1章　並列OS概論

目次

- 並列OSの目的と他システムとの相違
- マルチプロセッサの分類と研究課題
  - マルチプロセッサの分類
  - 研究課題
- OSの構成法
  - 基本的手法の分類
  - 構成法
- 単一プロセッサ上のOSデータ構造の統一性
  - プロセッサ放棄の分類
  - 解決方法
- 共有メモリ型マルチプロセッサ上のOSデータ構造の統一性
  - 問題点
  - マスタスレーブ方式
  - レントラント方式

計算機システムの高速化技法

- クロック周波数をあげる.
- アーキテクチャ上の工夫
  - パイプライン
  - スーパスカラ
  - VLIW，などなど
- プロセッサを複数にする.

並列OSの目的

- OSの目的
  - ユーザに使い勝手のよい環境の提供： ハードウェアの仮想化
  - 計算機資源の管理： ハードウェア／ソフトウェア資源
- OSの姿・具現化 → 社会に依存
  - ハードウェア環境
    - ✓ マルチプロセッサの高速化，分散システム
    - ✓ マイクロプロセッサの容易な構築
  - 社会環境
    - ✓ 計算能力への要求
    - ✓ 日常生活への浸透： マルチメディア，電子手帳，組込みシステム，数十個のマイコン／家庭
- OSが問うもの
  - 昔
    - ✓ メインフレーム時代，IBMの歴史
    - ✓ 同一タイプユーザ（計算）の大量処理
  - 今
    - ✓ 多様化への対処

並列OSの目的

- 並列計算機の目的
  - 高速処理
- ユーザに使い勝手のよい環境の提供
  - 応用プログラムごとに異なる事項パターンへの対処
    - ✓スケジューリング，メモリマッピング
- 並列計算機資源の効率的管理・制御
  - 計算機資源の効率的な共有
- 用語の問題
  - Multiprocessor OS
    - ✓マルチプロセッサ用OS
    - ✓実装システムだけに言及
  - Parallel OS
    - ✓さらに，OS自体が並列実行

他システムとの相違

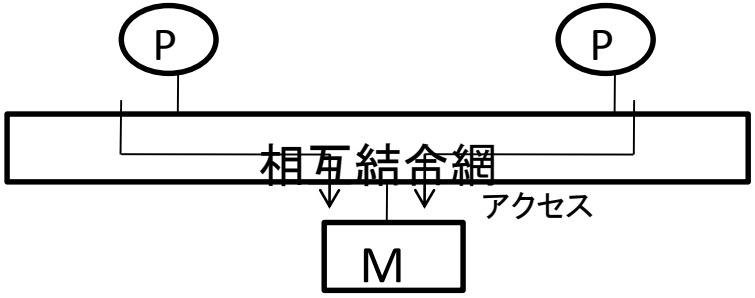
- 単一プロセッサ用OSとの相違
  - 考慮すべき次元
    - ✓単一プロセッサ用OS: 時間という1次元
    - ✓並列OS: 時間と空間の2次元
- 分散OSとの違い
  - 対象システムが類似：並列システム，分散システム
    - ✓共通話題： 負荷分散，プロセス移送，ファイルの透過性，．．．
  - 分散システム
    - ✓自然発生した分散資源の統一ビューを提供
    - ✓構成的，合成的（synthesis）な考え
  - 並列システム
    - ✓人為的にシステムの構築
    - ✓プログラムの人為的並列化
    - ✓解析的（analysis）な考え方

マルチプロセッサの分類

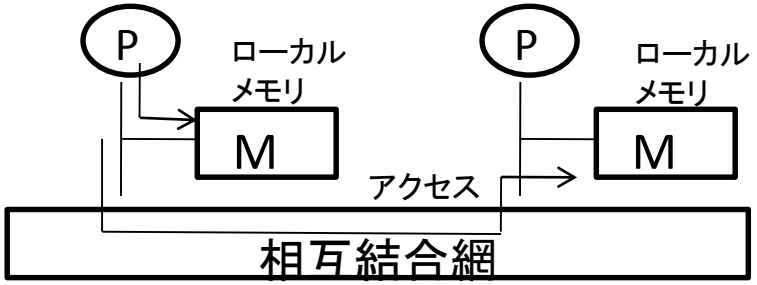
- 大きな分類
  - 共有メモリ型マルチプロセッサ
  - メッセージパッシング型マルチプロセッサ
- 共有メモリ型マルチプロセッサ
  - UMA(Uniform Memory Access)型マルチプロセッサ
    - ✓メモリアクセスが均一
  - NUMA(Non-Uniform Memory Access)型マルチプロセッサ
    - ✓メモリアクセスが不均一
    - ✓CC（Cache－Coherent）NUMA
- メッセージパッシング型マルチプロセッサ
  - NORMA(No Remote Memory Access)型マルチプロセッサ

マルチプロセッサの分類：最近は？

- メニーコア・プロセッサなど，アーキテクチャが複雑化している．
  - L1，L2，L3キャッシュ
  - 上記で，共有キャッシュなど．
  - 従来は，個別／私的キャッシュだったが．．．



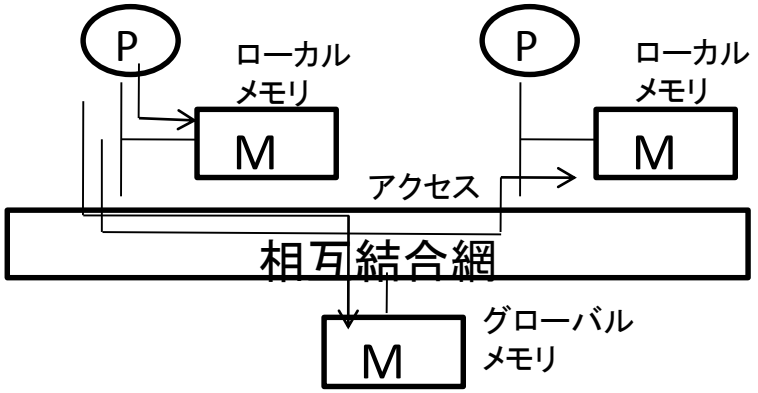
(a)UMA型



(c)NUMA型

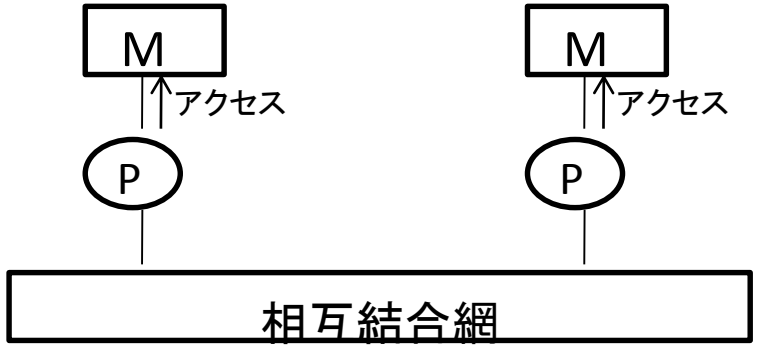
（ローカル／リモートアーキテクチャ）

（他のプロセッサからはリモートメモリ）



(c)NUMA型

（ローカル／グローバル／リモートアーキテクチャ）



(d)NORMA型

図1．1　メモリアーキテクチャからみたマルチプロセッサの分類

研究課題

- OS構成法の研究
- 応用プログラムの高速実行
- ユーザインタフェース

OS構成法の研究

- OSは大規模プログラム
  - 開発，移植，修正，変更，適用が困難.
- ポリシ／メカニズムの分離
- マイクロカーネルアーキテクチャ
- 構成の見通しの良さと性能のトレードオフ

応用プログラムの高速実行環境の提供

- 応用プログラムの実行
  - ユーザまたは並列化コンパイラによって，アクティビティに分解
  - OS提供の実行環境に，アクティビティをマッピング
  - アクティビティの実行
    - ✓ データ／コードへのアクセス
    - ✓ 同期
    - ✓ 入出力
- 軽い実行環境の提供
  - UNIXプロセスは重い.
- スケジューリング機構
  - 同期するプロセス群（スレッド群）の協調を意識
- 高速なメモリアクセス
  - メモリの階層化： キャッシュ，メモリ（ローカル／リモート）
  - NUMAアーキテクチャのメモリアクセスの不均一性
  - 各種高速化技法の採用： キャッシング，パイプライン
- 通信・同期機構
  - ハードウェアアーキテクチャを考慮： アルゴリズム
- 入出力
  - ハードウェアアーキテクチャ，ディスク
- 並列化コンパイラとの役割分担

ユーザインタフェース

- GUI

OSの構成法ー基本的手法ー

- OSプログラムの特徴
  - 大規模プログラム　→　モジュール化
  - ハードウェアと応用プログラムの間に位置する．→　階層化
- モジュール分割
  - 機能分割
    - ✓例：　プロセス管理，メモリ管理，ファイル管理，デバイス管理
  - データ分割
    - ✓PCB，各種管理テーブル
- 階層化
  - 上位層は下位層が提供する機能で実現
  - プロセス管理とメモリ管理の上下関係は？
    - ✓プロセス管理：　プロセス生成にメモリ割当
    - ✓メモリ管理：　メモリ状況に応じてスケジューリング
  - THEシステム
    - ✓階層化の最初のOS（Dijkstra）

OSの構成法

- 単層（monolithic）カーネル
  - 1つのプログラム
  - カーネルデータは共有データ
  - 速い
  - 柔軟性に欠ける.
- プロセス指向システム
  - 複数プロセスの集合体
  - マイクロカーネルアーキテクチャ

単一プロセッサ上のOSデータ構造の統一性

- OS（カーネル）データの統一性とは？
  - データの正しい管理，一貫性
- OSデータ構造
  - プロセス，メモリ，ファイル，デバイス，．．．
- 単一プロセッサの特徴
  - 同時に実行できるコードは1つ

単一プロセッサ上のOSデータ構造の統一性 ―プロセッサ放棄の分類―

1）強制的放棄： 実行中のコードが知らないうちに横取り

- スケジューリングによるコンテキストスイッチ
  - ✓タイマ割込み，ラウンドロビン，
- 割込み
  - ✓割込みハンドラ／ルーチン

2）自発的放棄： 実行コードが明示的に放棄

- 入出力要求を出して放棄

単一プロセッサ上のOSデータ構造の統一性 ―解決法―

- スケジューリングによるコンテキストスイッチ
  - 応用プログラムの実行コード
    - ✓ユーザ走行モード： コンテキストの保存で対処
    - ✓カーネル走行モード： コンテキストスイッチは禁止（UNIX）
- 割込み
  - 割込みハンドラのデータを他カーネルコードがアクセス
    - ✓ベースレベルカーネルコードがアクセス（図１．２参照）
    - ✓ベースレベルカーネルコード実行中には，割込み禁止にする.
- 自発的放棄
  - ファイルアクセスの排他制御
    - ✓読書き問題
  - UNIXの場合（図１．３参照）
    - ✓関数sleepを用いて，自分自身のプロセッサ放棄
    - ✓関数wakeupの処理
      - 待ちの全プロセスを起こす.
      - レディキュー（ランキュー）に連結
    - ✓Sleep, wakeup関数は，ユーザには提供していない.

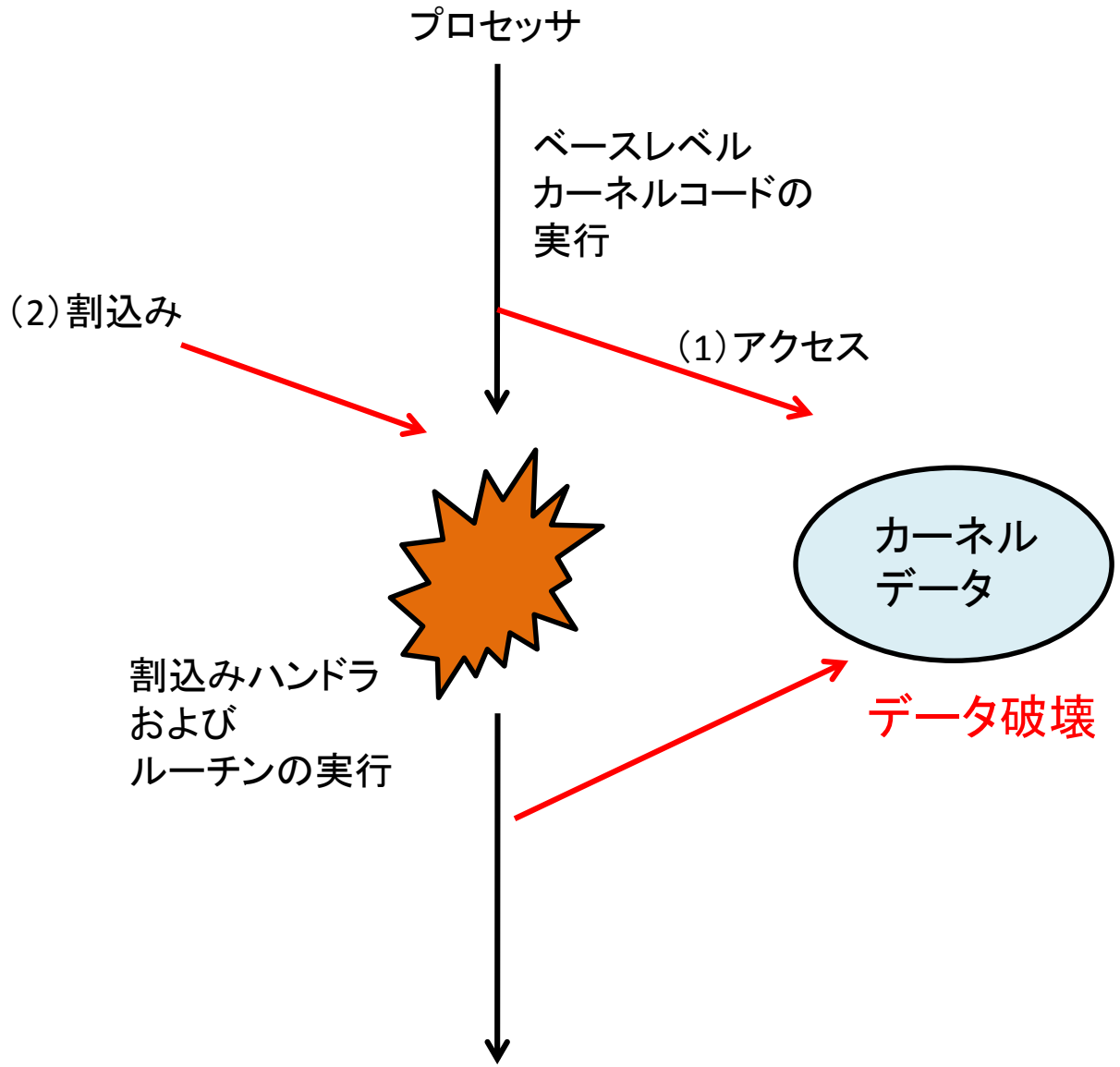


図1．2　割込みハンドラおよびルーチンによるデータ破壊
（単一プロセッサの場合）

UNIXにおける共有オブジェクトの排他制御コード

```
void
lock_object(char *flag_prt)
{
        while(*flag_ptr)
                sleep(flag_ptr);
        *flag_ptr = 1;
}
```

(a) 共有オブジェクトへのロックコード

```
void
unlock_object(char *flag_prt)
{
        *flag_ptr = 0;
        wakeup(flag_ptr);
}
```

(b) 共有オブジェクトへのアンロックコード

図1．3　UNIXにおける共有オブジェクトの排他制御コード（文献26）

共有メモリ型マルチプロセッサ上のOSデータ構造の統一性

- 単一プロセッサの論理を適用しても，解決できない.
- 本質は，複数のプロセッサが同時にカーネルに入れること.
  - スケジューリングによるコンテキストスイッチ
    - ✓カーネルモード走行時のコンテキストスイッチは行わない.
    - ✓他プロセッサがカーネルに入れる.
  - 割込み
    - ✓ベースレベルコード実行時には割り込みは不可
    - ✓割込み不可にできるのは，１台のみ.
    - ✓他プロセッサがベースレベルを実行できる.
  - 自発的放棄
    - ✓ファイルの排他アクセス　→　割込み不可
    - ✓他プロセッサが実行できる.

並列OSの実装方式

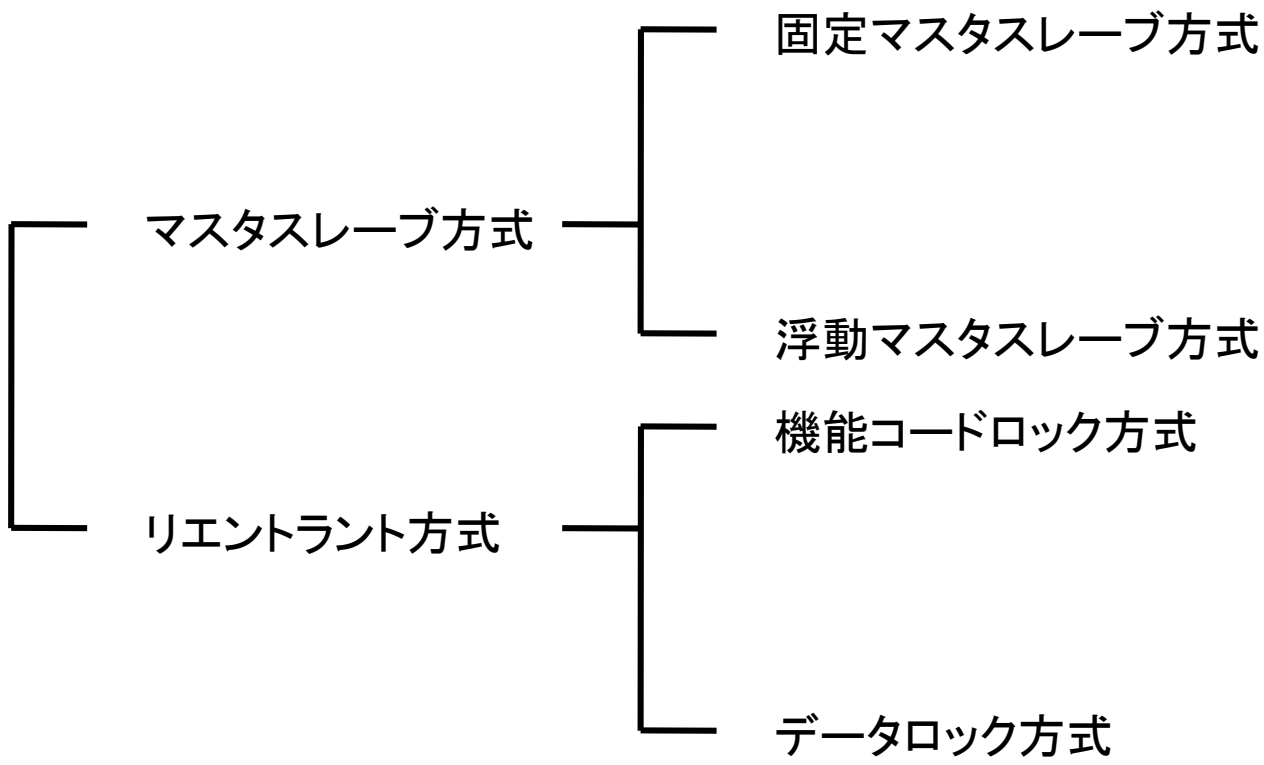


図1．4　カーネルの実現方式

マスタスレーブ方式（1／7）

- 考慮すべき問題
  - プロセッサの割当て方法
    - ユーザモード，カーネルモード
  - ユーザプロセスのプロセッサ間移動
    - ユーザモード／カーネルモード走行のプロセッサが異なる.
  - 割込みへの対処
    - どのプロセッサで割り込みを処理するのか？

マスタスレーブ方式（２／７）

解決策

- ユーザプロセスのプロセッサの割当て方法
  - モードごとにレディキューを分離
    - ✓MP／SP用
  - SP用キューの構成
    - ✓単一，または複数？
- モード間のプロセッサ移動機構
  - SPからMPへの移動
    - ✓実行コンテキストをMP用キューに連結
      - – SPが行う.
    - ✓SPは，その後，他プロセスを実行
  - MPからSPへの移動
    - ✓実行コンテキストをSP用キューに連結
      - – MPが行う.

マスタスレーブ方式（3／7）
―割込みへの対処―

- 割込みの種類
  - 入出力割込み
  - トラップ
  - タイマ割込み
- 入出力割込み
  - デバイスの接続形態に依存
  - MPのみに接続
    - ✓ MPで処理
  - SPにも接続
    - ✓ SPで受け付け処理
    - ✓ MPで実際の処理
- トラップ
  - SPで生じるトラップ
    - ✓ 算術演算オーバ／アンダフロー，特権命令違反，ページフォールト
  - SPでトラップ受付処理
  - MPで実際の処理

マスタスレーブ方式（4／7）
―割込みへの対処―

- タイマ割込み
  - 受付レベル
    - ✓MPのみ？，SPにも？
  - タイマ割込みによるコンテキストスイッチの契機（UNIX）
    - ✓ラウンドロビンによるタイムスライスの終了
      - – コンテキストスイッチ＋実行プロセスをレディキューへ
    - ✓SPレディキューに高優先度のプロセス連結
      - – コンテキストスイッチ＋実行プロセスをレディキューへ
    - ✓シグナルがプロセスへ発行
      - – 実行プロセスをMPへ移動

マスタスレーブ方式（５／７）
―SP用レディキューの排他制御―

- ロックの取り方
  - スピンロック
  - サスペンドロック
  - クリティカルセクションの大きさに依存
- キュー操作
  - キューへの連結（エンキュー）
  - キューからの取出し（デキュー）
  - 検索
- エンキュー操作
  - １）割込み禁止
  - ２）不可分命令を用いて，スピンロック
  - ３）要素をキューに連結
  - ４）ロックの解除
  - ５）割込み禁止の解除（元のレベルに戻す）
- 上記１）の割込み禁止の必要性
  - デッドロックの生じる可能性あり.
  - 図１．５参照

マスタスレーブ方式（6／7）
ーエンキュー操作に伴うデッドロックー

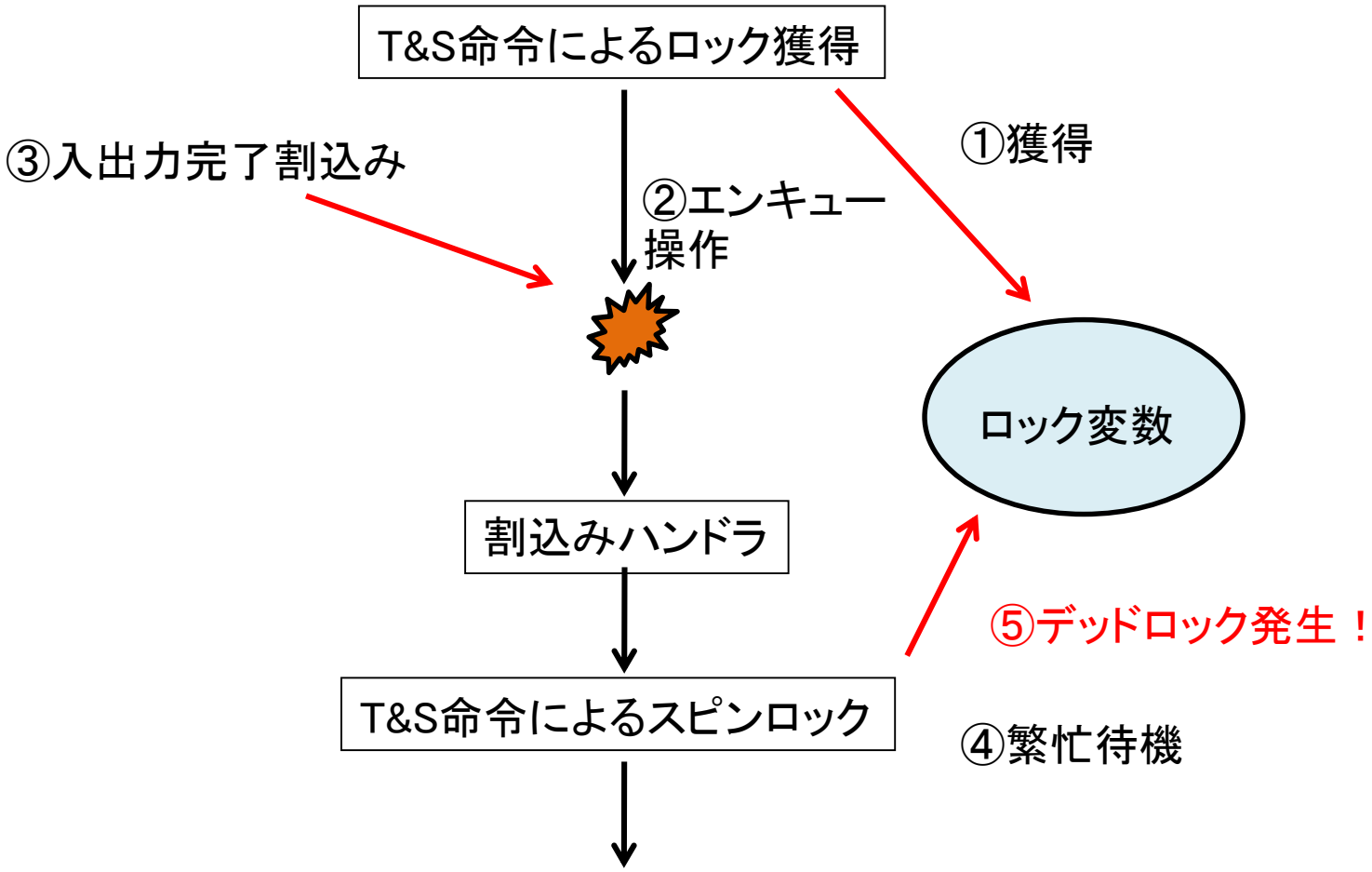


図1．5　エンキュー操作に伴うデッドロック

マスタスレーブ方式(7／7)
ー固定マスタスレーブの利点，欠点ー

- 利点
  - ➢実装が容易
    - ✓単一プロセッサ用の論理が適用できる.
- 欠点
  - ➢MPがボトルネック
- 実装の初期段階で採用

浮動マスタスレーブ方式（１／２）

- 任意のプロセッサがマスタプロセッサになれる.
- 実装方法
  - カーネル全体をロックする.
  - ジャイアントロック（giant　locking）方式とも呼ばれる.
- 割込みハンドラの排他制御
  - 方法
    - ✓ 割込み禁止ではだめ（図１．６参照）
    - ✓ ソフトウェアでスピンロックをとる必要あり,
  - デッドロックの可能性あり.
- デッドロックへの対処
  - ロック変数を１つ.
    - ✓ 競合するカーネルコード実行時には，ロックをかける.
    - ✓ 利点：実装が容易.
    - ✓ 欠点：他デバイスでも待たされる.
  - 割込みデバイス対応にロック変数を設ける.
    - ✓ 利点：他デバイスであれば，待たされない.
    - ✓ 欠点：　デバイス対応のデータの分離が必要で，実現が上記より困難.
- 実装段階
  - 中間段階での実装

浮動マスタスレーブ方式（2／2）

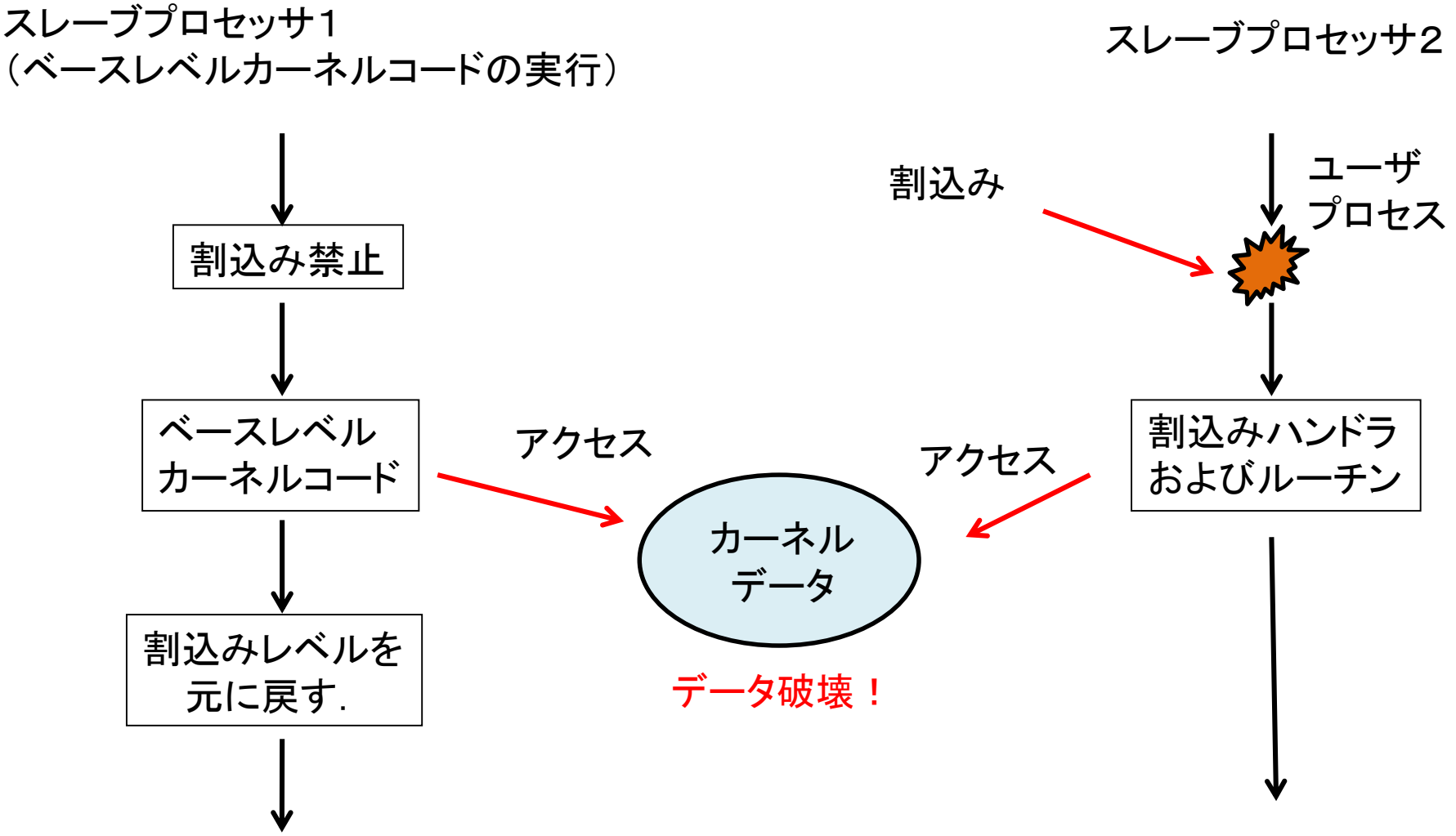


図1．6　割込みハンドラおよびルーチンによるデータ破壊（マルチプロセッサの場合）

固定マスタスレーブと浮動マスタスレーブの比較

- 浮動方式
  - 利点
    - ✓ユーザ／カーネル間のプロセッサ間移動がない.
  - 欠点
    - ✓キャッシュが活用できない.
      - – モードが切り替わるとき，キャッシュ破壊？

リエントラント方式

- カーネル全体をリエントラント（再入可能）にする.
- カーネル内部部分がクリティカルセクション
- 実現方法の分類
  - どの構成要素に注目するか？
    - ✓機能コード
    - ✓データ
  - 排他制御の粒度は？
    - ✓粗粒度ロック方式（coarse-grained locking）
    - ✓細粒度ロック方式（fine-grained locking）
- 組合せ
  - 粗粒度／コードロック方式
  - 細粒度／コードロック方式
  - 粗粒度／データロック方式
  - 細粒度／データロック方式

粗粒度／コードロック方式

- OS要素の機能分割
  - 例： プロセス管理，メモリ管理，ファイルシステム，ネットワークシステム，
- デッドロックの可能性あり
  - メモリマップドファイルの場合
  - メモリ管理： ページフォールト処理でファイルシステムを呼ぶ．
    - ✓メモリ管理，ファイルシステムの順にロックをとる．
  - ファイルシステム： ファイルの仮想ページの操作にメモリ管理を呼ぶ．
    - ✓ファイルシステム，メモリ管理の順にロックをとる．
  - メモリ管理とファイルシステムの間でデッドロックが生じる．
  - デッドロックの対処
    - ✓防止（prevention），回避（avoidance）などの策

細粒度／コードロック方式

- サブシステムをさらに細分割
- 粗粒度と細粒度では，何が違うか？
  - 細粒度では，データ共有の可能性が高くなる.
- 機能の細分割は，必ずしも並列実行を意味しない.
  - データ共有の場合が多くなるから.

データロック方式

- データ構造にロックをかける方式
- 並列実行できる可能性大
- 実現が比較的困難
  - ➢既存OSの再構築
  - ➢データ抽象型，オブジェクト指向設計

以上