有変数
receipt DS    1    // 入金 (3万円)
payment DS    1    // 引き落とし (2万円)
account DS    1    // 残高 (10万円)

// 入金管理スレッド          // 引き落とし管理スレッド

// 会社から給料 (3万円) を受領し          // カード会社から引き落としを
// receipt に金額を格納した。              // 受信し payment に金額を格納した。

// 口座 account に足し込む
(1) LD      GO,account
(2) ADD     GO,receipt
(3) ST      GO,account

// 次の処理に進む          // 次の処理に進む
```

3 / 40

クリティカルセクション (CriticalSection)

複数のプロセス（スレッド）が同時に実行すると競合が発生する！！
 例えば共有変数を変更する処理はクリティカルセクションである。
 (前の例で「(1) から (3)」と「(a) から (c)」)
 (クリティカルリージョン (CriticalSection) とも呼ぶ)
 クリティカルセクションには複数のスレッドが入ってはならない。

クリティカルセクションの競合問題を効率よく解決するには、
 (1) 二つ以上のスレッドが同時に入らない。
 (2) 入っているスレッドがない時は、すぐに入れる。
 (3) 入るためにスレッドが永遠に待たされることがない。

プロセス同期

4 / 40

相互排除 (mutual exclusion)

複数のスレッドが同時にクリティカルセクションに入らない制御！！
 (排他制御、相互排他も呼ぶ)

相互排除を行うプログラムの部分のことを次のように呼ぶ。

- エントリーセクション (Entry Section)
 クリティカルセクションに入る手続き
- エグジットセクション (Exit Section)
 クリティカルセクションを出る手続き

5 / 40

割込み禁止

シングルプロセッサシステムで相互排除に使用できる。

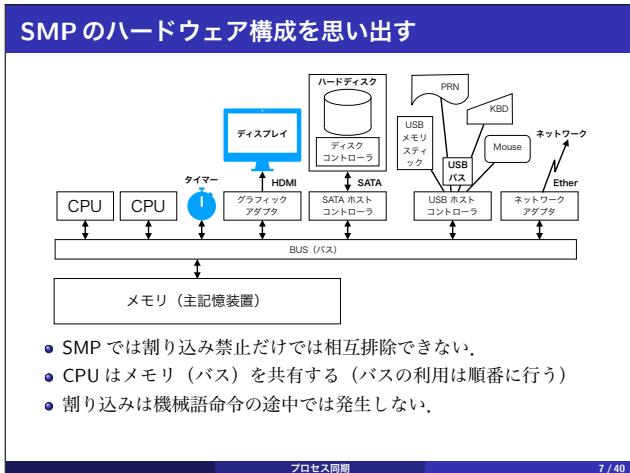
// 口座 account に足し込む	// 口座 account から差し引く
DI // Entry Section	DI // Entry Section
(1) LD GO,account	(a) LD GO,account
(2) ADD GO,receipt	(b) SUB GO,payment
(3) ST GO,account	(c) ST GO,account
EI // Exit Section	EI // Exit Section

- エントリーセクション (Entry Section)
 割り込み禁止の場合は DI 命令
- エグジットセクション (Exit Section)
 割り込み禁止の場合は EI 命令

DI 命令、EI 命令は特権命令なのでカーネル内だけで可能。
 長時間の割り込み禁止は NG なので注意が必要。

プロセス同期

6 / 40



専用命令 (TS 命令)

TS (Test and Set) 命令は SMP システムでの相互排除に使用できる。「TS R, M」は以下をアトミック (atomic) に実行する。

1. バスをロックする
 2. $R \leftarrow [M]$
 3. if ($R == 0$) $Zero \leftarrow 1$ else $Zero \leftarrow 0$
 4. $[M] \leftarrow 1$
 5. バスのロックを解除する
- プロセス同期 8 / 40

専用命令 (TS 命令の使用例)

```
// エントリーセクション
L1   DI      // クリティカルセクションでプリエンプションしないように
    TS     GO, FLG // ゼロを取得できるプロセス(スレッド)は一時には一つだけ
    JZ     L2      // ゼロを取得できた場合だけクリティカルセクションに入る
    EI     // ビジーウェイティングの間はプリエンプションのチャンスを作る
    JMP   L1      // クリティカルセクションに入れない場合はビジーウェイティング

// クリティカルセクション
L2   ...

// エグジットセクション
LD   GO, #0
ST   GO, FLG // フラグのクリアは普通のST命令でOK
EI   // クリティカルセクション終了、プリエンプションしても良い

// 非クリティカルセクション
...

// メモリ上に置いたフラグ(CPUのフラグと混同しないこと)
FLG  DC      0       // 初期値ゼロ(TS命令により1に書き換えられる)

● ビジーウェイティング (Busy Waiting) の一種。
```

プロセス同期 9 / 40

専用命令 (SW 命令)

SW (Swap) 命令も SMP システムでの相互排除に使用できる。「SW R, M」は以下をアトミック (atomic) に実行する。

1. バスをロックする
2. $T \leftarrow [M]$
3. $[M] \leftarrow R$
4. $R \leftarrow T$
5. バスのロックを解除する

ここで T は CPU 内部の一時的なレジスタ (T レジスタの存在はプログラムから見えない)

- 次の例もビジーウェイティング (Busy Waiting) の一種。

プロセス同期 10 / 40

専用命令 (SW 命令の使用例)

```
// エントリーセクション
L1   DI      // クリティカルセクションでプリエンプションしないように
    LD     GO, #1 // フラグに書き込む値
    SW     GO, FLG // ゼロを取得できるプロセス(スレッド)は一時には一つだけ
    CMP   GO, #0 // ゼロを取得できたかテスト
    JZ    L2      // ゼロを取得できた場合だけクリティカルセクションに入る
    EI     // ビジーウェイティングの間はプリエンプションのチャンスを作る
    JMP   L1      // クリティカルセクションに入れない場合はビジーウェイティング

// クリティカルセクション
L2   ...

// エグジットセクション
LD   GO, #0
ST   GO, FLG // フラグのクリアは普通のST命令でOK
EI   // クリティカルセクション終了、プリエンプションしても良い

// 非クリティカルセクション
...

// メモリ上に置いたフラグ(CPUのフラグと混同しないこと)
FLG  DC      0       // 初期値ゼロ(SW命令により1に書き換えられる)
```

プロセス同期 11 / 40

専用命令 (CAS 命令)

CAS (Compare And Swap) 命令も SMP システムでの相互排除に使用できる。「CAS R0, R1, M」は、以下をアトミック (atomic) に実行する。

1. バスをロックする
2. $T \leftarrow [M]$
3. if ($T == R0$) $\{ [M] \leftarrow R1; Zero \leftarrow 1; \}$
 $\text{else } \{ R0 \leftarrow T; Zero \leftarrow 0; \}$
4. バスのロックを解除する

// 口座 account に足し込む	// 口座 account から差し引く
L1 LD GO,account	L2 LD GO,account
LD G1, G0	LD G1, G0
ADD G1, receipt	SUB G1, payment
CAS GO, G1, account	CAS GO, G1, account
JNZ L1	JNZ L2

ロックフリー (Lock-free) なアルゴリズム

プロセス同期 12 / 40

フラグを用いる方法 (Peterson のアルゴリズム)

```
// スレッド間の共有変数
boolean flag[] = {false, false}; // クリティカルセクションに入りたい
int turn = 0; // 後でやってきたのはどちら

// スレッド0
...
// エントリーセクション
flag[0] = true;
turn = 0;
while (turn==0 && flag[1]==true)
    ; // ビジーウェイティング

// クリティカルセクション
...
// エグジットセクション
flag[0] = false;

// 非クリティカルセクション
...

// スレッド1
...
// エントリーセクション
flag[1] = true;
turn = 1;
while (turn==1 && flag[0]==true)
    ; // ビジーウェイティング

// クリティカルセクション
...
// エグジットセクション
flag[1] = false;

// 非クリティカルセクション
...
```

プロセス同期

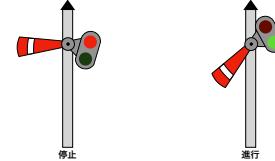
13 / 40

セマフォ (Semaphore)

1965年にE.W.Dijkstraが提案したデータ型である。

- ビジーウェイティング (Busy Waiting) を用いない
- オペレーティングシステムが提供する洗練された同期機構
- システムコール等でユーザプロセスに提供
- サブルーチンとしてサービスモジュール等に提供

セマフォ (Semaphore: 腕木式信号機) の元々の意味はこれ！



プロセス同期

14 / 40

セマフォ (Semaphore)

- セマフォはデータ構造 (**セマフォ型**, セマフォ構造体)
例ええばC言語で:「`typedef struct { ... } Semaphore;`」
- カウンタは0以上の整数値 (0は**赤信号**の意味)
- プロセスの待ち行列を作ることができる。
- セマフォ型の変数にP操作とV操作ができる。
- P操作 (*Proberen:try*)
- V操作 (*Verhogen:raise*)
- ユーザプロセスにはP, Vシステムコールが提供される
- サービスマジュールやデバイスドライバにはP, Vサブルーチン

セマフォはプロセス (スレッド) の状態を待ち (Waiting) 状態に変える。
ビジーウェイティング (Busy Waiting) では無いのでCPUを無駄遣いすることはない。

プロセス同期

15 / 40

セマフォ (Semaphore) のP操作

P操作 (*P(S)*)

1. セマフォ (S) の値が1以上ならセマフォの値を1減らす。
2. 値が0ならプロセス (スレッド) を待ち (Waiting) 状態にし、
3. セマフォの待ち行列に追加する。

クリティカルセクションのエントリーセクション等で使用できる。

```
void P(Semaphore S) {
    if (S > 0) {
        S--;
    } else {
        プロセスを待ち(Waiting)状態にする;
        プロセスを S の待ち行列に追加する;
    }
}
```

プロセス同期

16 / 40

セマフォ (Semaphore) のV操作

V操作 (*V(S)*)

1. 待っているプロセス (スレッド) が無い場合は、セマフォ (S) の値を1増やす。
 2. セマフォ (S) の待ち行列にプロセス (スレッド) がある場合は、それらの一つを起床させる。
- クリティカルセクションのエグジットセクション等で使用できる。

```
void V(Semaphore S) {
    if (S の待ち行列は空) {
        S++;
    } else {
        一つのプロセスを待ち行列から取り出す;
        そのプロセスを実行可能(Ready)状態にする;
    }
}
```

プロセス同期

17 / 40

セマフォの使用例 (相互排除問題)

```
int account; // スレッド間の共有変数 (残高)
Semaphore accSem = 1; // 初期値1のセマフォaccSem (accountのロック用)
void receiveThread() {
    for ( ; ; ) {
        int receipt = receiveMoney(); // 入金管理スレッドは以下を繰り返す
        P(&accSem); // ネットワークから入金を受信する
        account = account + receipt; // account変数をロックするための P 操作
        V(&accSem); // account変数を変更する(クリティカルセクション)
    }
}
void payThread() {
    for ( ; ; ) {
        int payment = payMoney(); // 引落し管理スレッドは以下を繰り返す
        P(&accSem); // ネットワークから支払い額を受信する
        account = account - payment; // account変数をロックするための P 操作
        V(&accSem); // account変数を変更する(クリティカルセクション)
    }
}
```

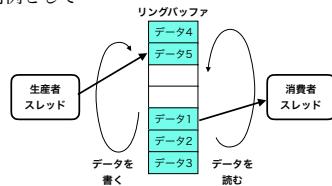
初期値が1のセマフォを用いる。

プロセス同期

18 / 40

生産者・消費者問題

セマフォの使用例として



- 生産者スレッドはデータをバッファに書き込む。
- 消費者スレッドはデータをバッファから読む。
- バッファが溢れないように両者で歩調を合わせる必要がある。

プロセス同期

19 / 40

セマフォの使用例 (生産者・消費者問題)

```

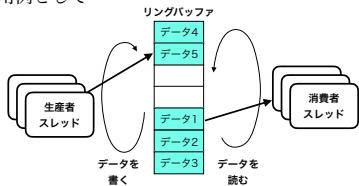
Data      buffer[N];
Semaphore emptySem = N;
Semaphore fullSem = 0;
void producerThread() {
    int in = 0;
    for ( ; ; ) {
        Data d = produce();
        P( &emptySem );
        buffer[ in ] = d;
        in = (in + 1) % N;
        V( &fullSem );
    }
}
void consumerThread() {
    int out = 0;
    for ( ; ; ) {
        P( &fullSem );
        Data d = buffer[ out ];
        out = (out + 1) % N;
        V( &emptySem );
        consume( d );
    }
}
    
```

プロセス同期

20 / 40

複数生産者・複数消費者問題

セマフォの使用例として



- 生産者スレッド同士で相互排除が必要。
- 消費者スレッド同士でも相互排除が必要。

プロセス同期

21 / 40

セマフォの使用例 (複数生産者・複数消費者問題 1/2)

```

Data      buffer[N];
Semaphore emptySem = N;
Semaphore fullSem = 0;
int      in = 0;
Semaphore inSem = 1;
void producerThread() {
    for ( ; ; ) {
        Data d = produce();
        P( &emptySem );
        P( &inSem );
        buffer[ in ] = d;
        in = (in + 1) % N;
        V( &inSem );
        V( &fullSem );
    }
}
    
```

- 複数の生産者スレッドがproducerThread()を実行する。
- in変数は生産者スレッド間の共有変数になった。
- in変数の使用で相互排除するためにinSemを準備した。

プロセス同期

22 / 40

セマフォの使用例 (複数生産者・複数消費者問題 2/2)

```

int out = 0;
Semaphore outSem = 1;
void consumerThread() {
    for ( ; ; ) {
        P( &fullSem );
        P( &outSem );
        Data d = buffer[ out ];
        out = (out + 1) % N;
        V( &outSem );
        V( &emptySem );
        consume( d );
    }
}
    
```

- 複数の消費者スレッドがconsumerThread()を実行する。
- out変数は消費者スレッド間の共有変数になった。
- out変数の使用で相互排除するためにoutSemを準備した。

プロセス同期

23 / 40

リーダ・ライタ問題

次の場合、単にロックするより並行性 (concurrency) を高くできる。



- ライタスレッドはデータ読んでを変更して書き込む。
- データ変更中は他のスレッドはデータにアクセスしてはならない。(排他ロック (exclusive lock))
- リーダスレッドはデータを変更しない。
- 複数のリーダスレッドが同時にデータにアクセスしても良い。しかし、その間、ライタスレッドはデータにアクセスできない。(共有ロック (shared lock))

プロセス同期

24 / 40

セマフォの使用例 (リーダ・ライタ問題 1/2)

データとライタスレッド部分

```
Data records; // 共有するデータ
Semaphore rwSem = 1; // リーダとライタの排他用セマフォ
void writerThread() {
    for ( ; ; ) {
        Data d = produce(); // 新しいデータを作る
        P( &rwSem ); // ライタスレッドは以下を繰り返す
        writeRecords( d );
        V( &rwSem );
    }
}
```

- 複数のライタスレッドがwriterThread() を実行する。
- rwSem=1 は、普通の相互排除と同じ。
- ライタスレッドは、排他ロック (exclusive lock) を用いる。

次ページのスライドがリーダスレッド部分

プロセス同期

25 / 40

セマフォの使用例 (リーダ・ライタ問題 2/2)

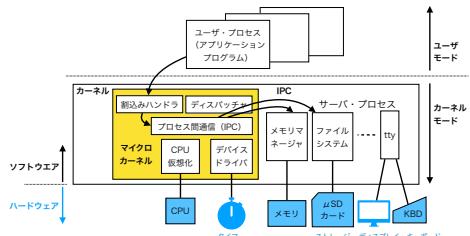
```
int cnt = 0; // リーダ間の共有変数 (読み出し中のリーダ数)
Semaphore cntSem = 1; // cnt の他の制御用セマフォ
void readerThread() {
    for ( ; ; ) {
        P( &cntSem ); // リーダスレッドは以下を繰り返す
        if ( cnt == 0 ) P( &rwSem ); // 自分が最初のリーダなら、代表してロックする
        cnt = cnt + 1; // cnt をインクリメント
        V( &cntSem ); // cnt のロックを外す
        Data d = readRecords(); // データを読みだす
        P( &cntSem ); // cnt にロックを掛ける
        cnt = cnt - 1; // cnt をデクリメント
        if ( cnt == 0 ) V( &rwSem ); // 自分が最後のリーダなら、代表してロックを外す
        V( &cntSem );
        consume( d );
    }
}
```

- 複数のリーダスレッドがreaderThread() を実行する。
- cnt は、クリティカルセクション中のリーダスレッド数。
- cntSem=1 は cnt の相互排除に用いる。
- リーダスレッドは、共有ロック (shared lock) を用いる。

プロセス同期

26 / 40

実装例

第20章
TacOSのセマフォ

プロセス同期

27 / 40

TacOSのセマフォ構造体 (カーネル内)

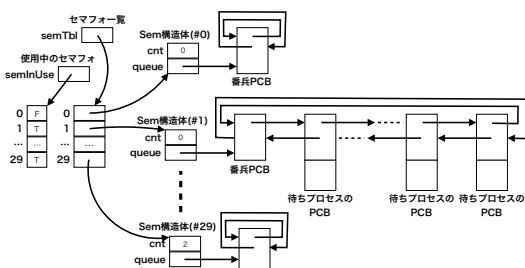
```
#define SEM_MAX 30 // セマフォは最大 30 個
struct Sem { // セマフォを表す構造体
    int cnt; // カウント
    PCB queue; // 待ち行列
};
```

- セマフォは最大 30 個 (TaC のメモリは小さい)
- セマフォ構造体の名前は Sem
- cnt がセマフォの値 (0 以上)
- queue に、このセマフォを待っているプロセスの待ち行列を作る。

プロセス同期

28 / 40

TacOS のセマフォ関連データ構造 (カーネル内)



- TacOS では、セマフォを semTbl のインデックスで識別する。
- Sem 構造体 (#0, #1, #29) は、未使用、待ちあり、待ちなしの例

プロセス同期

29 / 40

TacOS でのセマフォの架空の使用例

```
1 #include <kernel.h> // スレッド間の共有変数 (残高)
2 int account; // account のロック用セマフォの番号
3 int accSem; // プロセスの初期化ルーチン
4 void initProc() { // 初期値1のセマフォを確保する
5     accSem = newSem(1);
6 }
7 void receiveThread() { // 入金管理スレッド
8     for ( ; ; ) {
9         int receipt = receiveMoney(); // ネットワークから入金を受信する
10        semP( accSem ); // account変数をロックするための P 操作
11        account = account + receipt; // account変数を変更する(クリティカルセクション)
12        semV( accSem ); // account変数をロック解除するための V 操作
13    }
14 }
15 void payThread() { // 引落し管理スレッド
16     for ( ; ; ) {
17         int payment = payMoney(); // ネットワークから入金を受信する
18         semP( accSem ); // account変数をロックするための P 操作
19         account = account - payment; // account変数を変更する(クリティカルセクション)
20         semV( accSem ); // account変数をロック解除するための V 操作
21    }
22 }
```

プロセス同期

30 / 40

TacOS のセマフォ割当て解放ルーチン (カーネル内)

```

1 Sem[] semTbl=array(SEM_MAX);           // セマフォの一覧表
2 boolean[] semInUse=array(SEM_MAX);      // どれが使用中か (falseで初期
3
4 // セマフォの割当て
5 public int newSem(int init) {
6     int r = setPri(DI|KERN);             // 割り込み禁止、カーネル
7     for (int i=0; i<SEM_MAX; i=i+1) {    // 全てのセマフォについて
8         if (!semInUse[i]) {              // 未使用のものを見ついたら
9             semInUse[i] = true;           // 使用中に変更し
10            semTbl[i].cnt = init;        // カウントを初期化し
11            setPri(r);                // 割込み状態を復元し
12            return i;                  // セマフォ番号を返す
13        }
14    }
15    panic("newSem");
16    return -1;
17 }
18
19 // セマフォの解放
20 // (書き込み1回で仕事が終わるので割込み許可でも大丈夫)
21 public void freeSem(int s) {
22     semInUse[s] = false;               // 未使用に変更
23 }
```

プロセス同期

31 / 40

TacOS の P 操作ルーチン (カーネル内)

```

1 public void semP(int sd) {           // 割り込み禁止、カーネル
2     int r = setPri(DI|KERN);          // 不正なセマフォ番号
3     if (sd<0 || SEM_MAX<=sd || !semInUse[sd]) {
4         panic("semP(%d)", sd);
5
6     Sem s = semTbl[sd];
7     if (s.cnt>0) {                  // カウンタから引けるなら
8         s.cnt = s.cnt - 1;           // カウンタから引く
9     } else {                         // カウンタから引けないなら
10        delProc(curProc);          // 実行可能列から外し
11        curProc.stat = P_WAIT;       // 待ち状態に変更する
12        insProc(s.queue, curProc);   // セマフォの行列に登録
13        yield();                   // CPUを解放し
14    }                                // 他プロセスに切換える
15    setPri(r);                      // 割込み状態を復元する
16 }
```

プロセス同期

32 / 40

TacOS の PCB リスト操作関数 (カーネル内)

```

1 // プロセスキューでp1の前にp2を挿入する p2 -> p1
2 void insProc(PCB p1, PCB p2) {
3     p2.next=p1;
4     p2.prev=p1.prev;
5     p1.prev=p2;
6     p2.prev.next=p2;
7 }
8
9 // プロセスキュー(実行可能列やセマフォの待ち行列)で p を削除する
10 void delProc(PCB p) {
11     p.prev.next=p.next;
12     p.next.prev=p.prev;
13 }
```

プロセス同期

33 / 40

TacOS の V 操作ルーチン (1/2) (カーネル内)

```

1 // ディスパッチを発生しないセマフォのV操作
2 // (V 操作をしたあとまだ仕事があるとき使用する)
3 // (kernel 内部専用、割込み禁止で呼出す)
4 boolean iSemV(int sd) {
5     if (sd<0 || SEM_MAX<=sd || !semInUse[sd]) {      // 不正なセマフォ番号
6         panic("iSemV(%d)", sd);
7     }
8     boolean ret = false;                               // 起床するプロセスなし
9     Sem s = semTbl[sd];                             // 操作するセマフォ
10    PCB q = s.queue;                            // 待ち行列の番兵
11    PCB p = q.next;                            // 待ち行列の先頭プロセス
12    if (p==q) {                                // 待ちプロセスが無いなら
13        s.cnt = s.cnt + 1;                      // カウンタを足す
14    } else {                                    // 待ち行列から外す
15        delProc(p);                           // 待ちプロセスがあるなら
16        p.stat = P_RUN;                        // 実行可能に変更
17        schProc(p);                          // 実行可能列に登録
18        ret = true;                           // 起床するプロセスあり
19    }
20    return ret;                                 // 実行可能列に変化があった
21 }
```

● iSemV() は割込禁止で呼び出す。

プロセス同期

34 / 40

TacOS の V 操作ルーチン (2/2) (カーネル内)

```

23 // セマフォの V 操作
24 // 待ちプロセス無し : カウンタを1増やす
25 // 待ちプロセス有り : 待ち行列からプロセスを1つ外して実行可能にした後、
26 // ディスパッチャを呼び出す
27 public void semV(int sd) {
28     int r = setPri(DI|KERN);           // 割り込み禁止、カーネル
29     if (iSemV(sd)) {                 // V 操作し必要なら
30         yield();                    // プロセスを切り替える
31     }
32     setPri(r);                      // 割込み状態を復元する
33 }
```

- iSemV() を呼び出す前に割込禁止にする。
- iSemV() が true で返ったらプロセスの切替えを試みる。
- yield() でブリエンプションしたプロセスは、yield() から実行が再開される。

プロセス同期

35 / 40

TacOS の CPU フラグ操作関数 (カーネル内)

```

1 ; CPU のフラグの値を返すと同時に新しい値に変更
2 _setPri
3     ld    g0,2,sp    ; 引数の値を GO に取り出す
4     push  g0          ; 新しい状態をスタックに積む
5     ld    g0.flag    ; 古いフラグの値を返す準備をする
6     reti             ; reti は FLAG と PC を同時に pop する
```

- CPU の PSW のフラグに割込禁止ビットがある。
- C--言語から setPri() 関数として呼び出せるようにするには、アセンブリ言語プログラムで _setPri ラベルを宣言する必要がある。
- C--言語プログラムは引数をスタックに積んで関数を CALL する。
- アセンブリ言語プログラムで引数を参照するには、(SP 相対で) スタックから取り出す。(SP+0 番地が PC, SP+2 番地が第1引数)
- 関数の返り値は、GO レジスターに入れて返す。
- reti 命令はスタックからフラグと PC を一度に取り出す。

プロセス同期

36 / 40

練習問題

練習問題

プロセス同期

37 / 40

練習問題（1）

- 次の言葉の意味を説明しなさい。
 - 競合
 - クリティカルセクション
 - 相互排除
 - ビジーウェイティング
 - ロックフリーなアルゴリズム
 - セマフォ
 - 相互排除問題
 - 生産者と消費者問題
 - リーダライタ問題

プロセス同期

38 / 40

練習問題（2）

- なぜ割込みを禁止することで相互排除ができるか？
- 割込み禁止による相互排除がマルチプロセッサシステムでは不十分な理由は？
- 割込み禁止による相互排除はクリティカルセクションの三つの条件を満たしているか？
- CPU が割込み禁止になっている間に発生した割込みはどのように扱われるか？
- DI 命令や EI 命令が特権命令でなかつたら、どのような不都合が生じるか？
- シングルプロセッサシステムにおいて、機械語命令はアトミック (*atomic*) と言えるか？
- マルチプロセッサシステムにおいて、機械語命令はアトミック (*atomic*) と言えるか？

プロセス同期

39 / 40

練習問題（3）

- TS 命令と SW 命令に共通な特長は何か？
- TS 命令を用いたビジーウェイティングはシングルプロセッサシステムでも使用できるか？
- セマフォを相互排除に使用する手順を説明しなさい。
- 生産者と消費者の問題において、二つのセマフォはどのような値に初期化されたか？
二つのセマフォは何の役割を持っていたか？

プロセス同期

40 / 40

オペレーティングシステム 第6章 プロセス間通信

<https://github.com/tctsigemura/OSTextBook>

プロセス間通信 1 / 30

プロセス間通信の必要性

プロセス間通信 (IPC : Inter-Process Communication)
複数のプロセスが情報を共有し協調して処理を進めることができる。
次のようなメリットが期待できる。

- 複数のプロセスが共通の情報へアクセスすることができる。
- 並列処理による処理の高速化ができる可能性がある。
- システムを見通しの良いモジュール化された構造で構築できる。

プロセス間で情報を共有する代表的な機構として、**共有メモリ**と**メッセージ通信**がある。

プロセス間通信 2 / 30

共有メモリ

● 同じ物理メモリを複数のプロセスの仮想メモリ空間に貼り付ける。
 ● MMU (Memory Management Unit) の働きで可能になる。
 ● 貼り付けが終わればシステムコールなしでデータ交換可能。
 ● プロセス間の同期機構は他に必要。

プロセス間通信 3 / 30

UNIX の共有メモリシステムコール等 (前半)

```
共有メモリなどの識別に使用するキーを生成(ライブラリ)
key_t ftok(const char *path, int id);
  戻り値 : 引数から作成されるキー値
  path   : 実際に存在するファイルのパス
  id     : キーの作成に使用する追加の情報(同じ path から異なるキーを作る)
```

```
共有メモリセグメントの作成(システムコール)
int shmget(key_t key, size_t size, int flag);
  戻り値 : 共有メモリセグメント ID
  key    : キー
  size   : セグメントサイズ(バイト単位)
  flag   : 作成フラグとモード
```

- ftok() は、path と id から一意な key 値を生成する。
- shmget() は、key 値で識別されるメモリセグメント ID を返す。
- shmget() は、メモリセグメントを作ることもできる。
- flag は、rwxrwxrwx と IPC_CREAT 等のフラグ

プロセス間通信 4 / 30

UNIX の共有メモリシステムコール等 (後半)

```
共有メモリセグメントをプロセスの仮想アドレス空間に貼り付ける(システムコール)
void *shmat(int shmid, void *addr, int flag);
  戻り値 : 共有メモリセグメントを配置したアドレス
  shmid  : 共有メモリセグメント ID
  addr   : 貼り付けるアドレス(NULL(0) は、カーネルに任せる)
  flag   : 貼り付け方法等

共有メモリセグメントをプロセスの仮想アドレス空間から取り除く(システムコール)
int shmdt(void *addr);
  戻り値 : 0=正常,-1=エラー
  addr   : 取り除く共有メモリセグメントのアドレス

共有メモリセグメントの制御(システムコール)
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
  戻り値 : 0=正常,-1=エラー
  shmid  : 共有メモリセグメント ID
  cmd    : 削除(IPC_RMID) 等のコマンド
  buf    : コマンドのパラメータ
```

プロセス間通信 5 / 30

UNIX の共有メモリサーバ例 (前半)

```
1 // 共有メモリサーバ(ipcUnixSharedMemoryServer.c):共有メモリからデータを読みだし表示する
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <string.h>
5 #include <unistd.h>
6 #include <sys/types.h>
7 #include <sys/ipc.h>
8 #include <sys/shm.h>
9 #define SHMSZ 512 // 共有メモリのサイズ
10 int main() {
11   key_t key=ftok("shm.dat",'R'); // キーを作る
12   if (key== -1) { // エラーチェック
13     perror("shm.dat");
14     exit(1);
15   }
16   int shmid=shmget(key,SHMSZ,IPC_CREAT|0666); // 共有メモリを作る
17   if (shmid<0) { // エラーチェック
18     perror("shmget");
19     exit(1);
20 }
```

プロセス間通信 6 / 30

UNIX の共有メモリサーバ例 (後半)

```

21  char *data=shmat(shmid,NULL,0);           // 共有メモリを貼り付ける
22  if (data==((char *)-1) {                   // エラーチェック
23      perror("shmatt");
24      exit(1);
25  }
26  strcpy(data, "initialization...\n");       // 共有メモリに書き込む
27  while(1) {                                // 共有メモリの内容を
28      printf("sheared memory:%s",data);        // 5秒に1度メモリを表示
29      if (strcmp(data, "end\n") == 0) break;    // "end"なら終了
30      sleep(5);
31  }
32  if (shmctl(data) == -1){                  // 共有メモリをアドレス空間
33      perror("shmctl");
34      exit(1);
35  }
36  if (shmctl(shmid, IPC_RMID, 0) == -1){    // 共有メモリを廃棄する
37      perror("shmctl");
38      exit(1);
39  }
40  return 0;
41

```

プロセス間通信

7 / 30

UNIX の共有メモリクライアント例 (前半)

```

1 // 共有メモリクライアント(ipcUnixSharedMemoryClient.c):共有メモリにデータを書き込む
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <errno.h>
5 #include <sys/types.h>
6 #include <sys/ipc.h>
7 #include <sys/shm.h>
8 #define SHMSZ 512                         // メモリのサイズ
9 int main() {
10     int shmid;
11     key_t key;
12     char *data, *s;
13     if ((key=fork("shm.dat",'R')) == -1) { // サーバ側と同じキーを作る
14         perror("shm.dat");
15         exit(1);
16     }

```

プロセス間通信

8 / 30

UNIX の共有メモリクライアント例 (後半)

```

17 if ((shmid=shmget(key,SHMSZ,0666))<0) { // 共有メモリを取得する
18     perror("shmget");
19     exit(1);
20 }
21 data=shmat(shmid,NULL,0);                 // 共有メモリを貼り付ける
22 if (data == ((char *)-1) {                // エラーチェック
23     perror("shmat");
24     exit(1);
25 }
26 printf("Enter a string: ");
27 fgets(data,SHMSZ,stdin);                 // 共有メモリに直接入力する
28 if (shmctl(data)==-1){                  // 共有メモリをメモリ空間と
29     perror("shmctl");
30     // 切り離す
31 }
32 return 0;
33

```

プロセス間通信

9 / 30

UNIX の共有メモリプログラム実行例

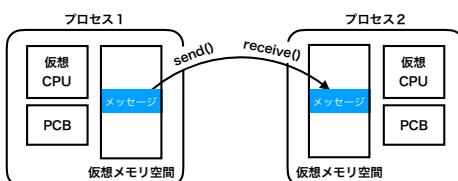
[Terminal No.1]	[Terminal No.2]
\$./ipcUnixShearedMemoryServer	\$./ipcUnixShearedMemoryClient
sheared memory:initialization...	Enter a string: abcdefg
sheared memory:initialization...	\$./ipcUnixShearedMemoryClient
sheared memory:abcdefg	Enter a string: 1234567
sheared memory:abcdefg	\$./ipcUnixShearedMemoryClient
sheared memory:1234567	Enter a string: end
sheared memory:1234567	\$
sheared memory:end	

- このプログラムは相互排除をやっていない。
- このプログラムは使用してはならない。

プロセス間通信

10 / 30

メッセージ通信

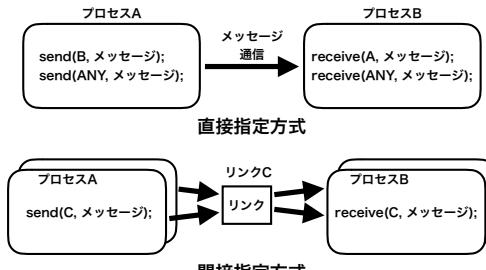


- `send()` システムコールでメッセージを送信する。
- `receive()` システムコールでメッセージを受信する。
- メッセージ通信は同期機構も含んでいる。

プロセス間通信

11 / 30

通信相手の指定方式 (Naming)



- 直接指定方式でも ANY を用いることで多対多通信が可能。
- 間接指定方式は自然に多対多通信が可能

プロセス間通信

12 / 30

通信方式

一般に

- バッファリング（あり／なし）
- メッセージ長（固定／可変）
- メッセージ形式（タグあり／なし）
- 同期方式
 - 非同期方式（ノンブロッキング）
 - 同期方式（ブロッキング）
 - ランデブー方式（クライアント・サーバモデルに特化）

UNIXの場合は

- 間接指定方式
- バッファリング=あり
- メッセージ長=可変長
- メッセージ形式=タグあり
- 同期方式／非同期方式どちらも可能

プロセス間通信

13 / 30

UNIXのメッセージ通信システムコールなど（前半）

```
メッセージ構造体(以下の構造体を自分で宣言して使用する)
struct msgbuf {
    long mtype;           // メッセージの型
    char mtext[N];        // メッセージの本体(N バイト)
};
```

```
メッセージキューの ID を返す。
int msgget(key_t key, int msgflg); (システムコール)
返り値 : メッセージキュー ID
key : キー(ftok() で作成したもの)
msgflg : IPC_CREAT 等のフラグとアクセス許可ビット
```

- mtype がタグの役割を持つ。
- key は共有メモリで紹介したものと同じ (ftok() 関数で作る)。
- 「メッセージキュー」 = 「リンク」
- msgsnd(), msgrcv() で msgflg に IPC_NOWAIT を指定すると **非同期**。

プロセス間通信

14 / 30

UNIXのメッセージ通信システムコールなど（後半）

```
int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
返り値 : 0=正常, -1=エラー
msqid : メッセージキュー ID
msgp : メッセージ構造体のポインタ
msgsz : メッセージ本体のバイト数
msgflg : IPC_NOWAIT 等のフラグ

メッセージキューからメッセージを受信する(システムコール)
int msgrcv(int msqid, const void *msgp, size_t msgsz, long msgtyp, int msgflg);
返り値 : -1=エラー、受信したメッセージの本体バイト数
msqid : メッセージキュー ID
msgp : メッセージ構造体のポインタ
msgsz : メッセージ本体の最大バイト数
msgtyp : 受信するメッセージの型
msgflg : IPC_NOWAIT 等のフラグ

メッセージキューの制御(システムコール)
int msgctl(int msqid, int cmd, struct msgid_ds *buf);
返り値 : -1=エラー、<commd>により異なる
msqid : メッセージキュー ID
cmd : 削除(IPC_RMID)等のコマンド
buf : コマンドのパラメータ
```

プロセス間通信

15 / 30

UNIXのメッセージ通信プログラム例1

```
// ipcUnixMessage.h : メッセージ構造体の宣言
#define MAXMSG 100
struct msgBuf {
    long mtype;           // メッセージ本体の長さ
    char mtext[MAXMSG];  // メッセージ格納用構造体
}; // メッセージの型
// メッセージの本体

// ipcUnixMessageWriter.c : メッセージキューを作成し送信する
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include "ipcUnixMessage.h" // msgBuf 構造体の宣言
int main() {
    struct msgBuf buf; // メッセージ領域
    int msqid;          // メッセージキュー ID
    key_t key;           // メッセージキューの名前
    if ((key=ftok("msgq.dat",'b'))==-1) { // ftok はファイル名から
        perror("ftok");
        exit(1);
    } // 重複のない名前(キー)を生成する
}
```

プロセス間通信

16 / 30

UNIXのメッセージ通信プログラム例2

```
16 if ((msqid=msgget(key,0644|IPC_CREAT))==-1) { // メッセージキューを作る
17     perror("msgget");
18     exit(1);
19 }
20 printf("Enter lines of text, ^D to quit:\n";
21 buf.mtype = 1; // メッセージの型
22 while (fgets(buf.mtext,MAXMSG,stdin)!=NULL) { // キーボードから1行入力
23     if (msgsnd(msqid,&buf,MAXMSG,0)==-1) { // メッセージを送信
24         perror("msgsnd");
25     }
26     break;
27 }
28 if (msgctl(msqid,IPC_RMID,NULL) == -1) { // メッセージキューを削除
29     perror("msgctl");
30     exit(1);
31 }
32 exit(0);
33 }
```

プロセス間通信

17 / 30

UNIXのメッセージ通信プログラム例3

```
// メッセージ受信プログラム(ipcUnixMessageReader) : メッセージキューから受信する
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include "ipcUnixMessage.h"
int main() {
    struct msgBuf buf;
    int msqid;
    key_t key;
    if ((key=ftok("msgq.dat",'b'))==-1) { // 送信側と同じキーを作る
        perror("ftok");
        exit(1);
    }
    if ((msqid=msgget(key,0644))==-1) { // ipcUnixMessageReader が作った
        perror("msgget");
        // メッセージキューを取得
        exit(1);
    }
}
```

プロセス間通信

18 / 30

UNIXのメッセージ通信プログラム例4

```

20 printf("ready to receive messages.\n");
21 for(;;) {
22     if (msgrecv(msqid,&buf,MAXMSG,0,0)==-1) { // 先頭のメッセージを読み出す
23         perror("msgrecv");
24         exit(1);
25     }
26     printf("%ld:%s",buf.mtype,buf.mtext); // 受信したメッセージを表示
27 }
28 exit(0);
29 }

```

```

[Terminal No.1]          | [Terminal No.2]
$ ./ipcUnixMessageWriter | $ ./ipcUnixMessageReader
Enter lines of text, ^D to quit: | ready to receive messages.
abcdefg                   | 1:abcdefg
1234567                   | 1:1234567
^D                         | msgrecv: Identifier removed
$                           |

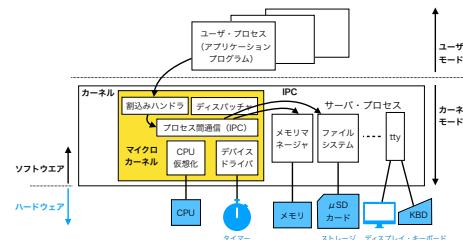
```

プロセス間通信

19 / 30

実装例

第21章 TacOSのメッセージ通信

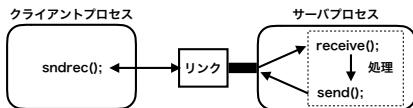


プロセス間通信

20 / 30

TacOSのメッセージ通信機構

クライアント・サーバモデルに特化したランデブー方式のプロセス間通信機構を提供する。



1. サーバプロセスがリンクを所有し通信を待ち受ける。
2. クライアントプロセスは sndrec() 関数でメッセージを送信する。
3. サーバプロセスは receive() 関数を用いてメッセージを受信する。
4. サーバプロセスはメッセージの内容に合った処理を行う。
5. サーバプロセスは処理結果を send() 関数を用いて返信する。
6. sndrec() 関数が完了しクライアントは処理結果を受取る。

プロセス間通信

21 / 30

TacOSのリンク構造体

```

1 #define LINK_MAX 5           // リンクは最大 5 個
2
3 struct Link {              // リンクを表す構造体
4     PCB server;           // リンクを所持するサーバ
5     PCB client;           // リンクを使用中のクライアント
6     int s1;                // サーバがメッセージ受信待ちに使用するセマフォ
7     int s2;                // クライアント同士が相互排除に使用するセマフォ
8     int s3;                // クライアントがメッセージ返信待ちに使用するセマフォ
9     int op;                // メッセージの種類
10    int prm1;              // メッセージのパラメータ 1
11    int prm2;              // メッセージのパラメータ 2
12    int prm3;              // メッセージのパラメータ 3
13 };

```

- リンクはサーバが所有する。
- セマフォを用いて相互排除と同期を行う。
- リンクに書き込めるメッセージの形式は固定。

プロセス間通信

22 / 30

TacOSのリンク作成ルーチン

```

1 Link[] linkTbl = array(LINK_MAX);           // リンクの一覧表
2 int linkID = -1;                            // リンクの通し番号
3
4 // リンクを生成する(サーバが実行する)
5 public int newLink() {
6     int r = setPri(DI[KERN]);
7     linkID = linkID + 1;                      // 通し番号を進める
8 #ifdef DEBUG
9     printf("newLink:ID=%d, SERVER=%d\n", linkID, curProc.pid);
10    if (linkID >= LINK_MAX)                  // リンクが多すぎる
11        panic("newLink");
12
13    Link l = linkTbl[linkID];
14    l.server = curProc;                      // 新しく割り当てるリンク
15    l.s1 = newSem(0);                        // リンクの所有者を記憶
16    l.s2 = newSem(1);                        // server が受信待ちに使用
17    l.s3 = newSem(0);                        // client が相互排他に使用
18    setPri(r);                             // client が受信待ちに使用
19    return linkID;                          // 割り込み復元
20
21 }

```

- 割込み禁止による相互排除を行っている。

プロセス間通信

23 / 30

TacOSのメッセージ通信ルーチン（サーバ用）

```

1 // サーバ用の待ち受けルーチン
2 public Link receive(int num) {
3     Link l = linkTbl[num];
4     if (l.server != curProc) panic("receive");           // 登録されたサーバではない
5     semP(l.s1);                                         // サーバをブロック
6     return l;
7 }
8
9 // サーバ用の送信ルーチン
10 public void send(int num, int res) {
11     Link l = linkTbl[num];
12     if (l.server != curProc) panic("send");             // 登録されたサーバではない
13     l.op = res;                                         // 処理結果を書込む
14     semV(l.s3);                                         // クライアントを起こす
15 }

```

- receive() はリンクにメッセージが届くのを待つ。
- receive() はメッセージが書き込まれたリンクを返す。
- send() はリンクに処理結果 (16bit) を返信する。

プロセス間通信

24 / 30

TacOS のメッセージ通信使用例（サーバ側）

```

1 // プロセスマネージャーのメインループ
2 public void pmMain() {
3     pmLink = newLink(); // リンクを生成する
4     while (true) { // システムコールを待つ
5         Link l = receive(pmLink); // システムコールを受信
6         int r=pmSysCall(l.op,l.prm1,l.prm2,l.prm3,l.client); // システムコール実行
7         send(pmLink, r); // 結果を返す
8     }
9 }
```

- プロセスマネージャ（サーバプロセス）の例。
- サーバはリンクを作成した後、受信、処理、返信を繰り返す。
- `receive()` を用いてリンクからメッセージを受信。
- `pmSysCall()` がプロセスマネージャの処理ルーチン。
- `send()` を用いてクライアントに処理結果を返す。

プロセス間通信

25 / 30

TacOS のメッセージ通信ルーチン（クライアント用）

```

1 // クライアント用メッセージ送受信ルーチン
2 public int sndrec(int num, int op, int prm1, int prm2, int prm3) {
3     Link l = linkTbl[num]; // 他のクライアントと相互
4     semP(l.s2); // 排除リンクを確保
5     l.client = curProc; // リンク使用中プロセス記録
6     l.op = op; // メッセージを書込む
7     l.prm1 = prm1;
8     l.prm2 = prm2;
9     l.prm3 = prm3;
10    int r = setPri(DI|KERN); // 割り込み禁止、カーネル
11    iSemV(l.s1); // サーバを起こす
12    semP(l.s3); // 返信があるまでブロック
13    setPri(r); // 割り込み復元
14    int res = l.op; // 返信を取り出す
15    semV(l.s2); // リンクを解放
16    return res;
17 }
```

- `iSemv()` を使用するので割り込み禁止による相互排除が必要。
- クライアント間での相互排除にセマフォ `s2` を使用。

プロセス間通信

26 / 30

TacOS のメッセージ通信使用例（クライアント側）

```

1 public int exec(char[] path, char[] argv) {
2     int r=sndrec(pmLink,EXEC,_AtoI(path),_AtoI(argv),0);
3     return r; // 新しい子の PID を返す
4 }
```

- `exec` システムコールを例にする。
- `exec` は `path` のプログラムを新しいプロセスで実行する。
- 上のプログラムは SVC 割込みハンドラから呼出される。
- SVC 割込みハンドラはシステムコールの種類を判断し、`exec` システムコールの場合に上のルーチンを呼出す。
- 割込みハンドラはプロセスのコンテキストで実行される。
- `exec` システムコールはプロセスマネージャが処理する。
- プロセスマネージャへのメッセージ通信により処理を依頼する。
- `_AtoI()` 関数は参照（アドレス）を `int` 型に変換する。

プロセス間通信

27 / 30

練習問題

練習問題

プロセス間通信

28 / 30

練習問題（1）

- 次の言葉の意味を説明しなさい。
 - 共有メモリ
 - メッセージ通信
 - 直接指定方式
 - 間接指定方式
 - パッファリング
 - 同期方式
 - 非同期方式
 - ランデブー方式
 - メッセージのタグ

プロセス間通信

29 / 30

練習問題（2）

- プロセス間の共有メモリとスレッド間の共有変数の違いは何か？
- UNIX の共有メモリ使用例を実際に実行し動作確認しなさい。
- 動作確認したプログラムでは、サーバプログラムは共有メモリが変更されたことを確認しないで、一定の時間間隔で共有メモリの内容を表示している。
 - どのような不都合が予想されるか？
 - クライアントとサーバで同期をする方法はあるか？
- メッセージ通信でバッファを大きくすることのメリットは何か？
- UNIX のメッセージ通信プログラム例を実際に実行し動作確認しなさい。
- UNIX のメッセージ通信プログラム例は生産者と消費者の問題の解になっている。複数生産者と複数消費者の問題の解にもなっているか？
- UNIX のメッセージ通信プログラム例が複数生産者と複数消費者の問題の解にもなっているか、動作確認する手順を説明しなさい。

プロセス間通信

30 / 30

オペレーティングシステム 第7章 モニタ

<https://github.com/tctsigemura/OSTextBook>

モニタ

1 / 20

モニタ（Monitor）の概要

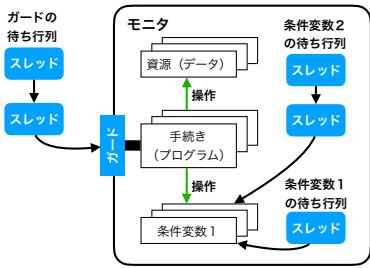
プロセス間同期のために使用できる高級言語の機能のこと。
モニタは抽象データ型（Java言語のクラスのようなもの）である。

- プログラマが定義できる型である。（抽象データ型で一般的）
- データと操作をまとめて定義する。（抽象データ型で一般的）
- 同期のための機能が組込まれている。（モニタ独特）

モニタ

2 / 20

モニタの模式図



- 資源（データ、変数）、手続き（メソッド）、ガード、条件変数
- 図では省略してあるが初期化プログラム

モニタ

3 / 20

モニタの構成要素

資源（データ、変数） 複数のスレッドによって共有される変数のことである。モニタの外から直接アクセスすることはできない。

手続き（操作、メソッド） 外部から呼び出されるプログラムである。手続きの実行はガードの働きにより排他的に行われ、同時に実行される手続きは必ず一つ以内である。

ガード モニタに一つのガードが存在し、手続きを排他的に実行するために用いられる。

条件変数 条件変数には wait と signal の二つの操作ができる。wait 操作を行ったスレッドはガードを外して条件変数の待ち行列に入る。signal 操作は条件変数の待ち行列から一つのスレッドを選んで実行可能にする。

初期化プログラム モニタのインスタンスを作成する時に、初期化のため実行されるプログラムである。

モニタ

4 / 20

モニタによる相互排除の実現（仮想言語）

```

1 // 銀行口座の残高を管理するモニタ
2 monitor MonAccount {
3     // 資源
4     int money; // スレッド間の共有変数(残高)
5     // 初期化プログラム
6     MonAccount(int m) {
7         money = m; // 口座の残高を初期化する
8     }
9     // 手続き
10    public void receive(int r) { // 入金手続き
11        money = money + r;
12    }
13    public void pay(int p) { // 引落手手続き
14        money = money - p;
15    }
16}

```

- 資源はスレッド間の共有変数 money である。
- 初期化プログラムはモニタのインスタンス生成時に実行される。
- 手続き（クリティカルセクション）は、ガードにより自動的に相互排除される。

モニタ

5 / 20

モニタによる相互排除の利用（仮想言語）

```

1 class MonAccountMain {
2     static MonAccount account = new MonAccount(0); // 残高 0円の口座を作る
3     static void receiveThread() {
4         for ( ; ; ) {
5             int receipt = receiveMoney(); // 以下を繰り返す
6             account.receive(receipt); // ネットワークから入金を受信
7         }
8     }
9     static void payThread() {
10        for ( ; ; ) {
11            int payment = payMoney(); // 以下を繰り返す
12            account.pay(payment); // ネットワークから支払いを受信
13        }
14    }
15    public static void main(String[] args) {
16        receiveThread(); // 入金管理スレッドを起動;
17        payThread(); // 引落し管理スレッドを起動;
18    }
}

```

- モニタのインスタンスを生成し、それに対して操作する。
- ガードは自動的ないので排他忘が無い。

モニタ

6 / 20

セマフォを用いた相互排除問題の解 (参考)

```

int account; // スレッド間の共有変数(残高)
Semaphore accSem = new Semaphore(1); // 初期値1のセマフォ accSem(accountのロック用)
void receiveThread() {
    for (; ; ) {
        int receipt = receiveMoney(); // 入金管理スレッドは以下を繰り返す
        P(&accSem); // account変数をロックするためのP操作
        account = account + receipt; // account変数を更新する(クリティカルセクション)
        V(&accSem); // account変数をロック解除するためのV操作
    }
}
void payThread() { // 引落し管理スレッド
    for (; ; ) {
        int payment = payMoney(); // ネットワークから支払い額を受信する
        P(&accSem); // account変数をロックするためのP操作
        account = account - payment; // account変数を更新する(クリティカルセクション)
        V(&accSem); // account変数をロック解除するためのV操作
    }
}

```

- P操作とV操作の使用はプログラマまかせ。
- 間違うとタイミング依存の発見の難しいバグになる。

モニタ

7 / 20

モニタによるキューの実現 (仮想言語、前半)

```

1 monitor BoundedBuffer {
2     // 資源(リングバッファ)
3     int N;
4     int[] buf;
5     int first, last, cnt;
6     // 条件変数
7     Condition empty;
8     Condition full;
9     // 初期化
10    BoundedBuffer(int n) {
11        N = n;
12        buf = new int[N];
13        first = last = cnt = 0;
14    }

```

- この例ではリングバッファのデータ構造が資源である。
- 資源はモニタ外部から直接アクセスすることはできない。
- 条件変数emptyはキューが空の時、消費者スレッドが待合せに使用。
- 条件変数fullはキュー満杯時に、生産者スレッドが待合せに使用。

モニタ

8 / 20

モニタによるキューの実現 (仮想言語、後半)

```

15 // 手続き
16 public void append(int x) { // (1)
17     if (cnt==N) full.wait(); // (1)
18     buf[last] = x; // (3)
19     last = (last + 1) % N; // (3)
20     cnt++; // (3)
21     empty.signal(); // (3)
22 }
23 public int remove() { // (2)
24     if (cnt==0) empty.wait(); // (2)
25     int x = buf[first]; // (2)
26     first = (first + 1) % N; // (2)
27     cnt--; // (2)
28     full.signal(); // (2)
29     return x; // (4)
30 }
31

```

- wait()はスレッドを状態変数の待ち行列に入れる。
- signal()は状態変数の待ちスレッドを起こす。
- signal()で起床したスレッドはただちに実行される。

モニタ

9 / 20

モニタによる生産者と消費者問題の解 (仮想言語)

```

1 class BoundedBufferMain {
2     static BoundedBuffer queue = new BoundedBuffer(10); // 大きさ10のキュー
3     static void producer() { // 生産者スレッドが実行
4         while(true) {
5             int x = データを作る(); // キューにデータを追加
6             queue.append(x);
7         }
8     }
9     static void consumer() { // 消費者スレッドが実行
10        while(true) {
11            int x = queue.remove(); // キューからデータを取り出す
12            データを使用する(x);
13        }
14    }
15 public static void main(String[] args) { // mainから実行を開始
16     生産者スレッドを起動;
17     消費者スレッドを起動;
18 }
19

```

- モニタにより定義されたキューのインスタンスを使用した解

モニタ

10 / 20

モニタと同等なキューをJavaのセマフォで実現 (1/4)

```

1 import java.util.concurrent.Semaphore; // セマフォ型を利用可能にする
2 public class SemBoundedBuffer {
3     private Semaphore guard = new Semaphore(1); // ガード用のセマフォ
4     private Semaphore next = new Semaphore(0); // signal時ブロック・スレッド用セマフォ
5     private int nextCount = 0; // signal時ブロック・スレッド数
6     private class Condition { // 内部クラス! 条件変数型を定義
7         Semaphore sem = new Semaphore(0); // 条件変数待ち用セマフォ sem
8         int count = 0; // 条件変数を待つスレッドの数
9         void await() { // 条件変数を待つメソッド
10             count++;
11             if (nextCount>0) { // この条件変数待ちスレッドの数
12                 next.release(); // signal()したスレッドを起床
13             } else { // 起床後にawait()した場合なら
14                 guard.release(); // ガードを外してからブロック
15             }
16             sem.acquireUninterruptibly(); // 条件変数のセマフォで待つ
17             count--;
18         }

```

- Javaのセマフォはカウンティングセマフォ(計数セマフォ)
- P操作: acquireUninterruptibly(), V操作: release()

モニタ

11 / 20

モニタと同等なキューをJavaのセマフォで実現 (2/4)

```

19 void signal() { // 条件変数で待つスレッドを起床
20     if (count>0) { // 待っているスレッドがあれば
21         nextCount++; // signal途中のスレッド数
22         sem.release(); // 待ちスレッドを起こす
23         next.acquireUninterruptibly(); // 起きたスレッドを先に実行
24         nextCount--;
25     }
26 }
27
28 private void exitProc() { // 終了の出口処理
29     if (nextCount>0) { // signalされた後なら
30         next.release(); // signalしたスレッドを起床
31     } else { // そうでなければ
32         guard.release(); // ガードを外す
33     }
34 }

```

- 条件変数型を内部クラスとして定義
- wait()の代わりにawait()を定義
- exitProc()は手続きの最後で呼出す。

モニタ

12 / 20

モニタと同等なキューをJavaのセマフォで実現(3/4)

```

35 // 資源(リングバッファ)
36 private int N;
37 private int[] buf;
38 private int first, last, cnt;
39 // 条件変数
40 private Condition empty = new Condition();
41 private Condition full = new Condition();
42 // 初期化
43 public SemBoundedBuffer(int n) {
44     N = n;
45     buf = new int[N];
46     first = last = cnt = 0;
47 }

```

- 資源はクラス外から見えないように `private` にする。
- 前半で宣言した条件変数型の変数を二つ使用。
- 初期化プログラムはクラスのコンストラクタで実現。

モニタ

13 / 20

モニタと同等なキューをJavaのセマフォで実現(4/4)

```

48 // 手続き
49 public void append(int x) {           // (1)
50     guard.acquireUninterruptibly();    // (1) ガードを取得
51     if (cnt==N) full.await();          // (1)
52     buf[last] = x;                  // (3)
53     last = (last + 1) % N;           // (3)
54     cnt++;                          // (3)
55     empty.signal();                // (3)
56     exitProc();                     // (3) 手続きの出口処理
57 }
58 public int remove() {                 // (2)
59     guard.acquireUninterruptibly();    // (2) ガードを取得
60     if (cnt==0) empty.await();          // (2)
61     int x = buf[first];              // (2)
62     first = (first + 1) % N;           // (2)
63     cnt--;                           // (2)
64     full.signal();                  // (2)
65     exitProc();                     // (4) 手続きの出口処理
66     return x;                        // (4)
67 }

```

- モニタなら自動的に実行されるものも明示

モニタ

14 / 20

Javaによる生産者と消費者問題の解(前半)

```

1 public class MonBoundedBuffer {
2     // 資源(リングバッファ)
3     private int N;
4     private int[] buf;
5     private int first, last, cnt;
6     // 初期化
7     public MonBoundedBuffer(int n) {
8         N = n;
9         buf = new int[N];
10        first = last = cnt = 0;
11    }
12    // 手続き
13    private void await() {
14        try{wait();}catch(InterruptedException e){}
15    }

```

- 資源には `private` を明示
- 初期化はコンストラクタにより実現
- Java の `wait()` は `try-catch` 構文で使用する必要がある。

モニタ

15 / 20

Javaによる生産者と消費者問題の解(後半)

```

16 public synchronized void append(int x) { // (1)
17     while (cnt==N) await();           // (1)
18     buf[last] = x;                  // (3)
19     last = (last + 1) % N;           // (3)
20     cnt++;                          // (3)
21     if (cnt==1) notify();           // (3)
22 }
23 public synchronized int remove() {      // (2)
24     while (cnt==0) await();           // (2)
25     int x = buf[first];              // (2)
26     first = (first + 1) % N;           // (2)
27     cnt--;                           // (2)
28     if (cnt==N-1) notify();           // (2)
29     return x;                        // (2)
30 }
31 }

```

- 手続きは `synchronized` 修飾したメソッド
- Java の `wait()` は別の理由でも終了するので `while` の中に使用
- 条件変数は一つしかないため、利用方法に工夫が必要
- `remove()` 最後の行の実行順序がモニタと異なる。

モニタ

16 / 20

練習問題

練習問題

モニタ

17 / 20

練習問題(1)

- 次の言葉の意味を説明しなさい。
 - 抽象データ型
 - モニタ
 - ガード
 - 資源
 - 手続き(操作、メソッド)
 - 条件変数
 - 初期化プログラム

モニタ

18 / 20

練習問題（2）

- 抽象データ型の定義を調べなさい。
- 「モニタによるキューの実現（仮想言語版）」は、cntなしでも記述できるか？
- 「モニタによるキューの実現（仮想言語版）」において、キューが空のとき一つのスレッドがremove()を実行した。その後、別のスレッドがappend()を実行した。この時のappend(), remove()内のプログラムが実行される順を答えなさい。
- モニタによるキューの実現（仮想言語版）は、複数生産者と複数消費者問題の解に使用できるか？
- Java風仮想言語のモニタを用いてリーダ・ライタ問題の解を示しなさい。
- Java風仮想言語のモニタを用いてセマフォを記述しなさい。

モニタ

19 / 20

練習問題（3）

- semBoundedBufferを実際に実行しなさい。
(メインルーチンを含むソースプログラムの入手先は教科書参照)
- signal()は手続きの最後でしか使用できないことになると、semBoundedBufferはどのように簡略化できるか。
- MonBoundedBufferを実際に実行しなさい。
(メインルーチンを含むソースプログラムの入手先は教科書参照)
- 「モニタによるキューの実現（仮想言語版）」と、「Javaのモニタ風機構による生産者と消費者問題の解」では、プログラム中のコメントで示したように実行順序が異なる。Javaのモニタ風機構と従来のモニタのどのような違いによるものか？
- その他に従来のモニタとJavaのモニタ風機構の違いは何があるか？
- モニタのsignalと、セマフォのV操作の違いは何があるか？

モニタ

20 / 20

オペレーティングシステム 第8章 主記憶（メモリ）

<https://github.com/tctsigemura/OSTextBook>

主記憶

1 / 15

主記憶

主記憶はプログラム、データ、スタック等を置くメモリのこと
CPUと同様に重要な装置

- TeC の 256 バイトの RAM
- H8/3664 の 32KiB の ROM と 2KiB の RAM
- PC のメモリ（4GiB ~ 16GiB 程度？）

この章では、主記憶を管理し複数のプロセスに適切に割り振り、かつ、プロセス同士が干渉しないように分離する方法を学ぶ。

主記憶

2 / 15

ハードウェア構成

メモリを共有する SMP (Symmetric Multiprocessing) システム

この講義で前提にしているコンピュータの構成

主記憶

3 / 15

本章で用いるモデル

(a) 単純なモデル

(b) 仮想化が可能なモデル

MMU (Memory Management Unit : メモリ管理装置)

- メモリ保護機構、メモリ再配置機構、仮想記憶
- 仮想アドレス、物理アドレス

主記憶

4 / 15

メモリ保護機構

- CPUを仮想化した
複数のプロセスを同時にロードし並列実行できるようになった
- ユーザーがプロセスが複数存在する
プロセスが他のプロセスやOSを破壊しないか?
他のプロセスのメモリを保護する機構が必要

プロセスは自身に割当てられたメモリ以外をアクセスできないようにする。

主記憶

5 / 15

上限・下限レジスタ

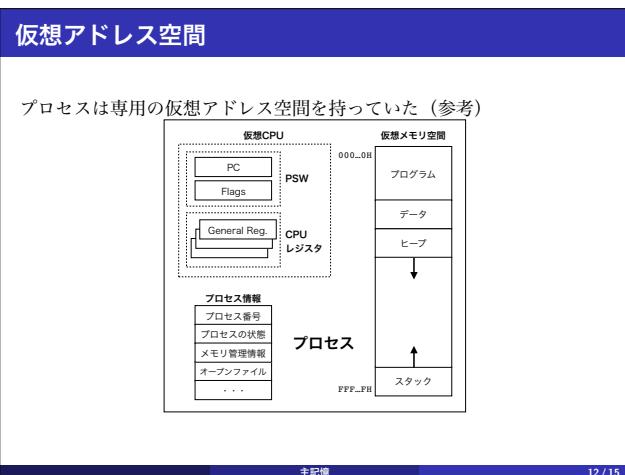
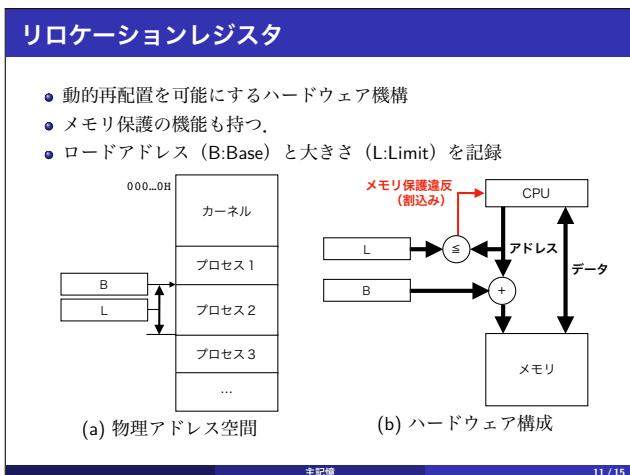
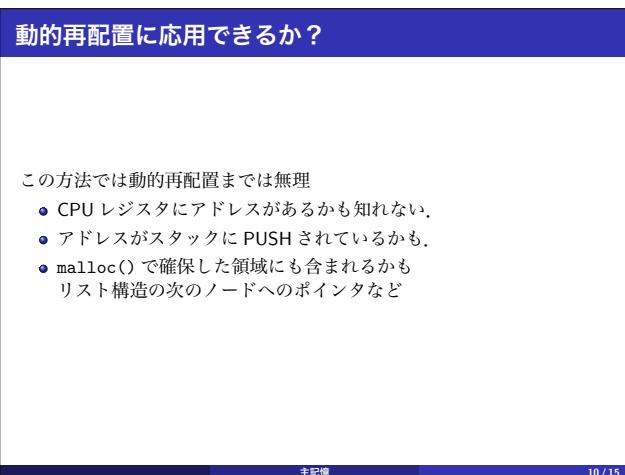
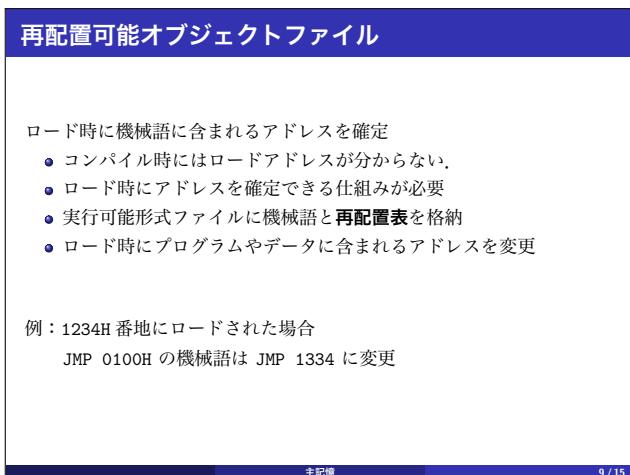
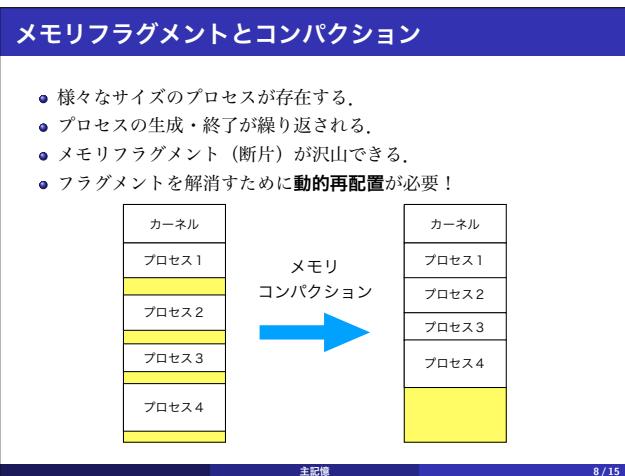
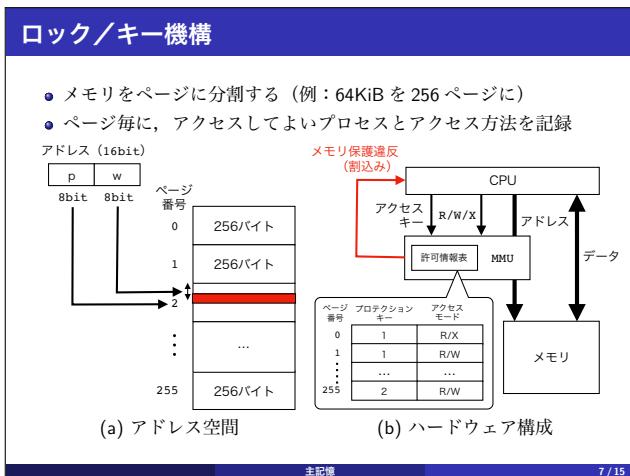
- プロセスがアクセスしても良いアドレスの範囲を設定する。
- プロセスのメモリアクセスはアドレスをチェックする（ハード）
- レジスタを操作できるのはカーネルだけ。

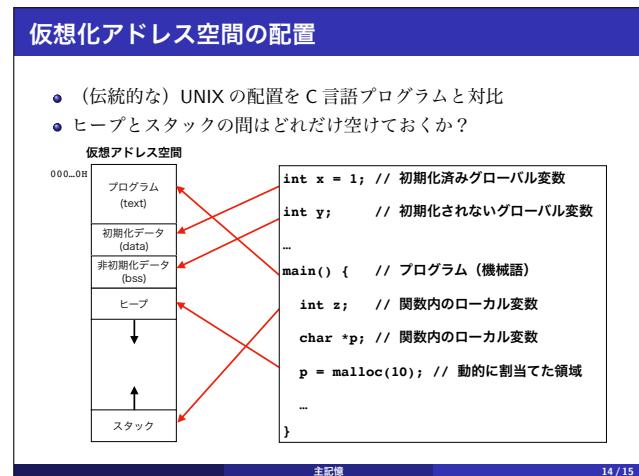
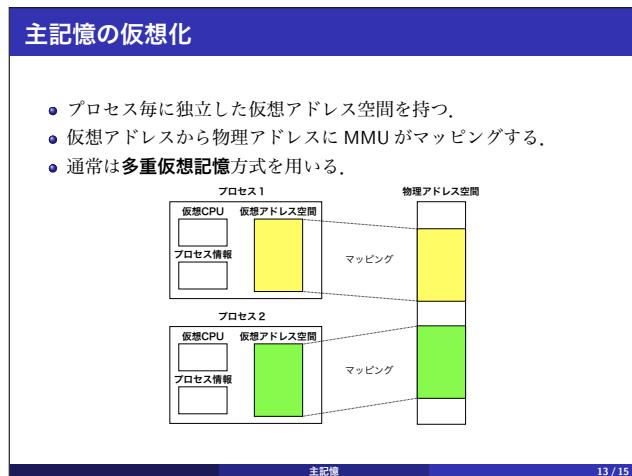
(a) 物理アドレス空間

(b) ハードウェア構成

主記憶

6 / 15





前のプログラムをアセンブリ言語に変換したもの

```

_x DW 1 // int x = 1;
_y WS 1 // int y;
_main PUSH FP // void main() {
    LD FP,SP // 機械語命令は
    PUSH G3 // text セグメントに出力
    PUSH G4 // char *p;
    LD G0,#10 //
    PUSH G0 //
    CALL _malloc // p = malloc(10);
    ADD SP,#2 //
    LD G4,G0 //
    POP G4
    POP G3
    POP FP
    RET // }

```

主記憶 15 / 15

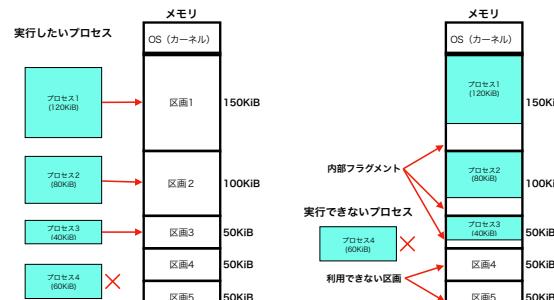
オペレーティングシステム 第9章 メモリ割付け方式

<https://github.com/tctsigemura/OSTextBook>

メモリ割付け方式

1 / 19

固定区画方式



メモリ割付け方式

2 / 19

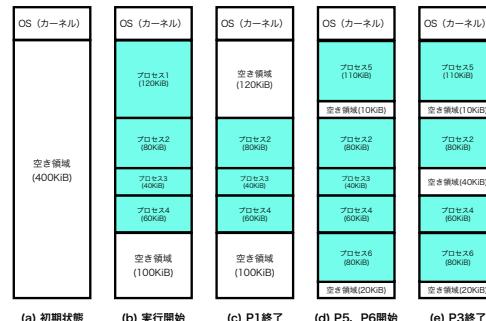
固定区画方式の特徴

- ① 空き領域の管理が容易である。
- ② 領域内部に無駄な領域（内部フラグメント）が生じる。
- ③ 小さな領域が複数空いていても大きなプロセスは実行できない。
- ④ 実行可能なプロセスのサイズに強い制約がある。
(図の例では、151KiBのプロセスは実行できない。)
- ⑤ 同時に実行できるプロセスの数に制約がある。
(図の例では、同時に五つ以上のプロセスは実行できない。)

メモリ割付け方式

3 / 19

可変区画方式

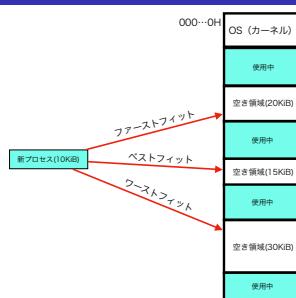


必要に応じて空き領域から区画を作る。
外部フラグメントが生じる。

メモリ割付け方式

4 / 19

可変区画方式の領域選択方式



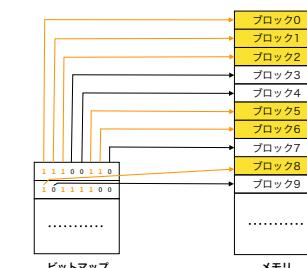
- ファーストフィット (first-fit) 方式：アドレス順にさがす。
- ベストフィット (best-fit) 方式：最小の領域を選択する。
- ワーストフィット (worst-fit) 方式：最大の領域を選択する。

メモリ割付け方式

5 / 19

空き領域の管理方式 (ビットマップ方式)

どこに利用可能な空き領域があるかビットマップで管理する。



- メモリを一定の大きさのブロックに分割する。
- ビットマップの1ビットが1ブロックに対応する。

メモリ割付け方式

6 / 19

ビットマップの大きさ

ビットマップの大きさを計算してみる。

- メモリサイズ : 8GiB
- ブロックサイズ : 4KiB
- ブロック数 : $8GiB \div 4KiB = (8 \times 2^{30}) \div (4 \times 2^{10}) = 2 \times 2^{20} = 2Mi$
- ビットマップのサイズ : $(2 \times 2^{20}) \div 8 = 2^{18} = 256KiB$

無視できるほど小さくはない。

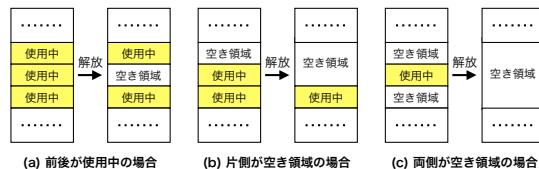
ビットマップを小さくするにはブロックサイズを大きくすればよい。
内部フラグメントが大きくなる。

メモリ割付け方式

7 / 19

空き領域の管理方式 (リスト方式)

空き領域をリストにして管理する。
様々なサイズの空き領域が混在しても良い。

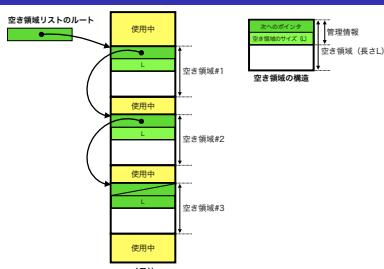


使用中の領域が解放されると隣接する空き領域と合体させる。

メモリ割付け方式

8 / 19

空き領域リストのデータ構造



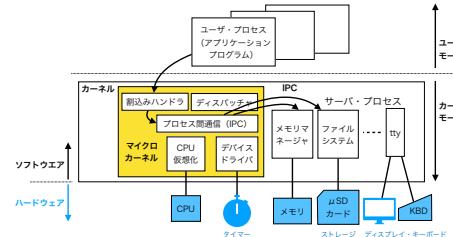
- 空き領域の一部を管理データの格納に使用する。
- アドレス順のリストにして管理する。
- ファーストフィットの探索に都合が良い。
- 隣接領域との合体にも都合が良い。

メモリ割付け方式

9 / 19

実装例

第22章 TacOS のメモリ管理



メモリ割付け方式

10 / 19

メモリ管理の実装例

TacOS のメモリ管理プログラムを実装例とする。

- 可変区画方式
- ファーストフィット
- OS がユーザプロセス領域の割当てに使用
- プロセスがヒープ領域を管理するプログラムも同じアルゴリズム

メモリ割付け方式

11 / 19

データ構造の初期化

```

1 #define MBSIZE sizeof(MemBlk)           // MemBlk のバイト数
2 #define MAGIC (memPool)               // 番兵のアドレスを使用する
3
4 // 空き領域はリストにして管理される
5 struct MemBlk {                     // 空き領域管理用の構造体
6     MemBlk next;                    // 次の空き領域アドレス
7     int size;                      // 空き領域サイズ
8 };
9
10 //-----
11 // 初期化ルーチン
12 //-----
13 // メモリ管理の初期化
14 MemBlk memPool = {null, 0};          // 空き領域リストの番兵
15 public int _end();                  // カーネルの BBS 領域の最後
16
17 void mmInit() {                   // プログラム起動前の初期化
18     memPool.next = _ltoA(addrOf(_end)); // 空き領域
19     memPool.next.size = 0xe000 - addrOf(_end); // 空きメモリサイズ
20     memPool.next.next = null;
21 }

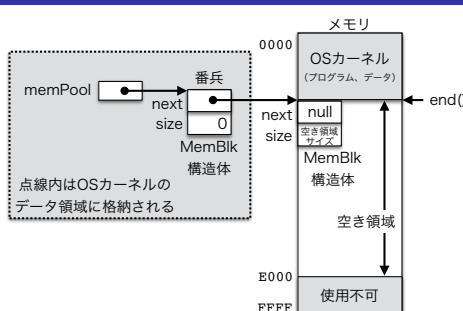
```

- `mmInit()` はカーネル起動時に一度だけ実行される。

メモリ割付け方式

12 / 19

初期化直後のデータ構造



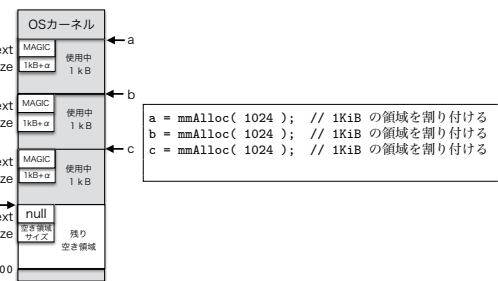
- `_end()` はカーネルサイズにより決まる。
- E000 より後ろはビデオメモリや IPL ROM がある。

メモリ割付け方式

13 / 19

メモリの割付け

右のプログラムで a, b, c を割付けたときのデータ構造



メモリ割付け方式

14 / 19

メモリの割付けプログラム

```

1 // メモリを割り付ける
2 int mmAlloc(int size) {
3     int s = (size + MBSIZE + 1) & -1;
4     MemBlk p = memPool;
5     MemBlk m = p.next;
6
7     while (_aCmp(m.size,s)<0) {           // 領域が小さい間
8         p = m;                            // リストを続ける
9         if (m==null) return 0;             // エラーを表す null ポインタ
10    }
11
12    if (_aCmp(p.size ,s+MBSIZE+2)<=0) { // 分割する領域がない領域サイズ
13        if (memPool.next==m && m.next==null) // リストの長さがゼロにならない
14            return 0;                      // ようにする
15        p.next = m.next;                 // リストから外す
16        p.size = m.size;                // 領域を分割する値がある
17        m.size = 0;                     // 残り領域
18        MemBlk n = _addrAdd(m, s);
19        n.next = m.next;
20        n.size = m.size - s;
21        p.next = n;
22        p.size = s;
23    }
24    m.next = MAGIC;                    // マジックナンバーグローバル
25    return _addrAdd(m, MBSIZE);        // 管理領域を除いて返す
26}

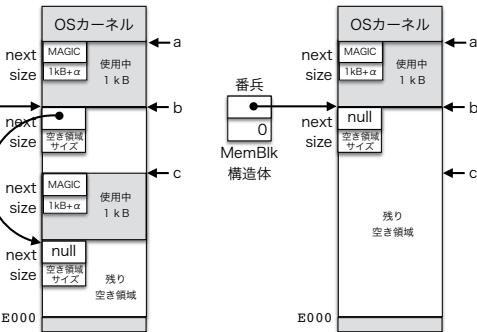
```

メモリ割付け方式

15 / 19

領域の解放

b, c を開放したときのデータ構造



メモリ割付け方式

16 / 19

メモリの解放プログラム (前半)

```

1 // メモリを解放する
2 int mmFree(void* mem) {
3     MemBlk q = _addrAdd(mem, -MBSIZE); // 領域解放
4     MemBlk p = memPool;               // 解放する領域
5     MemBlk m = p.next;               // 直前の空き領域
6
7     if (q.next!=MAGIC)              // 領域マジックナンバー確認
8         badaddr();                  // 領域マジックナンバー確認
9
10    while (_aCmp(m, q)<0) {        // 解放する領域の位置を探る
11        p = m;
12        m = m.next;
13        if (m==null) break;
14    }
15
16    void* ql = _addrAdd(q, q.size); // 解放する領域の最後

```

- 領域の本当の先頭アドレスを計算する。
- MAGIC を確認する。
- 空き領域リストを辿り、挿入位置を決める。

メモリ割付け方式

17 / 19

メモリの解放プログラム (後半)

```

17     void* pl = _addrAdd(p, p.size); // 直前の領域の最後
18
19     if (_aCmp(q,pl)<0 || m!=null&&_aCmp(m,ql)<0) // 未割り当て領域では？
20         badaddr();
21
22     if (pl==q) {                   // 直前の領域に隣接している
23         p.size = p.size + q.size;
24         if (q==m) {                // 直後の領域とも隣接している
25             p.size = p.size + m.size;
26             p.next = m.next;
27         }
28     } else if (ql==m) {           // 直後の領域に隣接している
29         q.size = q.size + m.size;
30         q.next = m.next;
31         p.next = q;
32     } else {                      // 他の場合
33         p.next = q;
34         q.next = m;
35     }
36     return 0;
37 }

```

メモリ割付け方式

18 / 19

練習問題

可変区画方式で管理される 100KiB の空き領域がある時、次の順序で領域の割付け解放を行った。ファーストフィット方式を用いた場合とベストフィット方式を用いた場合について、実行後のメモリマップを図示しなさい。

- 1 30KiB の領域を割付け
- 2 40KiB の領域を割付け
- 3 20KiB の領域を割付け
- 4 先程割付けた 40KiB の領域を解放
- 5 10KiB の領域を割付け

オペレーティングシステム 第10章 セグメンテーション

<https://github.com/tctsigemura/OSTextBook>

セグメンテーション 1 / 18

リロケーションレジスタ方式 (復習)

プロセスの仮想アドレス空間を物理アドレス空間にマッピングする。

- プロセス毎に独立した仮想アドレスを持つ。
- 仮想アドレス空間はいつも0番地から開始できる。
- 動的なメモリコンパクションができる。

セグメンテーション 2 / 18

リロケーションレジスタ方式の問題点 (1/2)

プロセスの仮想アドレス空間は次のような領域から構成された。

- 必要なメモリの見積もりが難しい。
実行時にヒープ領域やスタック領域が不足する可能性がある。
実行前に必要なメモリサイズを見積もる必要があり使い勝手が悪い。

セグメンテーション 3 / 18

リロケーションレジスタ方式の問題点 (2/2)

プロセスの仮想アドレス空間は次のような領域から構成された。

- 領域の性質応じたメモリ保護ができない。
 - プログラム領域は読み出しと実行だけ許可する。
 - データ、ヒープ、スタック領域に読み出しと書き込みだけ許可する。

セグメンテーション 4 / 18

セグメント

プロセスに複数のアドレス空間を持たせる。

- 複数持つことができるアドレス空間=セグメント
 - 前の例で「プログラム」、「データ」等をセグメントにする。
 - セグメント毎にサイズや保護モード (rwx) を決める。
 - セグメント番号とセグメント内アドレスの二次元空間になった。

仮想アドレス空間内の配置問題が解決！！

セグメンテーション 5 / 18

セグメント番号

セグメント番号はどこから供給するのか？

- 機械語命令にセグメント番号を持たせる。

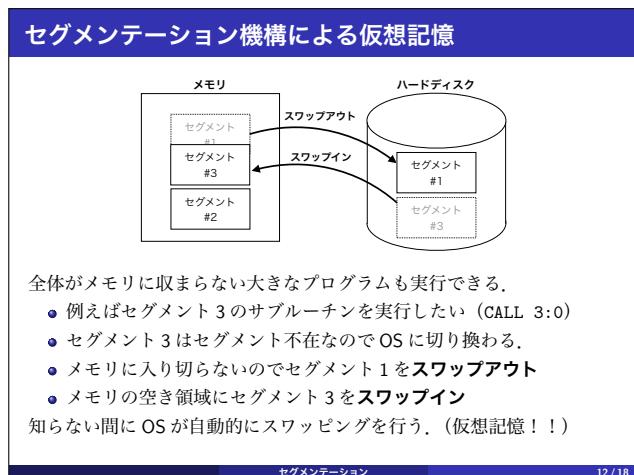
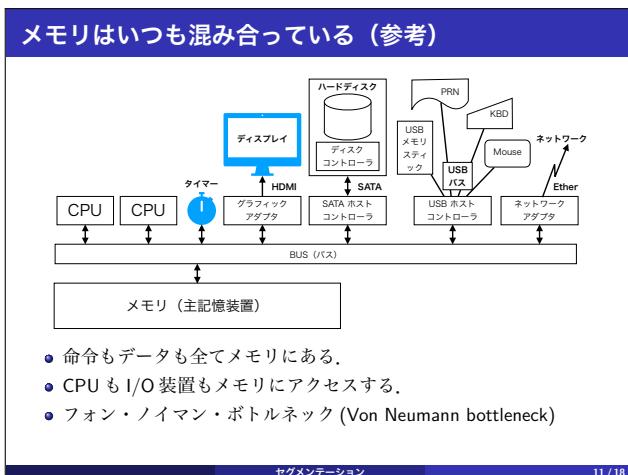
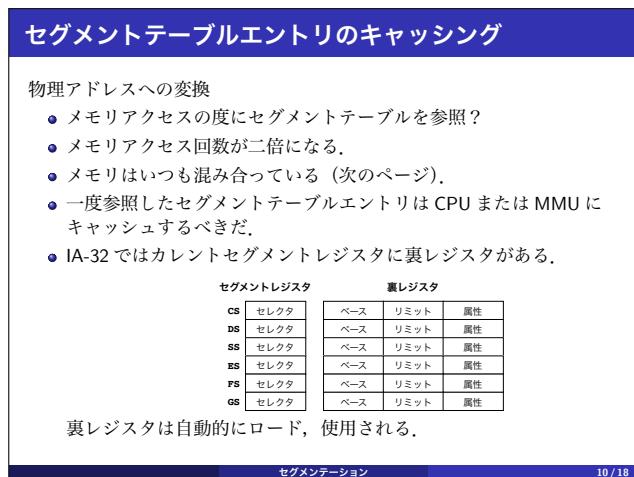
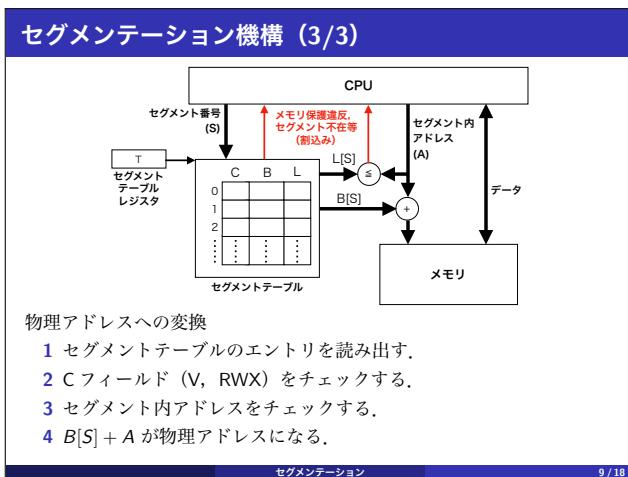
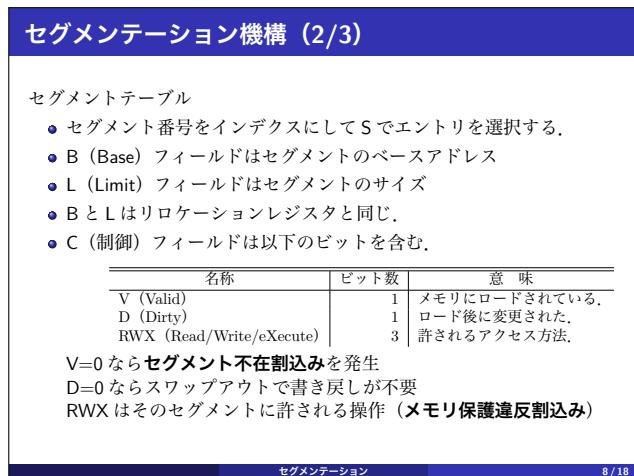
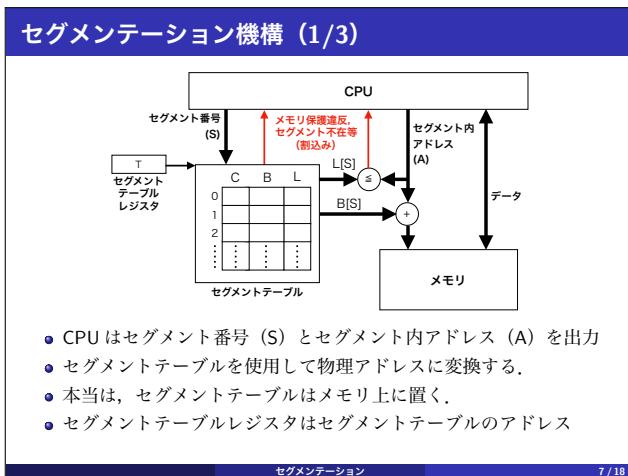
OP A	→	OP S A
従来の機械語		
セグメント番号付の機械語		

プログラムが大きくなる。

- カレントセグメント

CALLS, RETS 命令でセグメントが自動的に切り換わる。
IA-32 ではカレントセグメントが複数ある。
(CS:プログラム, DS:データ, SS:スタックセグメント等)

セグメンテーション 6 / 18



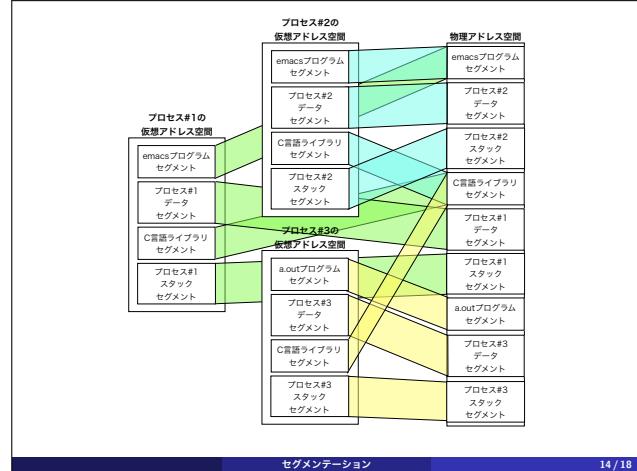
セグメントの共用

プロセス間でセグメントを共用しメモリを節約する。

- プログラムや定数等は書き変わらない。(共有可)
 - プログラム本体
 - サブルーチン(ライブラリ)
 - 定数表
- データ、ヒープ、スタック等は書き換わる。(共有不可)
プロセス毎に別のコピーを持つ必要がある。

セグメンテーション

13 / 18



セグメンテーション

14 / 18

セグメンテーションの利点・欠点 (1/2)

利点

- セグメントには、例えば「C言語ライブラリセグメント」のような、論理的な意味を持たせることができる。
- セグメントの論理的な意味を反映したメモリ保護が可能である。
- プログラムやデータの共用が容易である。
- セグメントの長さは自由に決められるので内部フラグメントが発生しない。
- セグメントの長さは動的に変化させることも可能である。
- セグメント単位のスワッピングを用いて仮想記憶を実現できる。

セグメンテーション

15 / 18

セグメンテーションの利点・欠点 (2/2)

欠点

- 物理アドレス空間に外部フラグメントが生じる。
- 外部フラグメントの解消にはメモリコンパクションが必要である。
- 物理メモリ上に連続した領域が必要である。
- 物理メモリより大きいセグメントを作ることができない。

これらの欠点を克服するために、ページングを組み合わせたシステムがある。(IA-32, MULTICS)

セグメンテーション

16 / 18

練習問題1

- セグメントテーブルが次のような状態の時、以下の間に答えなさい。なお、物理アドレスは8ビットとする。

	C	B	L
0	V=1	0x30	0x20
1	V=1	0x80	0x30
2	V=1	0x00	0x20
3	V=0	0x50	0x20
...

- 1 次の仮想アドレスに対応する物理アドレスを答えなさい。但し、物理アドレスに変換できない場合はエラーと答えなさい。

- (1) 0x0:0x10
- (2) 0x1:0x10
- (3) 0x1:0x40
- (4) 0x2:0x10
- (5) 0x2:0x20
- (6) 0x3:0x10

- 2 セグメントの配置を記入した物理アドレス空間のメモリマップを作成しなさい。

セグメンテーション

17 / 18

練習問題2

- セグメンテーション機構の図で、セグメントテーブルに適当な値を設定し幾つかの二次元アドレスが変換させる物理アドレスを確かめなさい。
- あるセグメントサイズが大きくなる場合の、セグメントテーブルの変更項目等を指摘しなさい。
- メモリコンパクションの手順を説明しなさい。
- スタックセグメントを意識した前向きに伸びるセグメントも利用可能なセグメンテーション機構を設計しなさい。
 - 1 セグメントテーブルに必要な変更は?
 - 2 機構に必要な変更は?
 - 3 他に必要な変更は?

セグメンテーション

18 / 18

オペレーティングシステム 第11章 ページング

<https://github.com/tctsigemura/OSTextBook>

ページング

1 / 25

ページング

ページングは以下のようなメモリ管理方式である。

- メモリを一様なページに分割し、ページ単位で管理する。
- メモリより広い仮想アドレス空間を使用できる。
- 外部フラグメンテーションを生じない。
- メモリコンパクションが不要である。
- Windows, macOS, Linux 等、現代の多くのOSが採用している。
- 用語
 - ページ：仮想アドレス空間をページに分割する。
 - フレーム：物理アドレス空間をフレームに分割する。

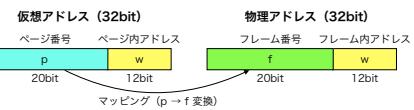
2 / 25

基本概念

ページング

3 / 25

ページとフレーム



- 仮想アドレスの上位ビットがページ番号 (p)
- 仮想アドレスの下位ビットがページ内アドレス (w)
- ページサイズは 2 の累乗にする。
- ページ内アドレスが w ビットならページサイズは 2^w バイト
- 物理アドレスの上位ビットがフレーム番号 (f)
- 物理アドレスの下位ビットがフレーム内アドレス (w)
- ページ内アドレスとフレーム内アドレスは同じ (w)
- p を f に変換することでページをフレームにマッピングする。
- p と f のビット数は異なっていても良い。

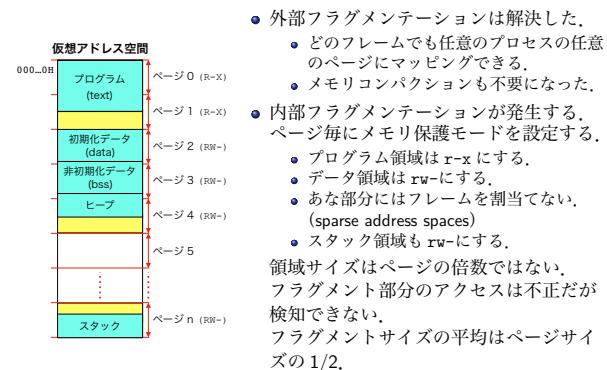
4 / 25

マッピング関数

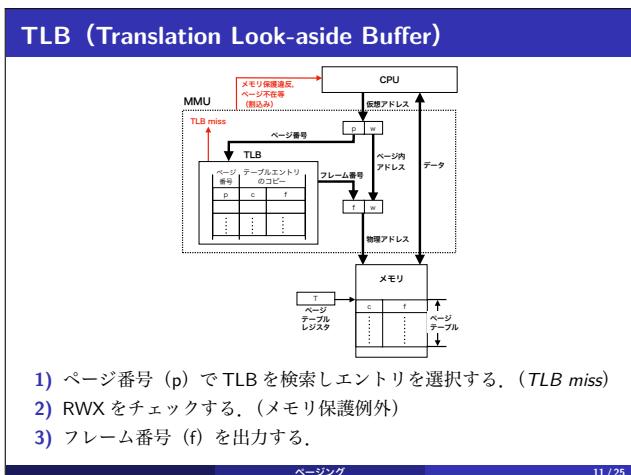
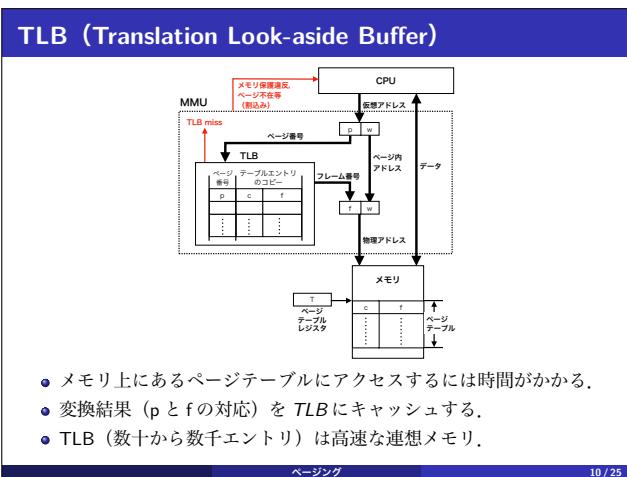
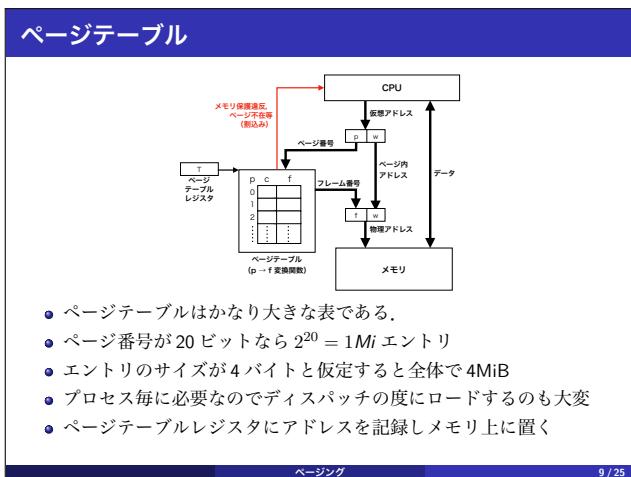
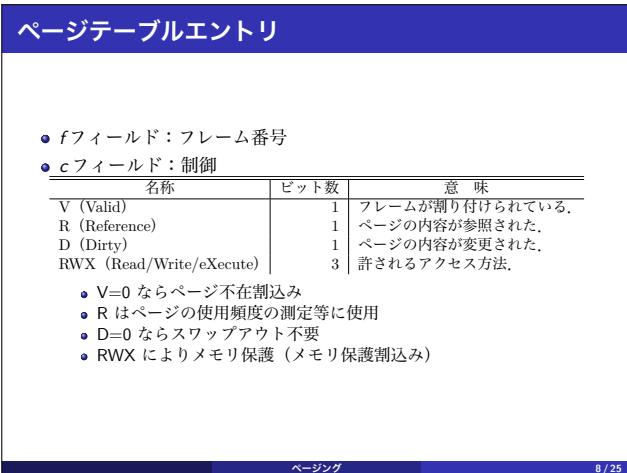
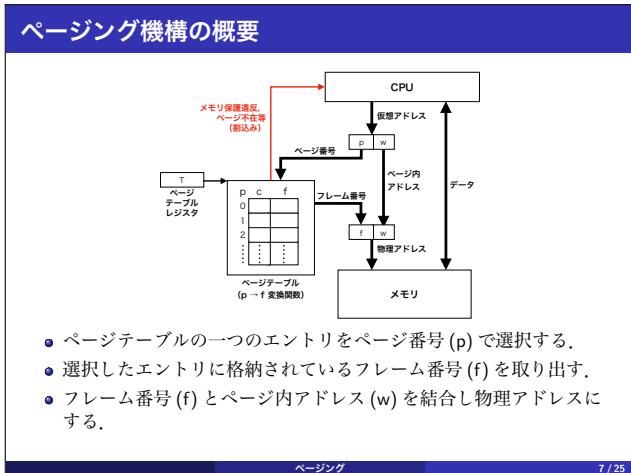
マッピング関数

5 / 25

フラグメンテーション



6 / 25



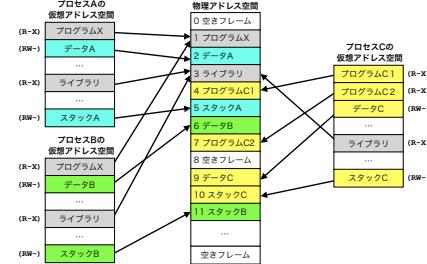
TLBエントリのクリア

- プロセススイッチのとき
- ページテーブルに変更があったとき
- TLB の内容は変更前のページテーブルのエントリなのでクリアする必要がある。
- TLB のクリアは大きなペナルティを伴うので避けたい。
- TLB のエントリがプロセス番号を含む方式
- TLB のエントリを個別にクリアできる方式

ページング

13 / 25

フレームの共用



- プロセスが変更しないページ (R-X) は共用できる。
- ページテーブルの操作により実現
- ライブリリースは位置独立コードでなければならぬ。

ページング

14 / 25

位置独立コード

位置独立コードのイメージ

```

CALL 200,PC      // 200番地先にあるサブルーチン実行
JMP 100,PC       // 100番地先にジャンプする
LD G0,4,FP        // ローカル変数はスタック上
ST G0,40,G11     // グローバル変数はレジスタ相対で参照

```

- ライブリリースはマッピングされる仮想アドレスが変化する。
- どのアドレスにマッピングされても大丈夫なプログラム => 位置独立コード
- PC 相対で JMP や CALL を使う。
- データはレジスタをベースにアクセスする。

ページング

15 / 25

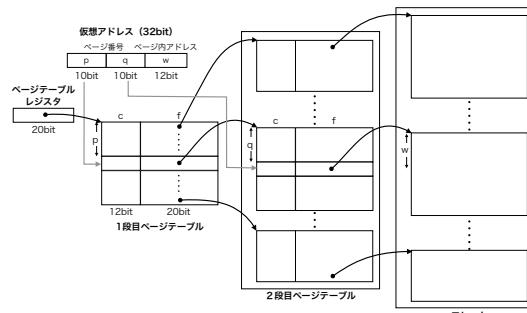
ページテーブルの編成方法

- ページテーブルは大きくなりすぎる。 (32ビットCPUの例)
 - 仮想アドレス32ビット、ページサイズ4KiB、エントリ4Bの例
 $2^{32} \div 4\text{KiB} = 2^{32} \div 2^{12} = 2^{20} = 1Mi$ エントリ
 $1Mi$ エントリ $\times 4B = 4MiB$
 - 32ビットCPUの普及が始まった当時のPCは、メモリを4MiB～16MiBしか搭載していなかった。
- ページテーブルは大きくなりすぎる。 (64ビットCPUの例)
 - 仮想アドレス48ビット、ページサイズ4KiB、エントリ8Bの例
 $2^{48} \div 4\text{KiB} = 2^{48} \div 2^{12} = 2^{36} = 64Gi$ エントリ
 $64Gi$ エントリ $\times 8B = 512GiB$
 - 現代の64ビットPCのメモリは、4GiB～16GiB程度？
- ページテーブルを小さくする工夫が必要！！

ページング

16 / 25

二段のページテーブル (IA-32の例)

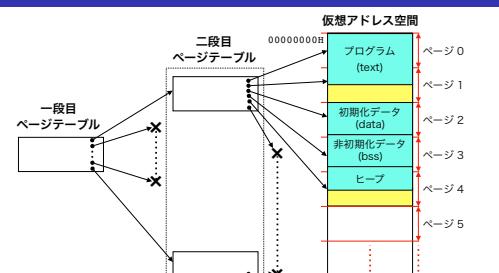


- 一段目のページテーブルサイズ $4\text{KiB} = \text{フレームサイズ}$
- 二段目のページテーブルの区画サイズ $4\text{KiB} = \text{フレームサイズ}$

ページング

17 / 25

ページテーブルフレームの節約



- 仮想アドレス空間の必要な部分の二段目を省略
- 一段目のページテーブルエントリの V=0 にする。
- 従来 1,025 フレーム => 3 フレーム

ページング

18 / 25

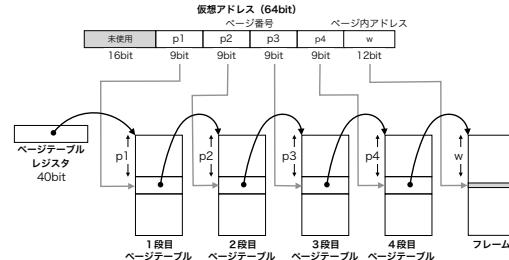
64ビット仮想アドレス空間 (x86-64の例)

- 実質48ビット仮想アドレス => 256TiB (十分大きい)
- 仮に二段のページテーブルならページテーブルの区画は18ビット (p), 18ビット (q), 12ビット (w) と仮定
エントリサイズ = 8B と仮定
 $2^{18} \times 8B = 2MiB$
- プロセスあたり最低でも3区画必要
 $2MiB \times 3 = 6MiB$
- 400個のプロセスがあったとすると
 $6MiB \times 400 = 2.4GiB$ (8GiBの30%)
- 二段のページテーブルでは区画が大きくなりすぎる。
- 区画を小さくするために段数を多くする。

ページング

19 / 25

四段のページテーブル (x86-64の例)

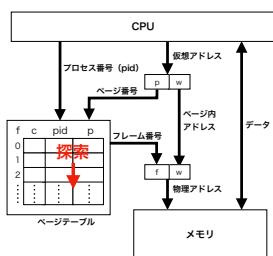


- ページサイズ (フレームサイズ) は 4KiB
- ページテーブルの区画は $2^9 \times 8B = 4KiB$
- ページテーブルは最低7フレーム (28KiB)
- 400プロセスでも約11MiBで済む (8GiBの0.13%)

ページング

20 / 25

逆引きページテーブル



- テーブルでフレーム番号とページ番号の立場が逆転 (逆引き)
- システム全体でページテーブル一つ (プロセス毎ではない)
- どの仮想アドレス空間のエントリか識別するための pid あり

ページング

21 / 25

逆引きページテーブル

ページテーブルのサイズ

- 8GiBのメモリを4KiBのページで分割する場合のエントリ数
 $8GiB \div 4KiB = 2^{33} \div 2^{12} = 2Mi$ エントリ
- 1エントリ8バイト仮定すると
 $2Mi \times 8B = 12MiB$
- システム全体で12MiBで済む (8GiBの0.2%)

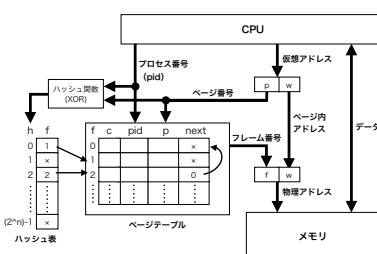
Page Table Walk

- CPUは仮想アドレスの他に pid (プロセス番号)を出力
- ページテーブルを pid と p (ページ番号)で探索する
- 線形探索などを用いると遅くて実用にならない

ページング

22 / 25

逆引きページテーブル (IBM 801 の例)



- ハッシュ表を用いて探索を高速化
- ハッシュ表の大きさは二の累乗 ($0 \sim 2^n - 1$)
- ページテーブルは next を使用してチェインを作る

ページング

23 / 25

逆引きページテーブル (IBM 801 の例)

ハッシュ表を用いた Page Table Walk

- pid と p を用いてハッシュ関数を計算する ($h \leftarrow f(pid, p)$)
(ハッシュ関数は pid と p の XOR。 \dots 。速度優先)
- ハッシュ表の第 h エントリを見る
 - 空 (図では x) ならページ不在 (削込み !)
 - 空でなければページテーブルのインデクス (f)
- ページテーブルの第 f エントリの内容を見る
 - pid, p が一致 → この時の f をフレーム番号にする (完了 !)
 - pid, p が一致しない → next を見る
 - 空 (図では x) ならページ不在 (削込み !)
 - 空でなければ
 - ページテーブルのインデクス (f) を更新して再度トライ

TLB: 不可欠!

ページング

24 / 25

練習問題

- (1) 一回のメモリアクセスに 5ns, page table walk に 20nsかかるとする。TLB のヒット率が 90 パーセントの時の平均メモリアクセス時間を計算しなさい。
- (2) 図 11.7において, $p = 1$ の仮想アドレスの範囲を 8 桁の 16 進数で答えなさい。
- (3) 図 11.7において, $p = 1, q = 1$ の仮想アドレスの範囲を 8 桁の 16 進数で答えなさい。
- (4) 逆引きページテーブルを用いる場合, TLB に格納すべき最低限の情報の範囲を考察しなさい。
- (5) 図 11.11 に, $pid = 3, p = 2$ のページがフレーム 1 にマッピングされるようなページテーブルの状態を書き込みなさい。
- (6) 逆引きページテーブルを用いるシステムで, プロセス間でページの共有が可能か考察しなさい。

オペレーティングシステム 第12章 仮想記憶

<https://github.com/tctsigemura/OSTextBook>

仮想記憶

1 / 36

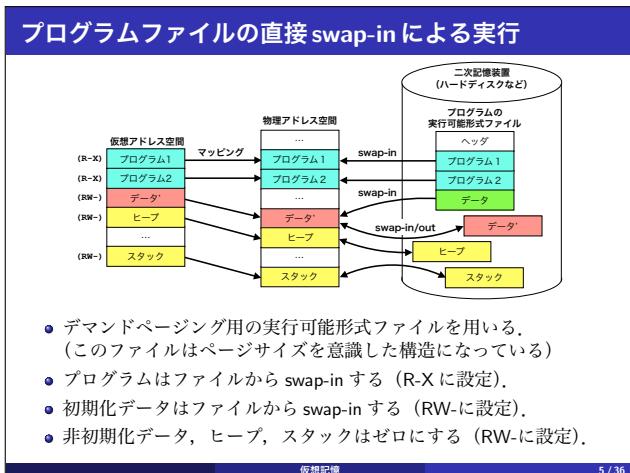
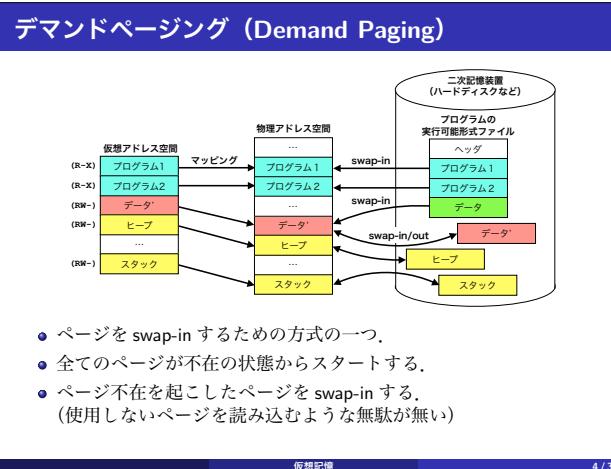
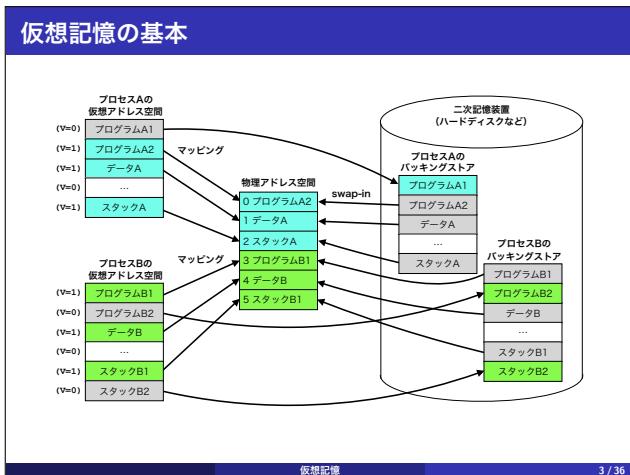
基本概念

ページングをベースに仮想記憶を実現する。

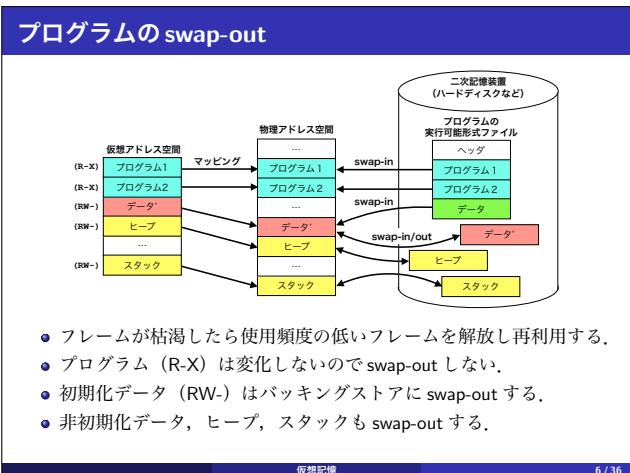
- システムの使用メモリ合計が物理メモリより大きい。→ 実行不可
- 単一のプログラムがメモリより大きい。→ 実行可
- ページテーブルの V=0 を上手く使用する。
- V=0 のページにアクセスするとページ不在割込み → OS へ
- ページテーブルの V=0 に二つの場合がある。
 - 無効な領域 → プロセス終了
 - バッキングストアに退避中 → 復旧して再開
- プロセス生成時にバッキングストアにプロセスのイメージを作る。
- Windows, macOS, Linux 等、現代の OS のほとんどが採用している。

仮想記憶

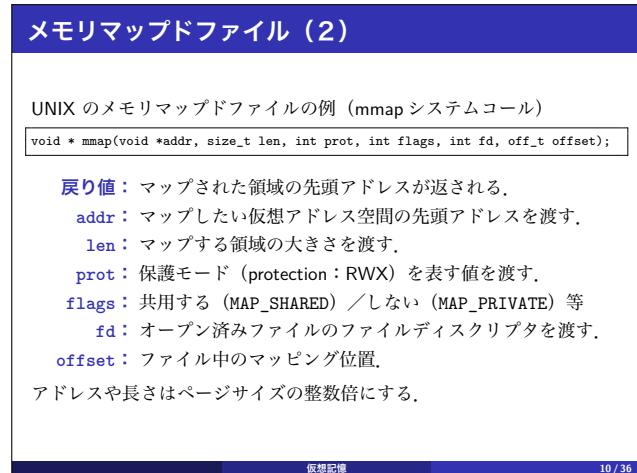
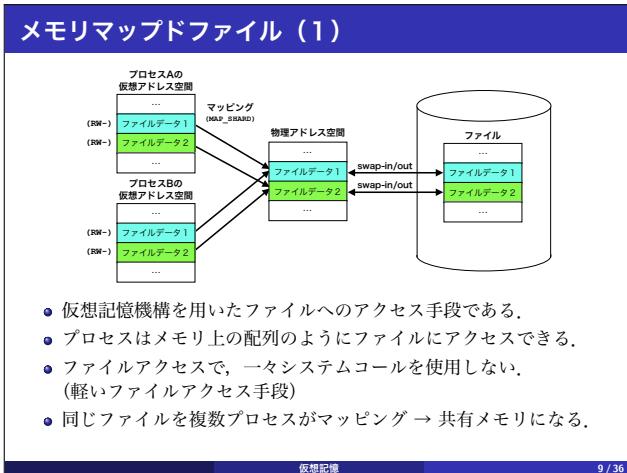
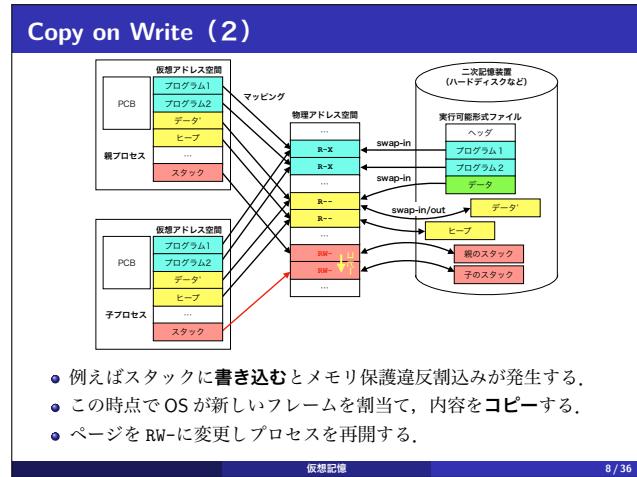
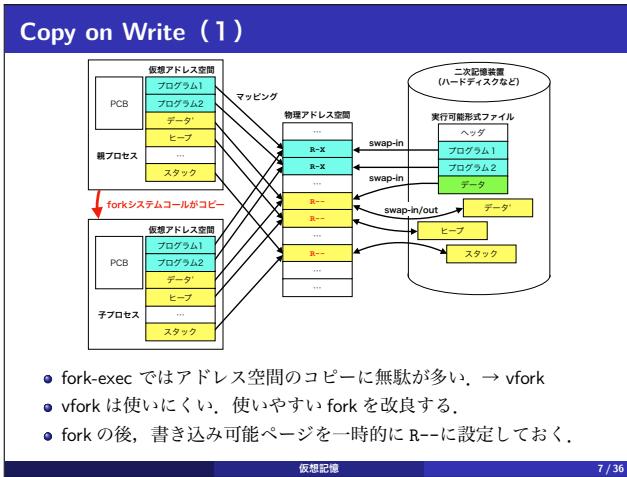
2 / 36



- デマンドページング用の実行可能形式ファイルを用いる。
(このファイルはページサイズを意識した構造になっている)
- プログラムはファイルから swap-in する (R-X に設定)。
- 初期化データはファイルから swap-in する (RW-に設定)。
- 非初期化データ、ヒープ、スタックはゼロにする (RW-に設定)。



- フレームが枯渇したら使用頻度の低いフレームを解放し再利用する。
- プログラム (R-X) は変化ないので swap-out しない。
- 初期化データ (RW-) はバッキングストアに swap-out する。
- 非初期化データ、ヒープ、スタックも swap-out する。

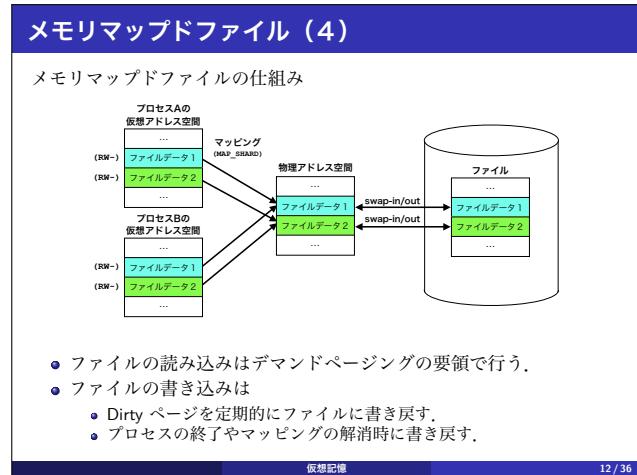


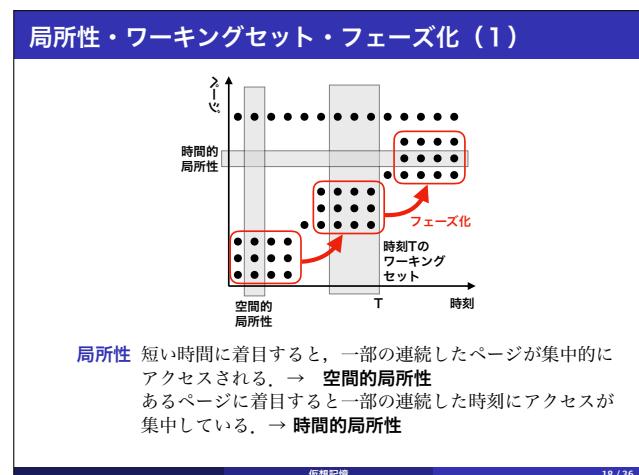
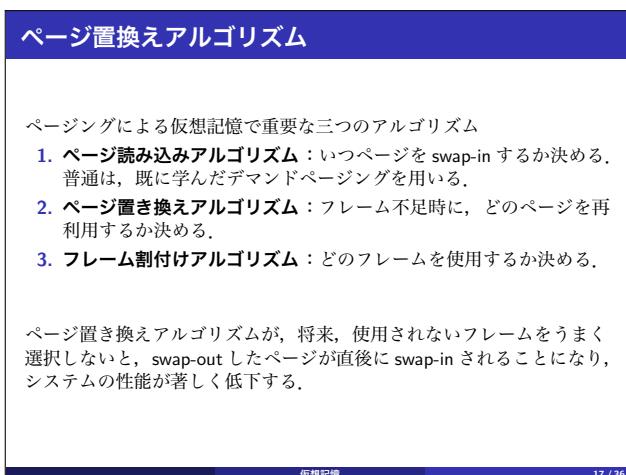
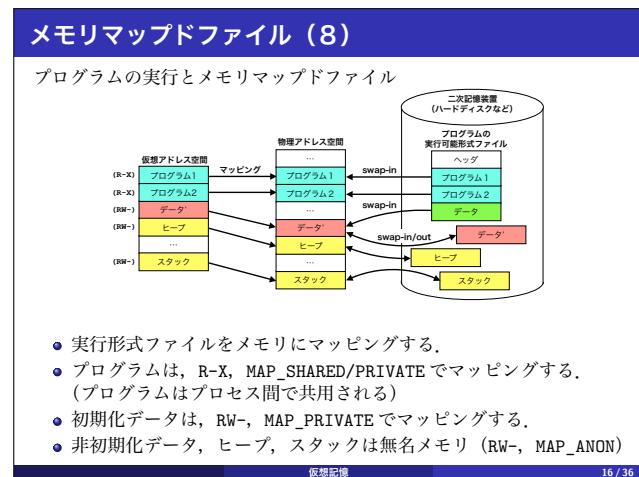
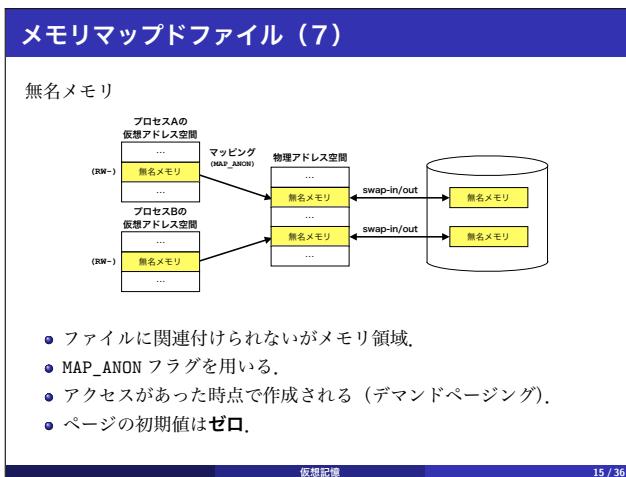
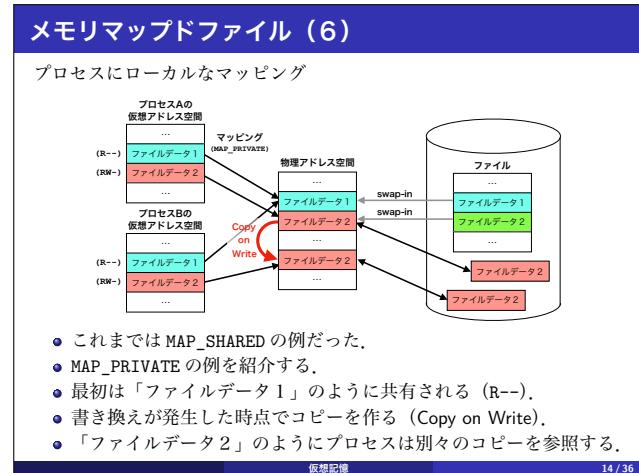
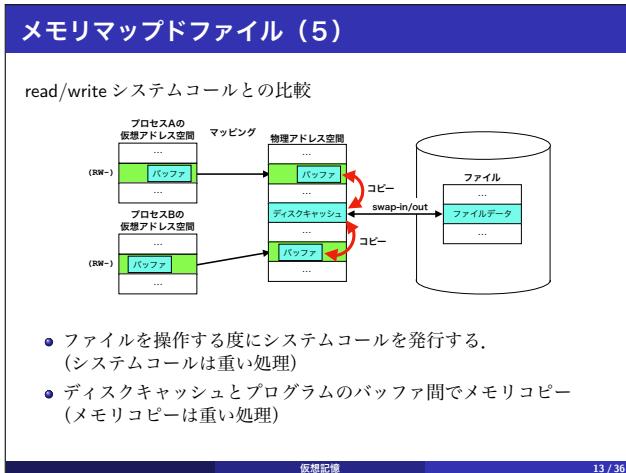
メモリマップドファイル (3)

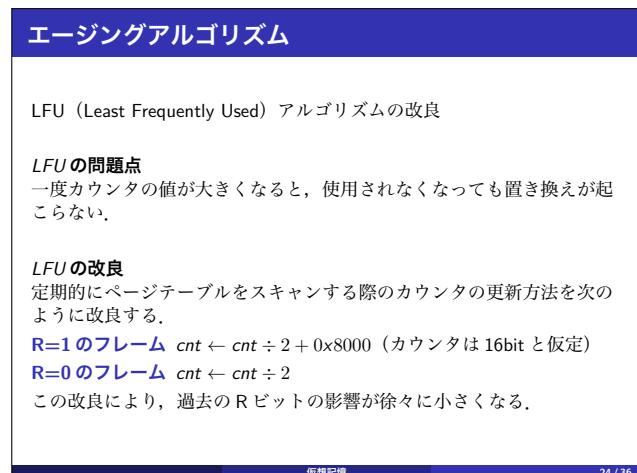
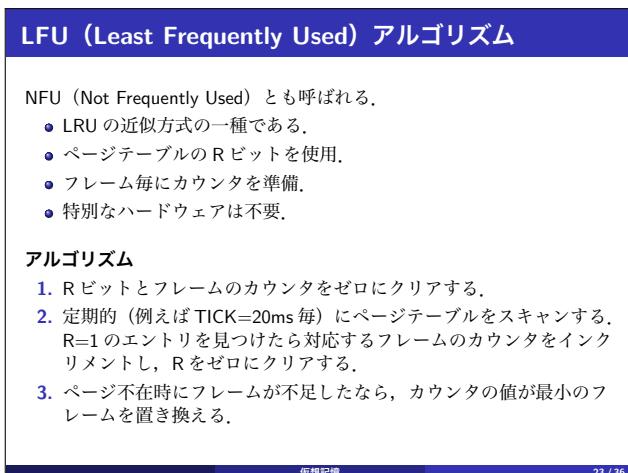
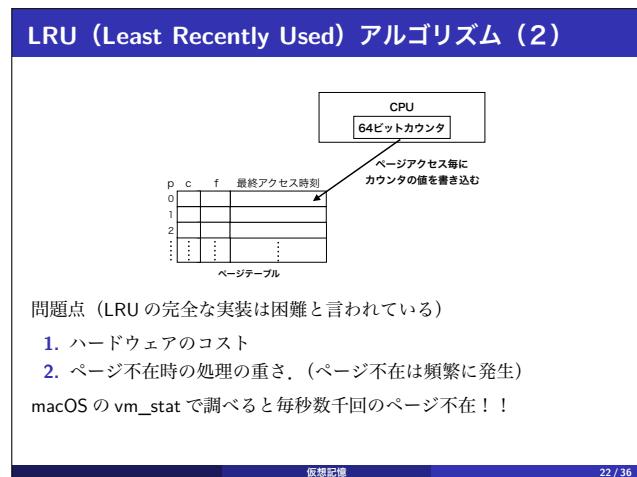
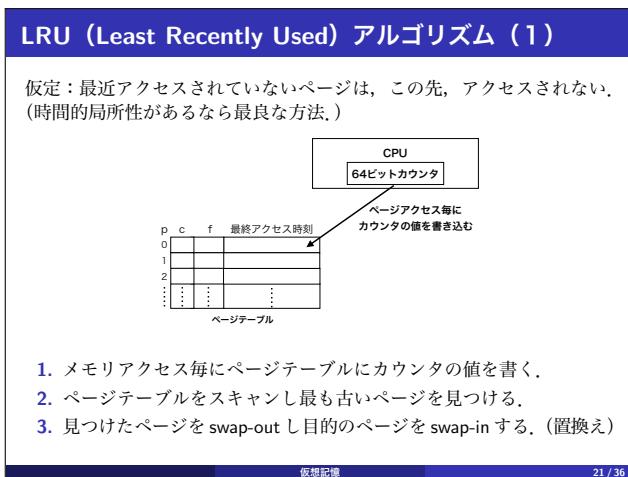
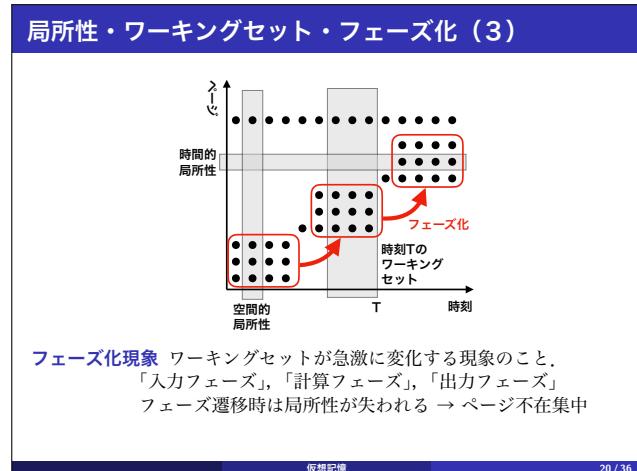
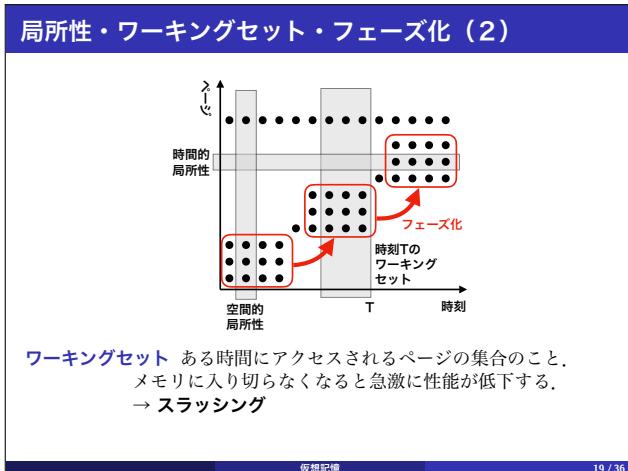
```

1 #include <stdio.h>           // perror のために必要
2 #include <fcntl.h>          // open のために必要
3 #include <unistd.h>          // close のために必要
4 #include <sys/mman.h>        // mmap のために必要
5 int main() {
6     int fd;
7     char *p, *fname="a.txt";
8     fd = open(fname, O_RDWR);    // 予め作成してある 4KiB のファイルを開く
9     if (fd<0) {
10         perror(fname);
11         return 1;
12     }
13     p = mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_FILE|MAP_SHARED, fd, 0);
14     if (p==MAP_FAILED) {
15         perror("mmap");
16         return 1;
17     }
18     close(fd);                // マップしたらクローズして良い
19     for (int i=0; i<4096; i++) { // ファイルに A~Z を繰り返し書き込む
20         p[i] = 'A' + (i % 26);
21     }
22     return 0;
23 }
```

仮想記憶 11 / 36

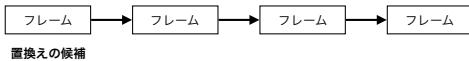






FIFO (First-In First-Out) アルゴリズム

仮定：「長くメモリに滞在しているページは役割を終えている」
特別なハードウェアを用いることなく、ソフトウェアだけで実現できる。
古いフレーム 新しいフレーム



アルゴリズム

1. swap-in したフレームをリストの最後に追加する。
2. フレームが不足時は、リストの先頭のフレームを置き換える。

ページテーブルのスキャンが不要なので非常に軽い。
常に使用されるページも時間が経過すると swap-out される問題がある。
*Belady の異常な振る舞い*をすることがある。

仮想記憶

25 / 36

Belady の異常な振る舞いの例

FIFO アルゴリズムを用い、
ページ参照ストリング (W : 1 2 3 4 1 2 5 1 2 3 4 5) の場合

- フレーム数 (m=3) の場合 (ページ不在 9 回)

	1	2	3	4	1	2	5	1	2	3	4	5
W	*1	*2	*3	*4	*1	*2	*5	5	5	*3	*4	*5
S	1	2	3	4	1	2	2	2	5	3	3	
	1	2	3	4	1	1	1	2	5	5	5	

- フレーム数 (m=4) の場合 (ページ不在 10 回)

	1	2	3	4	1	2	5	1	2	3	4	5
W	*1	*2	*3	*4	4	4	*5	*1	*2	*3	*4	*5
S	1	2	3	3	3	4	5	1	2	3	4	
	1	2	2	3	4	5	1	2	3	4	5	
	1	1	1	2	3	4	5	1	2	3	4	

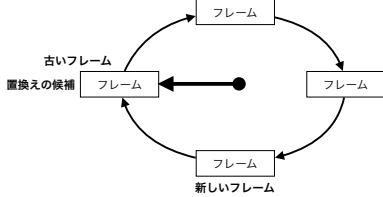
メモリが多い方 (m=4) のページ不在回数が多い。

仮想記憶

26 / 36

Clock アルゴリズム

環状リストを用いる。R ビットも使用する。

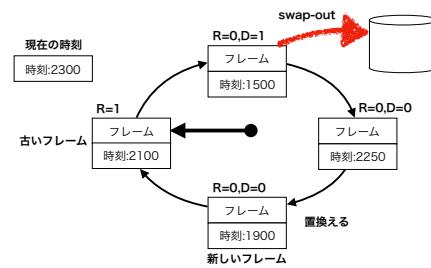


1. swap-in する度にフレームを環状リストに挿入していく。
2. 定期的 (例えれば TICK=20ms 毎) に R ビットをクリアする。
3. 時計の針が指しているフレームの R ビットを調べる。
R=0 の場合 ページは古い+最近アクセスされていない。→ 置換
R=1 の場合 ページは最近アクセスされている。→ 針を進める
最悪でも時計の針が一周回ると R=0 のページが見つかる。

仮想記憶

27 / 36

WSClock アルゴリズム (1)

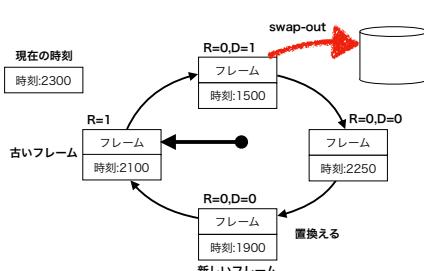


- ワーキングセットを考慮した Clock アルゴリズムである。
- 単純でパフォーマンスが良いので広く使用されている。
- アクセス時刻を記録した環状リストに用いる。

仮想記憶

28 / 36

WSClock アルゴリズム (2)

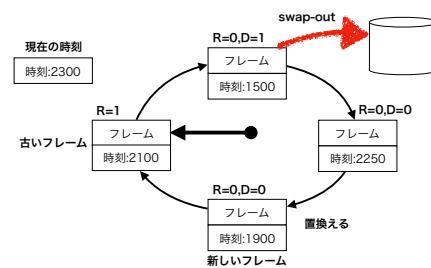


- 時刻が古くなっているフレームはワーキングセット外と判断。
- ページテーブルの R ビットと D ビットも使用する。

仮想記憶

29 / 36

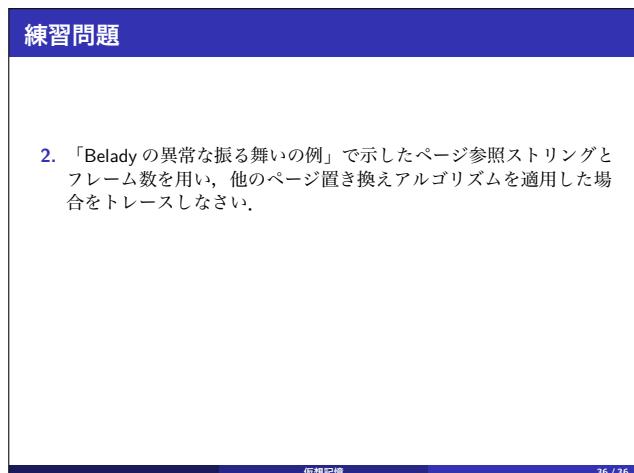
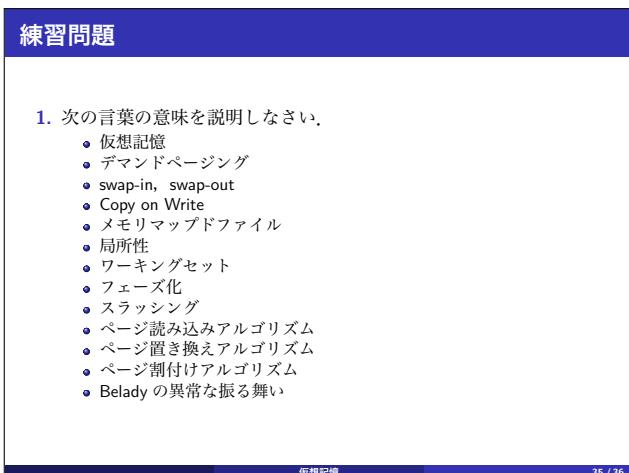
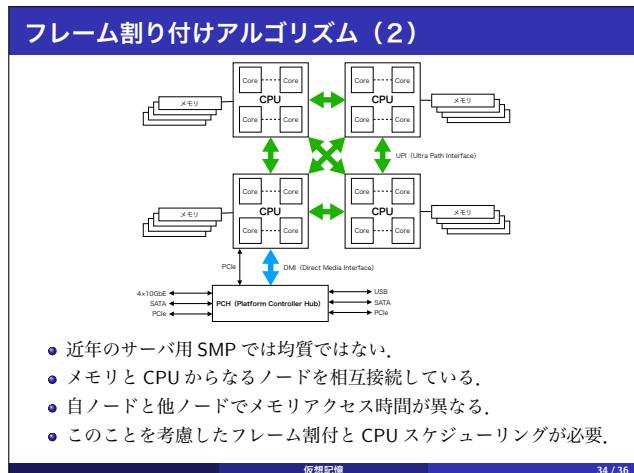
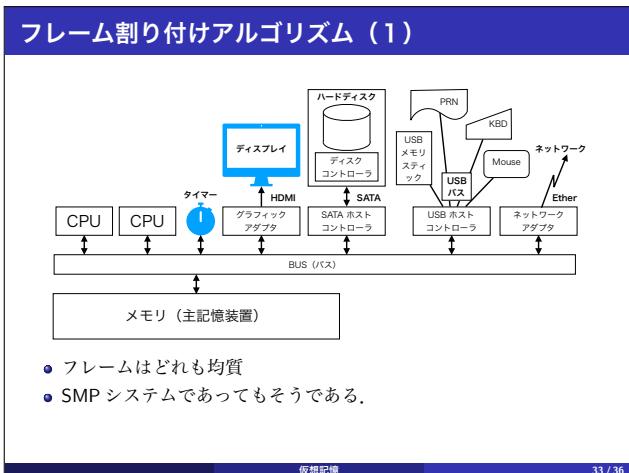
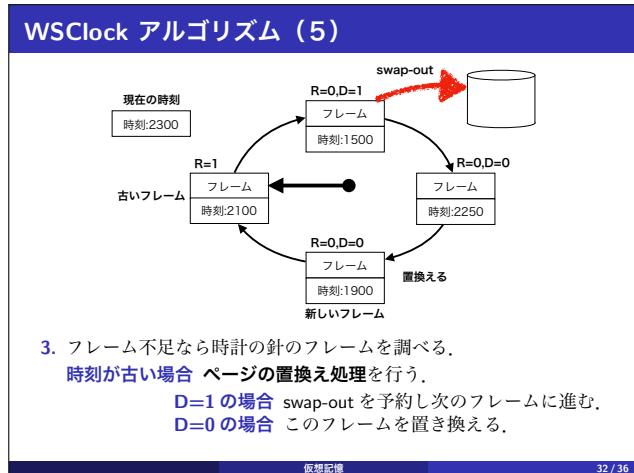
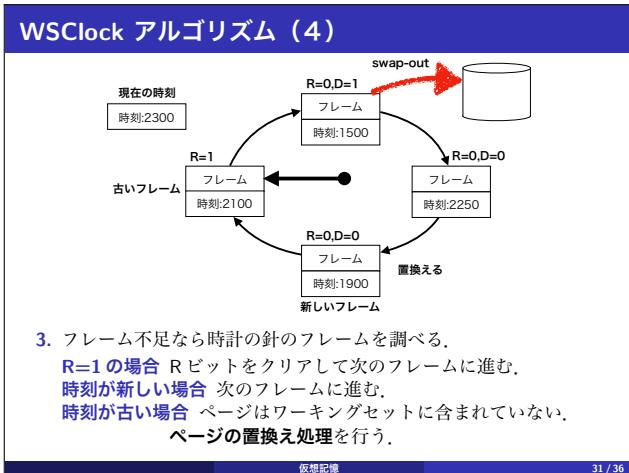
WSClock アルゴリズム (3)



1. swap-in する度にフレームを環状リストに挿入していく。
2. 定期的に全テーブルエントリの R ビットをクリアする。
その際、R=1 だったフレームだけに現在時刻を記録する。

仮想記憶

30 / 36



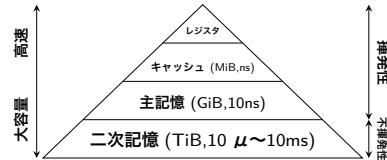
オペレーティングシステム 第13章 二次記憶装置（ストレージ）

<https://github.com/tctsigemura/OSTextBook>

二次記憶装置

1 / 19

記憶装置の階層（1）



- レジスタは CPU レジスタのこと。
容量は数十～数百バイト程度、高速アクセスが可能、揮発性
- 主記憶（メモリ）
アクセス時間は数十ナノ秒程度
容量は数 Gi バイト～数十 Gi バイト程度、揮発性
- 二次記憶装置
ハードディスクや SSD (Solid State Drive) のこと。
アクセス時間は数ミリ秒～数十ミリ秒（ハードディスク）、不揮発性

二次記憶装置

2 / 19

記憶装置の階層（2）

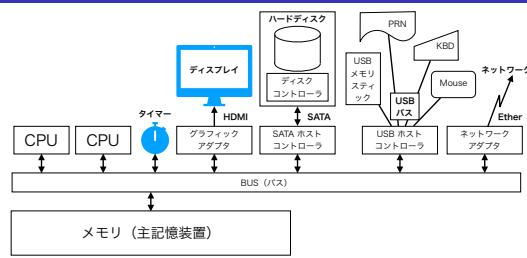
夫々の特性に合った使い方をする。
二次記憶装置の特性は次の通り。

- 大容量（ビット単価が安い）
オペレーティングシステム、アプリケーション、データなどの全てを格納できる。
- 不揮発性（電源を切っても消えない）
プログラムやデータの永続的な置き場所として適している。

二次記憶装置

3 / 19

二次記憶装置の種類（1）



接続方式

- CPU からはホストコントローラを介してアクセスする。
- 二次記憶装置はホストコントローラの先に接続される。
- USB メモリスティックやポータブルハードディスクは取り外し可能。
- 取り外し可能 => データ交換、バックアップ用途にも適する。

二次記憶装置

4 / 19

二次記憶装置の種類（2）



テープ型装置

- データのバックアップや輸送用（ビット単価が安い）
- シーケンシャルアクセス専用
- 読み出し位置まで進むために数分！！

二次記憶装置

5 / 19

二次記憶装置の種類（3）



ディスク型装置

- ランダムアクセスが可能
- ハードディスクのこと（CD-ROMなどの光ディスクも仲間）
- SSD、USB メモリ、その他メモリカードも仲間

二次記憶装置

6 / 19

ハードディスク (1)

ハードディスク

- データは磁気的に円盤の表面に記録され不揮発性。
- アドレスを指定してランダムアクセスが可能。
- セクタ単位で読み書きを行う。
- システムの起動ドライブ (OS, アプリ, データ全てが置かれる)
- 仮想記憶のバックストレージとしても使用される。
- ハードディスク管理が, OSの性能や使い勝手を左右する。
- ファイル管理機構はハードディスクを前提にしていることが多い

セクタ・トラック・シリンドラ

- 同心円のトラック (Track)
- トラックを区切ったセクタ (Sector)
- トラックをまとめたシリンドラ (Cylinder)

二次記憶装置

7 / 19

ハードディスク (2)



二次記憶装置

8 / 19

ハードディスク (3)

セクタのアドレッシング

512 バイト (4KiB) のセクタのアドレス付け方法

- CHS (Cylinder Head Sector) 方式**
 - Cylinder Head Sector の三次元アドレス。
 - Head は Track と同じ意味。
 - CHS は PC の世界で使用されてきた用語。
 - ハードディスクの物理的な構造通りのアドレッシング。
 - 過去、長く使われてきた方式。
- LBA (Logical Block Addressing)**
 - セクタの通し番号 (一次元) を用いる。
 - ハードディスクブラックボックス化 (物理構造が不明)
 - CHS は煩雑なだけにメリットがなくなった。

二次記憶装置

9 / 19

フォーマッティング (1)

ハードディスクの初期化の例

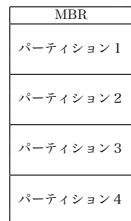
- 低レベル (物理) フォーマット**
ディスクの表面に磁気的にトラックを書き込む。
- パーティション (区画) に分割**
 - 装置全体を一つのボリューム => 大きすぎる
 - 区画に分割し区画をボリュームとして扱う => オペレーティングシステムのパーティション
 - ユーザデータのパーティション => ここだけバックアップ
 - 複数のオペレーティングシステムをインストール
第1パーティション (ボリューム) に Windows
第2パーティション (ボリューム) に Linux
第3パーティション (ボリューム) に FreeBSD
- 高レベル (論理) フォーマット**
各ボリュームの内部に該当オペレーティングシステムの空のファイルシステムを作る。

二次記憶装置

10 / 19

フォーマッティング (2)

PC用ハードディスクのパーティションの例



● MBR (Master Boot Record)

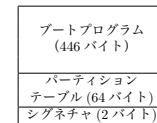
- ハードディスクの先頭セクタ (LBA0) に格納
- MBR のサイズは 512 バイト
- 内容はブートプログラムとパーティショントーブル

二次記憶装置

11 / 19

フォーマッティング (3)

PC用ハードディスクの MBR の内容



● MBR (Master Boot Record) (512 バイト)

- ブートプログラム (446 バイト)
PC の機械語プログラム (OS を起動するためのプログラム)
- パーティショントーブル (64 バイト)
各パーティションの位置と大きさ等を記録する 4 行の表
- シグネチャ (2 バイト)
フォーマッティングされている目印 (55H, AAH)

二次記憶装置

12 / 19

フォーマッティング (4)

PC用ハードディスクのパーティションテーブルの例

Flag (1)	Start CHS(3)	Type (1)	End CHS(3)	Start LBA(4)	Size (4)
80H	???	06H	???	00000003FH	00003FOOH
00H	???	A5H	???	00003F3FH	0000BD00H
00H	???	00H	???	?????????	???????
00H	???	00H	???	?????????	???????

項目	バイト数	意味	Type	意味
Flag	1	80H アクティブ/ 00H インアクティブ	00H	空き
Start CHS	3	開始アドレス (CHS 表現)	01H	FAT12
Type	1	ファイルシステムの種類	04H	FAT16(小)
End CHS	3	終了アドレス (CHS 表現)	06H	FAT16(大)
Start LBA	4	開始アドレス (LBA 表現)	07H	NTFS
Size	4	セクタ数 (LBA 表現)	08H	FAT32
			83H	Linux(ext2)
			A5H	FreeBSD

二次記憶装置

13 / 19

ブートストラップ (1)

PCの場合を例にブートストラップを説明する。

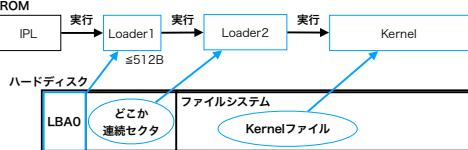
- ハードディスクからOSを起動する作業のこと。
- OSのカーネルを格納したファイルを見つけてロード・実行する。
- PCの製造時にはどんなOSがインストールされるか分からず。
=> ブートストラップは後で変更できる必要がある。
- 以下に説明する段階を経てOSをブートする。
- 以下の方法がPCでは標準的であるが様々な変種がある。
(段階が多い場合、強力なブートマネージャを備えている場合)

二次記憶装置

14 / 19

ブートストラップ (2)

ハードディスク = ボリュームの場合



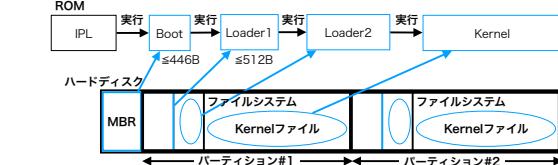
- **IPL (Initial Program Loader)**
PCのROMに格納されており電源ONと同時に動作開始
- ブートローダ (第1段階: Loader1) 512バイト以内
LBA0に格納されたIPLによってロード・実行される。
- ブートローダ (第2段階: Loader2)
ディスク上のどこか連続セクタに格納され Loader1 がロード・実行。
サイズに制限がない => 高機能にできる。
- OSのカーネル
ファイルシステムにファイルとして格納され Loader2 がロード・実行。

二次記憶装置

15 / 19

ブートストラップ (3)

パーティション = ボリュームの場合



- **IPL (Initial Program Loader)**
- ブートセレクタ・ブートマネージャ (Boot) 446バイト以内
LBA0 (MBR) に格納されIPLによってロード・実行される。
メニューを表示してユーザーにOSのパーティションを選択させる。
(勝手に次に進むものもある。)
- ブートローダ (第1段階: Loader1) 512バイト以内
- ブートローダ (第2段階: Loader2)
- OSのカーネル

二次記憶装置

16 / 19

練習問題 (1)

1. 次の言葉の意味を説明しなさい。

- 二次記憶装置
- 撃発性・不撃発性
- 記憶の階層
- テープ型装置・ディスク型装置
- シーケンシャルアクセス・ランダムアクセス
- セクタ・トラック・シリンドラ
- CHS・LBA
- ボリューム
- パーティション
- MBR
- IPL
- ブートストラップ

二次記憶装置

17 / 19

練習問題 (2)

2. 次のディスクに付いて答えなさい。

1台全体 1,024シリンドラ
1シリンドラ 8トラック
1トラック 128セクタ
1セクタ 512バイト

- ディスクの容量をセクタ単位で答えなさい。
- ディスクの容量をバイト単位で答えなさい。
- 最後のセクタのアドレスをLBAで答えなさい。
- 最後のセクタのアドレスをCHSで答えなさい。
(但し, C:0以上, H:0以上, S:1以上である。)

二次記憶装置

18 / 19

練習問題（3）

3. 例示したパーティションテーブルに付いて答えなさい。
 - 第1パーティションの位置を LBA で答えなさい。
 - 第1パーティションのサイズをセクタ数で答えなさい。
 - 第1パーティションの種類を答えなさい。
 - 第2パーティションの位置を LBA で答えなさい。
 - 第2パーティションのサイズをセクタ数で答えなさい。
 - 第2パーティションの種類を答えなさい。
4. PC 用の高機能なブートローダ GRUB について調査しなさい。

オペレーティングシステム 第14章 ファイルシステムの概念

<https://github.com/tctsigemura/OSTextBook>

ファイルシステムの概念

1 / 20

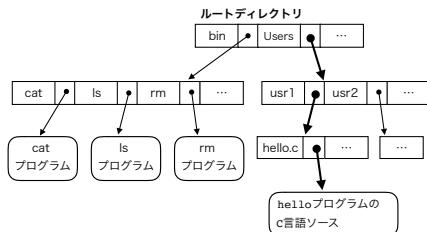
ファイルシステム

- ファイルシステムは二次記憶装置を
 - 管理する。(どのセクタが、どのファイルの一部?)
 - 抽象化する。(ハードディスク → ファイル)
 - 仮想化する。(1台のハードディスク → 多数のファイル)
- ファイルは一次元のバイト列 (バイトストリーム)
 - オペレーティングシステムはファイルの構造を決めない。
- ファイルは名前を持つ。
 - 名前とバイト位置でデータが決まる。
名前 = ファイル名、バイト位置 = ファイル内オフセット

ファイルシステムの概念

2 / 20

ファイルの名前付け



- ファイルは木構造のディレクトリシステムに格納する。
- ディレクトリは名前とファイル本体のポインタを格納する。
- 階層構造を持った名前(パス)でファイルを特定する。
- 絶対パスはルートディレクトリを起点にする。
- 相対パスはワーキングディレクトリを起点にする。

ファイルシステムの概念

3 / 20

ファイルの別名 (1)

別名があると便利な例 (最新のファイルはいつも同じ名前)

ある日

2017_06_30.log	2017年6月30日のファイル
2017_07_01.log	2017年7月1日のファイル
2017_07_02.log	2017年7月2日のファイル
today.log	→ 2017_07_02.log

次の日

2017_07_01.log	2017年7月1日のファイル
2017_07_02.log	2017年7月2日のファイル
2017_07_03.log	2017年7月3日のファイル
today.log	→ 2017_07_03.log

ファイルシステムの概念

4 / 20

ファイルの別名 (2)

- ハードリンク
 - ファイルシステムの仕組みとしてOSカーネルに組み込む。
 - ファイル本体が複数のディレクトリ・エントリから指される。
 - リンクカウントを用いる。
 - ディレクトリをリンクするとループ検出が厄介 → 禁止！
- シンボリックリンク
 - ファイルシステムの仕組みとしてOSカーネルに組み込む。
 - 他ファイルのパスを格納した特別なファイル。
 - リンク切れ状態が許される。(Webページのリンクに似ている)
- ファイルシステムの外で実装されるリンク
 - Windowsのショートカット、macOSのエイリアスなど
 - ファイルシステム本体が持つリンク機構は一定ではない。
→ 現代のOSは同時に複数のファイルシステムを使用する。
→ アプリに近い側でどのファイルシステムでも共通の仕組みを提供

ファイルシステムの概念

5 / 20

ファイルの別名 (3)

HFS+ファイルシステム上のmacOSのエイリアスの例

```

1 $ ls -l@ a.txt*
2 -rw-r--r-- 1 sigemura admin 5 Jun 27 10:19 a.txt
3 -rw-r--r--@ 1 sigemura admin 1012 Jun 27 10:19 a.txt のエイリアス
4 com.apple.FinderInfo 32

```

3行 拡張属性付きの通常ファイルとしてエイリアスが存在

4行 拡張属性の名前はcom.apple.FinderInfo

4行 拡張属性のサイズは32バイト

ファイルシステムのより汎用的な機構である拡張属性を利用して、エイリアスを実装している。

ファイルシステムの概念

6 / 20

ファイルの別名 (4)

FATファイルシステム上のmacOSのエイリアスの例

```
$ ls -la@ ./* a.txt*
-rwxrwxrwx 1 sigemura staff 4096 Jun 27 09:55 a.txt のエイリアス
-rw-rw-rw- 1 sigemura staff 5 Jun 27 09:55 a.txt
-rw-rw-rw@ 1 sigemura staff 1040 Jun 27 09:55 a.txt のエイリアス
com.apple.FinderInfo 32
$ rm ..a.txt*
-rwxrwxrwx 1 sigemura staff 5 Jun 27 09:55 a.txt
-rw-rw-rw@ 1 sigemura staff 1040 Jun 27 09:55 a.txt のエイリアス
```

4.5行 拡張属性付きの通常ファイルとしてエイリアスが存在
 2行 隠しファイルができている！！
 6行 隠しファイルを消してみる。
 9行 拡張属性が消えてしまった！！

FATファイルシステムの規約の範囲でエイリアスを実装している。

ボリュームのマウント

(a) マウント方式
 (b) ドライブレター方式

- 二つ目以降のボリュームの接続方法
- マウント方式
 - ボリュームを既存のディレクトリに接続する。
 - /Volumes/NO NAME/hello.c が USB メモリの C プログラム
- ドライブレター方式
 - ボリュームを区別するドライブレターを用いる。
 - D:\hello.c が USB メモリの C プログラム

ファイルの属性 (1)

- 名前**: ファイル名をファイルの属性と考える場合もある。
- 識別子**: ファイル本体の番号など。
- 型 (タイプ)**: 通常ファイル、ディレクトリ、リンクなど。
- 保護**: `rwxrwxrwx` など。(後で詳しく)
- 日時**: 作成日時、最終変更日時など。
- 所有者**: 所有者、グループなど。
- 位置**: ディスク上のどこにファイル本体があるか。(データを格納したブロック(セクタ)の番号など)
- サイズ**: ファイルのバイト数。
- 拡張属性**: 名前付きの小さな追加データ。ファイルシステムで用途を定めていない。

ファイルの属性 (2)

```
$ ls -l@ b.txt*
-rw-r--r--  2 sigemura staff 123 Jun 25 19:38 b.txt
-rw-r--r--@ 1 sigemura staff 836 Jun 25 19:39 b.txt のエイリアス
com.apple.FinderInfo 32
$ xattr -l b.txt のエイリアス
com.apple.FinderInfo:
00000000 61 6c 69 73 4d 41 43 53 80 00 00 00 00 00 00 00 |alisMACS.....|
00000010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 l.....|
```

1行 拡張属性付きでファイル一覧を表示。
 4行 拡張属性付きのファイルがある。
 5行 拡張属性の内容を表示してみる。

この例の拡張属性は、以下のようなものであった。

- 属性の名前: com.apple.FinderInfo
- 属性の大きさ: 32 バイト
- 意味: ファイルがエイリアスである。(ファイル本体がエイリアスのデータ)

アクセス制御 (1)

ファイルの保護属性に基づき、ファイルに誰が何ができるか制御する。

- ビット表現の保護モード**
 - UNIXで使用される `rwxrwxrwx` のような情報。
 - UNIXの場合、「所有者、グループ、その他」のユーザについて
`r`: 読める(Read),
`w`: 書ける(Write),
`x`: 実行できる(Execute)
 を指定する。

アクセス制御 (2)

- ACL (Access Control List)**
 ファイル毎に、ユーザやグループを指定して細かな制御が可能

```
$ ls -l a.txt
-rw-r--r-- 1 sigemura staff 4 Jul 5 21:55 a.txt
$ chmod +a "group:admin allow write" a.txt
$ chmod +a "group:admin deny delete" a.txt
$ ls -l a.txt
-rw-r--r--@ 1 sigemura staff 4 Jul 5 21:55 a.txt
0: group:admin deny delete
1: group:admin allow write
```

1行 a.txt に ACL が無いことを確認した。
 3,4行 chmod コマンドで a.txt に ACL 追加した。
 7,8行 二行の ACL が確認できる。

- リストの先頭から順に評価する。
- 許可・不許可が決まったら評価を完了する。
- ACL で決まらない場合は `rwx` を使用する。

ファイルの種類

- ファイルシステム (OS カーネル) で決まっている種類
(通常ファイル・ディレクトリ・リンクなど)
- アプリケーションなどが決めている種類
(通常ファイルの拡張子で区別する)

拡張子	意味
.c, .java, .s 等	ソース・プログラム (C 言語, Java 言語, アセンブリ言語)
.py, .pl, .php 等	スクリプト言語のプログラム (python, perl, PHP)
.txt, .html, .xml 等	プレーンテキスト, マークアップ言語
.jpg, .png, .bmp 等	画像データ
.mp3, .m4a, .wma 等	音声データ
.mpg, .mp4, .wmv 等	動画データ
.pdf, .ps, .eps 等	印刷・表示用の文書ファイル
.zip, .tar, .tbz 等	アーカイブファイル
.exe, .app, 拡張子無し	実行形式プログラム (Windows, macOS, UNIX)
.doc, .docx	MS Word 文書
.app	だけはディレクトリの拡張子

ファイルシステムの概念

13 / 20

ファイルシステムの操作 (1)

ディレクトリ操作

機能	対応する UNIX の API
ファイルの作成	creat, open(... O_CREAT ...) システムコール
ディレクトリの作成	mkdir システムコール
ファイルの削除	unlink システムコール
ディレクトリの削除	rmdir システムコール
リンクの作成	link, symlink システムコール
リンクの削除	unlink システムコール
名前の変更 (移動)	rename システムコール
ディレクトリエントリの読み出し	opendir, readdir, closedir 関数

- ファイルの作成は creat システムコールでもできる。
- ディレクトリの読み出しが opendir システムコールで行う。
- rename システムコールはファイルの移動もできる。

ファイルシステムの概念

14 / 20

ファイルシステムの操作 (2)

ファイル操作

機能	対応する UNIX の API
ファイルを開く	open システムコール
データを読む	read システムコール
データを書く	write システムコール
読み書き位置を移動	lseek システムコール
ファイルを閉じる	close システムコール
ファイルの切り詰め	truncate, open(... O_TRUNC) システムコール
ファイルのプログラムを実行	execve システムコール
ファイルの属性変更	chmod, chown, chgrp, utimes システムコール
ファイル属性の読み出し	stat システムコール

- open はファイルの保護属性をチェックする。
- 切り詰めは専用の truncate システムコールも使える。
- ファイルの属性の読み書きができるべき。

ファイルシステムの概念

15 / 20

ファイルシステムの操作 (3)

ファイルの共有とロック

```
#include <sys/file.h>
#define LOCK_SH 1 // 共有ロック
#define LOCK_EX 2 // 排他ロック
#define LOCK_NB 4 // ブロックしない
#define LOCK_UN 8 // ロック解除
int flock(int fd, int operation);
```

- LOCK_SH : 共有ロック (*shared lock*)
- LOCK_EX : 排他ロック (*exclusive lock*)
- LOCK_NB : ロックできない時、ブロックしないでエラー
- open システムコールにもロックの機能がある。

ワーキングディレクトリの変更

```
#include <unistd.h>
int chdir(const char *path);
```

ファイルシステムの概念

16 / 20

ファイルシステムの健全性 (1)

一貫性チェック

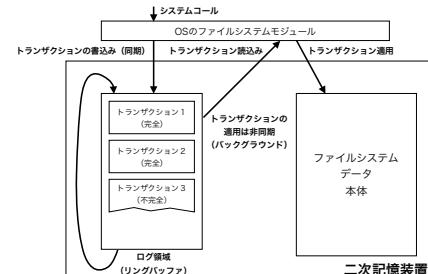
- 正常終了時にはファイルシステムにアンマウントの印をする。
- OS の起動時に印がなかったら一貫性チェックをする。
- メタデータの矛盾を解消するだけ。
- ファイルが消えたり、データが消えたりは修復できない。

ファイルシステムの概念

17 / 20

ファイルシステムの健全性 (2)

ジャーナリング・ファイルシステム



- データベースの WAL (Write Ahead Logging) のアイデア。
- NTFS, ext3, ext4, HFS+ 等が該当する。

ファイルシステムの概念

18 / 20

練習問題（1）

1. 次の言葉の意味を説明しなさい。
 - ディレクトリシステム
 - パス、絶対パス、相対パス
 - ディレクトリ、ファイル
 - ハードリンク、シンボリックリンク
 - ショートカット、エイリアス
 - マウント、ドライブレター
 - 拡張属性、ACL
2. 自分のオペレーティングシステムについて調査しなさい。
(GUIより CLI のコマンドを用い方がより詳しい観察ができる。)
 - ショートカット (Windows), エイリアス (macOS)
 - ファイルの属性 (保護, 日時, 所有者, サイズ等)
 - 拡張属性が使用できるオペレーティングシステムか?
 - ACL が使用できるオペレーティングシステムか?
 - USB メモリにはどのようなパスで到達できるか?
 - ファイルシステムの一貫性をチェックするコマンドは何か?

ファイルシステムの概念

19 / 20

練習問題（2）

3. 自分が使用しているオペレーティングシステムで試してみなさい。
 - ショートカットやエイリアスを作成し試してみなさい。

```
# macOS の場合の実行例
$ echo aaa > a.txt
$ open a.txt
$ open a.txt のエイリアス      <--- エイリアスは GUI で作る
$ cat a.txt
$ cat a.txt のエイリアス
```

- UNIX や macOS で実行して結果が異なる理由を考察しなさい。

# ハードリンクの場合	# シンボリックリンクの場合
\$ echo aaa > a.txt	\$ echo aaa > a.txt
\$ echo bbb > b.txt	\$ echo bbb > b.txt
\$ ln a.txt c.txt	\$ ln -s a.txt c.txt
\$ mv a.txt d.txt	\$ mv a.txt d.txt
\$ mv b.txt a.txt	\$ mv b.txt a.txt
\$ cat c.txt	\$ cat c.txt

- ショートカットやエイリアスの振る舞いを調べる。
(リンク先ファイルを削除・移動・別ファイルに置換した場合など)
- ACL の追加・削除とその効果を確認する。

ファイルシステムの概念

20 / 20

オペレーティングシステム 第15章 FAT ファイルシステム

<https://github.com/tctsigemura/OSTextBook>

FAT ファイルシステム 1 / 9

FAT ファイルシステム

- 1980年代からPCのファイルシステムとして利用してきた。
- 仕様が公開されているので様々なところで利用されている。
 - USBメモリ、SDカード、その他メモリカード
 - Windows、Linux、macOSはFATファイルシステムをサポート
 - 同じUSBメモリを使用してデータ交換が可能
 - デジカメ、音楽プレーヤー、デジタルテレビ、カーナビ...
 - デジカメのSDカードをPCで読める。
- ファイル名は半角8文字+3文字(例:IMG_1234.JPG)
- FATにはいくつかの種類がある。

種類	最大ボリュームサイズ	最大ファイルサイズ	ファイル名
FAT12	32MiB	32MiB	8+3文字
FAT16	2GiB	2GiB	8+3文字
FAT32	2TiB	4GiB	8+3文字
exFAT	16EiB	16EiB	255文字

FAT ファイルシステム 2 / 9

ボリューム内部の構造

(ディスク装置全体、または、パーティション) = ボリューム

- パーティションの位置と大きさはMBRのテーブルで決まった。
- ボリュームの内部構造はFATファイルシステムのルールで決まる。
- ボリューム内部のコンポーネントの大きさなどはBPBから分かる。
- FATは大切なデータなので2重化している。
- データ領域はクラスタと呼ばれるブロック単位で扱う。

FAT ファイルシステム 3 / 9

BPB (BIOS Parameter Block)

パラメータ	意味	位置	長さ	値の例
ジャンプ命令	ジャンプ機械語命令	0	3	0xeb 0x3 0x90
セクタサイズ	1セクタのバイト数	11	2	512バイト
クラスタサイズ	1クラスタのセクタ数	13	1	64セクタ
予約セクタ数	予約セクタ数(BPBを含む)	14	2	1セクタ
FAT数	FATを何重に記録するか	16	1	2個
rootDir サイズ	ディレクトリエントリ数	17	2	512エントリ
総セクタ数16	ボリュームのサイズ	19	2	0
FAT サイズ	FATのセクタ数	22	2	245セクタ
総セクタ数32	ボリュームのサイズ	32	4	3,999,681セクタ
ボリュームラベル	ボリュームの名前	43	11	"MICRODRIVE"
ブートプログラム	ブートプログラム	62	448	
シグネチャ	フォーマット済みマーク	510	2	0x55 0xaa (位置と長さの単位はバイト)

(値の例はボリュームサイズ2GiB、クラスタサイズ32KiB、FAT16の場合)
(「総セクタ数16」で表現できない場合は「総セクタ数32」を使用する)

- 論理フォーマット時に決めたパラメータを記録している。
- 値の例は、ボリューム=2GiB、クラスタ=32KiB、FAT16
- ブートプログラムは初代IBM PCの機械語で作成してある。

FAT ファイルシステム 4 / 9

ディレクトリエントリ

Bytes	8	3	1	10	2	2	2	4
FileName	Ext	Atr	Reserved	Time	Date	Cls	Size	
0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00	

- ルートディレクトリやディレクトリファイルに格納される。
- 前のBPBだとルートディレクトリに512個格納される。
- FileName: 8文字以内のファイル名
(0x00:以降未使用, 0x05:削除, 0x05:本当は0x05)
- Ext: 3文字以内の拡張子
- Atr: ファイルの属性
(0x01:read-only, 0x02:hidden, 0x10:directory...)
- Time: ファイルの最終変更時刻
- Date: ファイルの最終変更日
- Cls: データが格納されている先頭クラスタの番号
- Size: ファイルの大きさ(バイト単位)

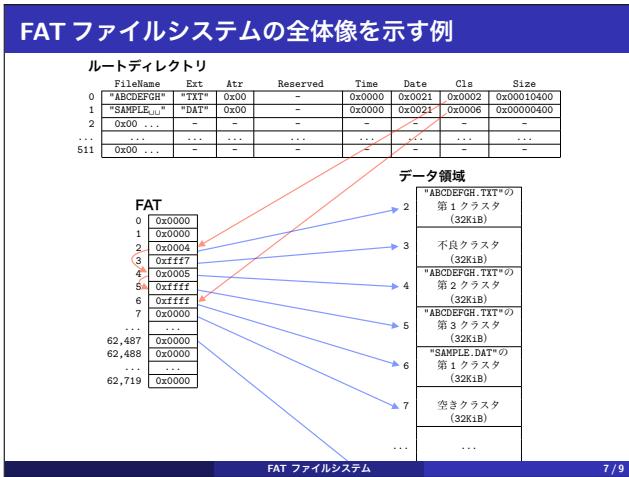
FAT ファイルシステム 5 / 9

FAT (File Allocation Table)

値	意味
0x0000	使用不可
0x0000	使用不可
0x0004	次は第4クラスタ
0xffff7	不良クラスタ
0x0005	次は第5クラスタ
0xffff	終了クラスタ
0x0000	未使用クラスタ
...	

- クラスタをファイルに割り当てる表
- FATエントリとクラスタが一対一対応
- FAT16ではFATエントリが16ビット
- 0x0002～0xffffが普通のクラスタ番号
- クラスタチェインを形成する。
- ディレクトリエントリのClsがチェインの先頭を指す。

FAT ファイルシステム 6 / 9



ディレクトリファイル

```
$ cd /Volumes/NO\ NAME
$ mkdir A
$ mkdir A/DIR
$ echo AAA > A/A.TXT
$ hexdump -C A
00000000 2e 20 20 20 20 20 20 20 20 20 30 00 aa 7d 98  |.          .|. 
00000010 e4 4c e4 4c 00 00 a9 98 e4 4c 21 00 00 00 00 00 |.L.L.....L!....| 
00000020 2e 2c 20 20 20 20 20 20 20 20 20 10 00 aa 7d 98 |..          ..|. 
00000030 e4 4c e4 4c 00 00 7d 98 e4 4c 00 00 00 00 00 00 |.L.L..J..L.....| 
00000040 44 49 52 20 20 20 20 20 20 20 20 10 00 31 a2 98 |DIR      .1.| 
00000050 e4 4c e4 4c 00 00 a2 98 e4 4c 31 00 00 00 00 00 |.L.L.....L1.....| 
00000060 41 20 20 20 20 20 20 20 54 58 54 20 00 13 a9 98 |A       TXT    .|. 
00000070 e4 4c e4 4c 00 00 a9 98 e4 4c 40 00 04 00 00 00 |.L.L.....L0.....| 
00000080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....| 
* 
000008000
```

- ルートディレクトリ以外のディレクトリは Atr=0x10 のファイル
- 上の実行例は macOS で FAT16 ファイルシステム上で実験したもの
- 最初の 2 エントリーは「.」と「..」を格納している。

FAT ファイルシステム

8 / 9

- 練習問題
- 次の言葉の意味を説明しなさい。
 - BPB
 - ルートディレクトリ
 - クラスタ
 - ディレクトリエンタリ
 - FAT
 - クラスタチェイン
 - ディレクトリファイル
 - ディレクトリファイルのダンプリストをディレクトリエンタリの構造と比較しながら解析しなさい。
 - 全体像を示す例の ABCDEFGH.TXT ファイルの第 0x00002000 バイトが格納されるクラスタの番号を答えなさい。
 - 前問と同様に、第 0x00004000, 0x00008000, 0x00010000 バイトに付いて答えなさい。
- FAT ファイルシステム
- 9 / 9

オペレーティングシステム 第16章 UNIX ファイルシステム

<https://github.com/tctsigemura/OSTextBook>

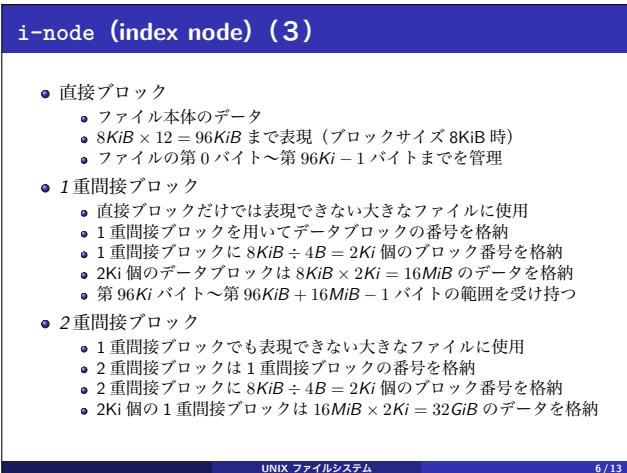
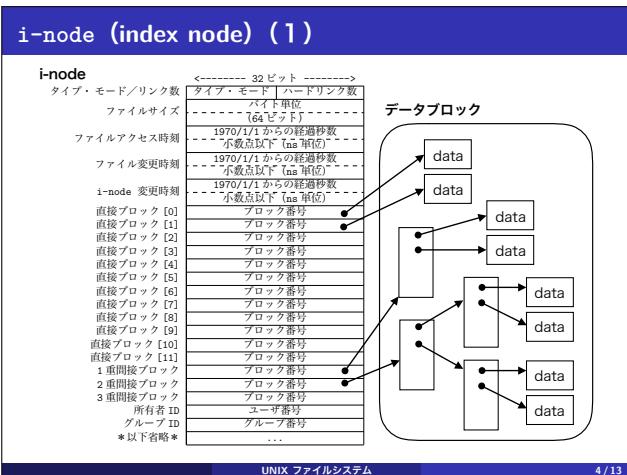
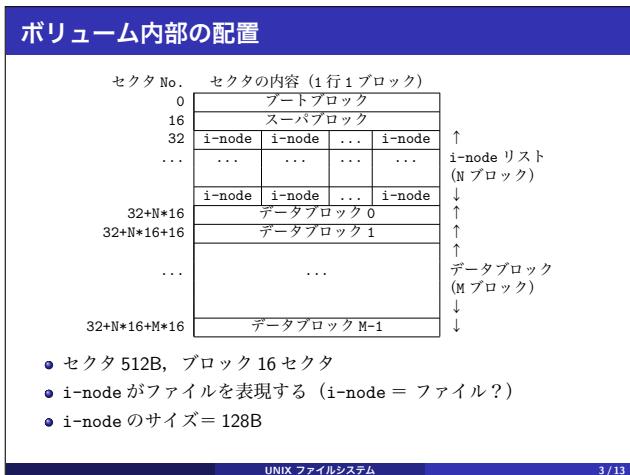
UNIX ファイルシステム 1 / 13

UNIX ファイルシステム

- UFS (UNIX File System)
- 1979年にリリースされたVersion 7 UNIXのファイルシステムとそれを改良した多くのファイルシステムをUFSと呼ぶ。
- 第14章で「UNIXの場合」=「UFSの場合」だった。
 - 木構造のディレクトリシステム
 - ハードリンク、シンボリックリンク
 - ボリュームのマウント
 - ファイルの属性(所有者:マルチユーザを意識)
 - ファイル操作のシステムコール、...
- 多くのファイルシステムがUFSに準拠している。
 - WindowsのNTFS
 - macOSのHFS+, APFS
 - Linuxのext3, ext4

UNIX ファイルシステム

2 / 13



i-node (index node) (4)

- 3重間接ブロック
 - 2重間接ブロックを2Ki個管理できる
 - 2Ki個の2重間接ブロックは $32GiB \times 2Ki = 64TiB$ のデータを格納
- 所有者ID: ファイル所有者のユーザ番号 (マルチユーザ)
- グループID: ファイルのグループ番号

インデクス方式 i-nodeのような構造を使う方式のこと。
高速なランダムアクセスができる。

スペースファイル 途中に穴の空いたファイルのこと。
インデクス方式はスペースファイルを表現できる。

UNIX ファイルシステム 7 / 13

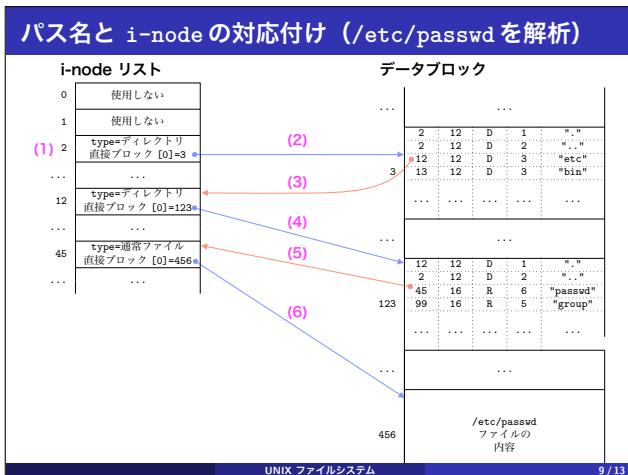
ディレクトリファイル

- ファイルの一種である。→ i-nodeにより表現
- ファイルの型 (type) がディレクトリを表す値のもの
- ディレクトリファイルはファイル名と i-node の対応表
- 対応表の1行がディレクトリエントリ

32bit	16bit	8bit	8bit	l ₂ バイト	詰め物
i-node番号	l ₁	型	l ₂	ファイル名	\0 ... \0
				l ₁ バイト	----->

- i-node番号: ファイルの i-node 番号
- l₁: ディレクトリエントリの大きさ (4の倍数バイト)
- 型: ファイルの型 (抹消されたディレクトリエントリの表現)
- l₂: ファイル名のバイト数
- ファイル名: 255文字以内
- 詰め物: エントリが4の倍数バイトになるように

UNIX ファイルシステム 8 / 13



- ### パス名と i-node の対応付け (/etc/passwd を解析)
- 絶対パスなのでルートディレクトリから探索を開始する。
ルートディレクトリの i-node 番号は必ず 2 と決められている。
 - ルートディレクトリの i-node から、データブロック 3 にルートディレクトリの内容が格納されていることが分かる。
 - データブロック 3 の 3 番目のエントリから、etc が 12 番の i-node に対応する事が分かる。
 - 12 番目の i-node から etc はディレクトリファイルであること、内容がデータブロック 123 に格納されていることが分かる。
 - データブロック 123 の 3 番目のエントリから、passwd が 45 番の i-node に対応する事が分かる。
 - 45 番目の i-node から passwd は普通のファイルであること、ファイルの内容がデータブロック 456 に格納されていることが分かる。
- UNIX ファイルシステム 10 / 13

練習問題 (1)

- 次の言葉の意味を説明しなさい。
 - ブートブロック
 - スーパーブロック
 - i-node
 - i-nodeリスト
 - インデクス方式
 - スペースファイル
 - ディレクトリファイル
 - ディレクトリエントリ
 - 直接ブロック
 - 間接ブロック

UNIX ファイルシステム 11 / 13

練習問題 (2)

- ブロックサイズが8セクタ (4KiB) の場合、直接ブロックだけ用いて表現できるファイルの最大サイズを答えなさい。
- ブロックサイズが8セクタ (4KiB) の場合、1重間接ブロックを用いることによって、直接ブロックだけの場合と比較して、ファイルサイズを最大でどれだけ大きくできるか答えなさい。
- ブロックサイズが8セクタ (4KiB) の場合、2重間接ブロックを用いることによって、直接ブロックと1重間接ブロックだけ使用する場合と比較して、ファイルサイズを最大でどれだけ大きくできるか答えなさい。

UNIX ファイルシステム 12 / 13

練習問題 (3)

5. 4 ページの例がスペースファイルを表現しているとする。また、ブロックサイズ等は「ボリューム内部の配置」で示したものと同じとする。次のアドレスはデータブロックが割り当てられているか答えなさい。

- (a) 第 0x00000000 バイト
- (b) 第 0x00001000 バイト
- (c) 第 0x00010000 バイト
- (d) 第 0x00100000 バイト
- (e) 第 0x01000000 バイト
- (f) 第 0x10000000 バイト

オペレーティングシステム 第17章 ZFS

<https://github.com/tctsigemura/OSTextBook>

ZFS

1 / 24

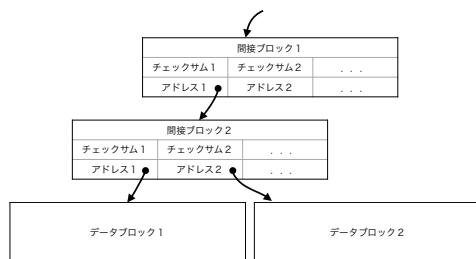
ZFSの特徴

- 2005年にサン・マイクロがOpenSolarisに実装して公開
- オープンソースで開発が続いている（FreeBSD等でも使用可）
- 大きな主記憶、高速マルチプロセッサが前提（8GiB、64bit CPU）
- COW（Copy On Write）でデバイスに書き込む。（デバイスのブロックを上書きしないので前状態を維持）
- Uberblockを書き込みと同時に新しい状態になる。（Uberblockはファイルシステム管理データの根）
- チェックサムにより高い信頼性を確保（次ページ）
- 一瞬でスナップショットやクローンを作成（COWの手法を使用し違いの出たブロックだけコピーし変更）
- ボリュームの代わりにストレージプールを使用（後のページ）
- ファイルサイズ等の制約が事実上無い（ $Zetta = 2^{70}$ ）（ファイルサイズ $2^{64}B$ 、ストレージプールサイズ $2^{70}B$ ）
- ミラー、RAID-Z等が構成され信頼性・可用性が向上
- データ圧縮、重複除去機能があり（ディスクI/Oを少なくする）

ZFS

2 / 24

全ブロックにわたるチェックサム

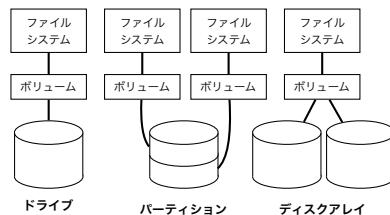


- ブロックポインタがチェックサムを持つ。
- メタデータだけでなく普通のデータにもチェックサムあり。
- チェックサムの不整合が見つかった場合、データの2重化（ミラー）がされていれば、自動的にミラーからデータを修復する。

ZFS

3 / 24

従来の方式のボリューム

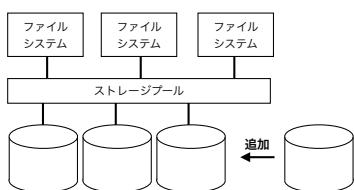


- ファイルシステムの初期化以前にボリュームを決定し、
- 後でサイズの変更などはできない。

ZFS

4 / 24

ZFSのストレージプール

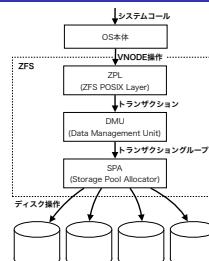


- ストレージプールは沢山のデバイスを収容する。
- ファイルシステムからの要求に応じてブロックを割り付ける。
- C言語プログラムのmalloc()やfree()に似ている。
- ストレージプールに後でデバイスを追加することも可能。

ZFS

5 / 24

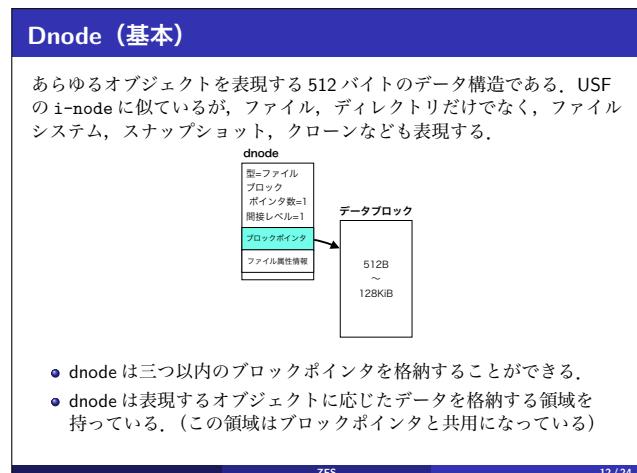
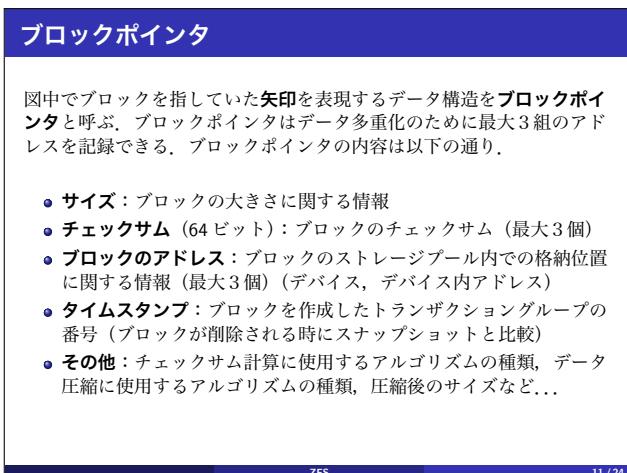
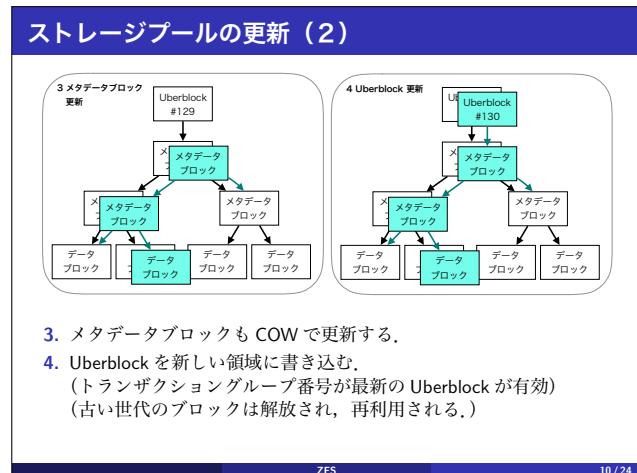
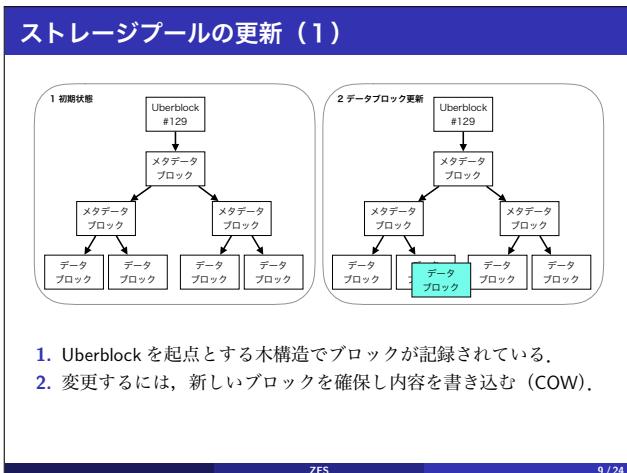
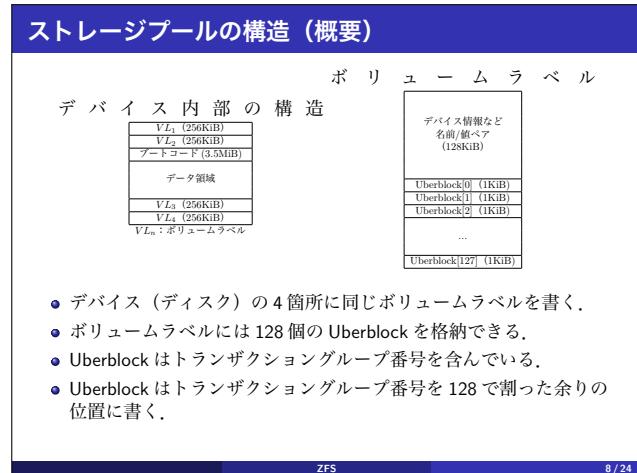
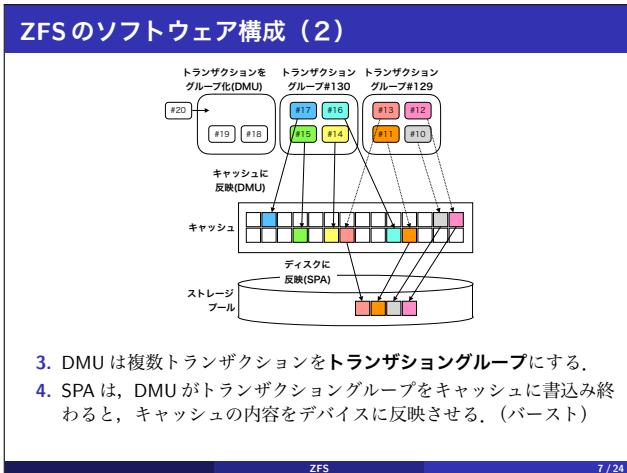
ZFSのソフトウェア構成 (1)

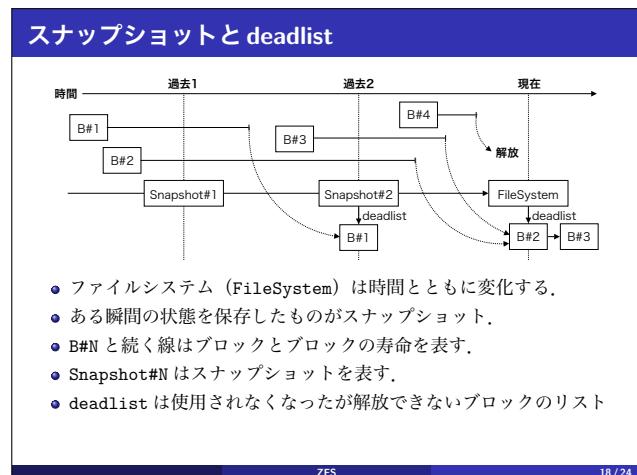
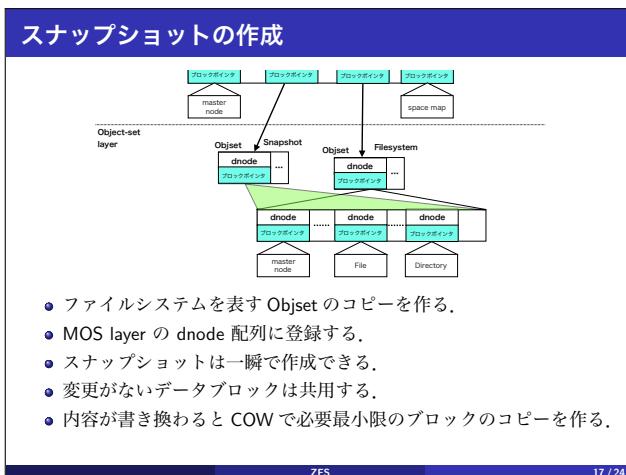
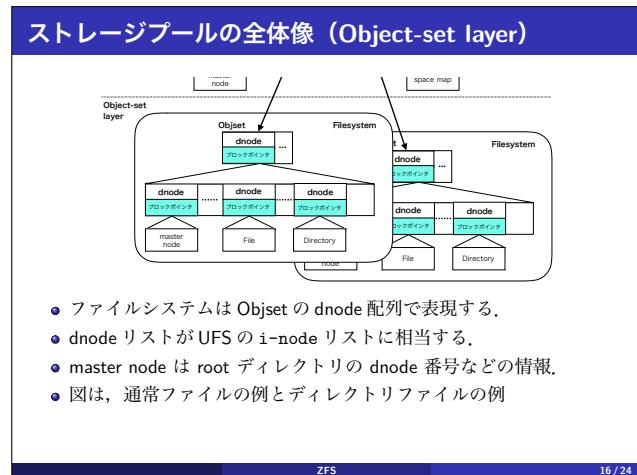
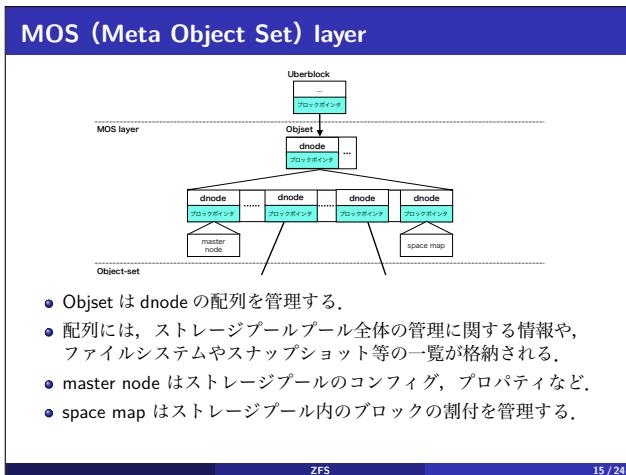
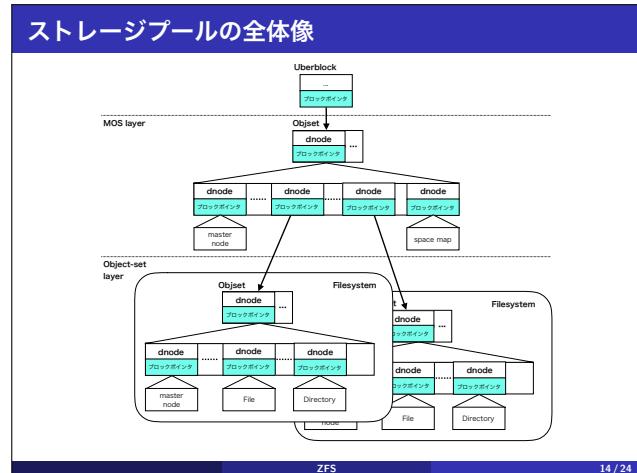
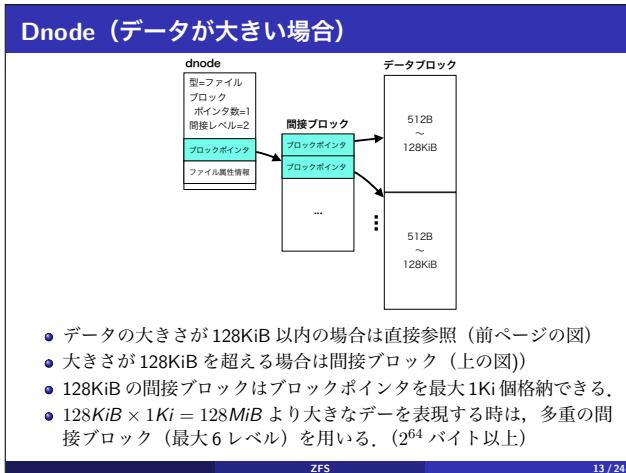


1. システムコールは、OSカーネル本体がVNODE操作に変換する。
2. ZPLはVNODE操作をZFSのトランザクションに変換する。
3. DMUは複数トランザクションをトランザクショングループにする。
4. SPAは、DMUがトランザクショングループをキャッシュに書き込み終わると、キャッシュの内容をデバイスに反映させる。

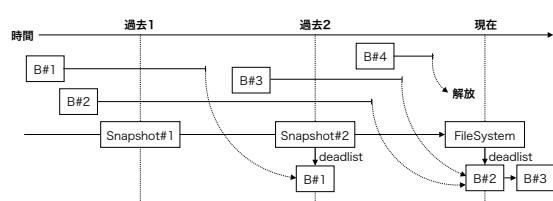
ZFS

6 / 24





ブロックの解放 (1)

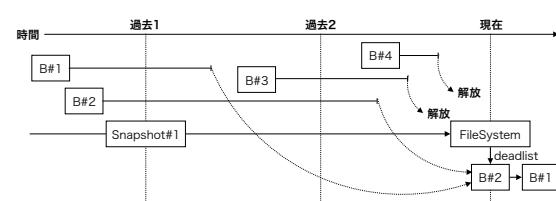


- B#1 は Snapshot#1 で使用されているので解放できない。
- B#2 と B#3 もスナップショットで使用されているので解放できない。
- B#4 はスナップショットで使用されていないので解放できる。
- 解放できないブロックは deadlock に格納される。

ZFS

19 / 24

ブロックの解放 (2)



- Snapshot#2 を削除した状態。
- B#1 と B#2 は Snapshot#1 で使用されているので解放できない。
- B#3 は Snapshot#2 が削除されたので解放できる。
- B#1 と B#2 は FileSystem の deadlock に格納される。

ZFS

20 / 24

まとめ

- ZFS は大きな主記憶と高速な CPU を前提に設計されている。
以下の特徴を持つ。
- COW (Copy On Write) を用い既存のブロックを上書きしない。
 - COW のお蔭でシステムのクラッシュでも壊れない構造を実現。
 - デバイス上の全てのデータについてチェックサムを持つ。
 - スナップショットやクローンを一瞬で作ることができる。
 - ボリュームの代わりにストレージプールを使用する。
 - ストレージプールに後で新しいデバイスを追加可能。

ZFS

21 / 24

練習問題 (1)

次の言葉の意味を説明しなさい。分からぬ言葉は調べなさい。

- COW (Copy On Write)
- メタデータ
- チェックサム
- スナップショット
- クローン
- ボリューム
- ストレージプール
- ZPL (ZFS POSIX Layer)
- DMU (Data Management Unit)
- SPA (Storage Pool Allocator)
- VNODE

ZFS

22 / 24

練習問題 (2)

- Uberblock
- ブロックポインタ
- Dnode
- MOS (Meta Object Set) layer
- Object-set layer
- space map
- master node

ZFS

23 / 24

練習問題 (3)

- トランザクショングループ番号は 64 ビットです。毎秒 100 トランザクショングループを処理したとして、トランザクショングループ番号がオーバーフローするまでに約何年かかるか計算しなさい。
- ZFS で使用できるチェックサム計算アルゴリズムについて調べなさい。
- ファイルを表現する dnode が間接レベル 2 の時、最大のファイルサイズは何バイトになるか計算しなさい。
- ブロックにリンクカウントを設け、スナップショットからの参照数を管理することで、ブロックの解放を判断するアイデアの利点と問題点を挙げなさい。

ZFS

24 / 24

オペレーティングシステム 番外 デッドロック

<https://github.com/tctsigemura/OSTextBook>

デッドロック 1 / 5

デッドロック (deadlock) とは

資源を確保したい複数のプロセス (スレッド) があるとき、資源が解放されるのを待ち合う状態。

```

    graph LR
        subgraph ProcessA [プロセスA]
            procA["procA() {  
    P(S1);  
    ...  
    P(S2);  
    V(S2);  
    V(S1);  
}"]
        end
        subgraph ProcessB [プロセスB]
            procB["procB() {  
    P(S2);  
    ...  
    P(S1);  
    V(S1);  
    V(S2);  
}"]
        end
        S1["資源1 (S1)"]
        S2["資源2 (S2)"]

        P_A_S1["確保済み"] --> S1
        P_A_S2["確保要求 (ブロック)"] --> S2
        P_B_S2["確保済み"] --> S2
        P_B_S1["確保要求 (ブロック)"] --> S1
    
```

デッドロック 2 / 5

デッドロックが発生する条件

デッドロックが発生するための必要条件

- (1) 相互排除 (資源が排他的に利用される)
- (2) 確保待ち (資源を確保した状態で待つ)
- (3) 横取り不可 (使用中の資源を横取りできない)
- (4) 循環待ち (待ちがループしている)

(1), (3) は資源の種類によってはどうしようもない。
プリンタは、相互排除、横取り不可な資源の例。

```

    graph TD
        PA["プロセスA"] -- "確保済み" --> Printer["プリンター"]
        PB["プロセスB"] -- "確保要求 (ブロック)" --> Printer
    
```

デッドロック 3 / 5

デッドロックの発生を防止する方法 (1)

前出の(2), (4)のどちらかを防止すればデッドロックは発生しない。
資源の一括確保 (`P_AND`) … (2) 確保待ちにしない。

```

    graph TD
        subgraph ProcessA [プロセスA]
            procA["procA() {  
    P_and(S1, S2);  
    ...  
    V(S2);  
    V(S1);  
}"]
        end
        subgraph ProcessB [プロセスB]
            procB["procB() {  
    P_and(S1, S2);  
    ...  
    V(S1);  
    V(S2);  
}"]
        end
        S1["資源1 (S1)"]
        S2["資源2 (S2)"]

        P_A_S1["確保済み"] --> S1
        P_A_S2["確保要求"] --> S2
        P_B_S2["確保済み"] --> S2
        P_B_S1["確保要求"] --> S1
    
```

資源の利用率が悪くなる。

デッドロック 4 / 5

デッドロックの発生を防止する方法 (2)

資源の確保順序に制約 … (4) 循環待ちにしない。

```

    graph TD
        subgraph ProcessA [プロセスA]
            procA["procA() {  
    P(S1);  
    ...  
    P(S2);  
    ...  
    V(S2);  
    V(S1);  
}"]
        end
        subgraph ProcessB [プロセスB]
            procB["procB() {  
    P(S1);  
    P(S2);  
    ...  
    V(S2);  
    V(S1);  
}"]
        end
        S1["資源1 (S1)"]
        S2["資源2 (S2)"]

        P_A_S1["確保済み"] --> S1
        P_A_S2["確保要求"] --> S2
        P_B_S2["確保済み"] --> S2
        P_B_S1["確保要求"] --> S1
    
```

不公平が生じるかも (順番付けできない例「食事する哲学者問題」)。

デッドロック 5 / 5