# マルチプロセッサアプリケーション プログラミングライブ

本田晋也　　一場利幸
名古屋大学 大学院情報科学研究科
honda@ertl.jp  ichiba@ertl.jp

最終更新日：2011年5月27日

# 概要

- ASPカーネル用のサンプルプログラムをFMPカーネル上で動作させるまでの手順を，FMPカーネル開発者がステップ・バイ・ステップで実演
  - プロセッサ間通信
  - プロセッサ間排他制御
  - マイグレーション

- 実演環境はFMPのサイトから配布していますので一緒に実習していただくことも可能です.
  - http://www.toppers.jp/fmp-kernel.html
- インストール方法は配布パッケージのREADME.txt を参照して下さい

# アジェンダ

FMPカーネルの概要

- マルチプロセッサ用RTOS
- TOPPERSプロジェクトにおけるこれまでの取り組み
  - TOPPERS/FDMPカーネル
  - TOPPERS/SMPカーネル
- TOPPERS/FMPカーネル
  - 仕様・実装
  - 検証
  - 可視化ツール

FMPカーネルアプリケーションプログラミング

- 開発環境について
- ASPカーネルのサンプルプログラムの解説
- ASP用サンプルアプリケーションを1プロセッサで動作させる
- ASP用サンプルアプリケーションを2プロセッサで動作させる
- タスク間の排他制御の方法
- タスクとハンドラ間の排他制御の方法
- 起動時タスクマイグレーション API(mact_tsk)の使い方
- タスクマイグレーションAPI(mig_tsk) の使い方

# 配布パッケージについて

- ファイル構成
  - README.txt
    - 開発環境の構築手順を解説
  - fmp_1_2_log.zip
    - FMP カーネル Release 1.2.0 RC
      - TOPPERS新世代カーネル用コンフィギュレータ Release 1.6.0
      - 演習用のプログラム
  - skyeye_devm_package-1.0.5.zip
    - TOPPERSカーネル向けシミュレーション環境（SkyEye） 1.0.5
  - TLV_1.3.zip
    - TLV Release 1.3（可視化ツール）
- ダウンロード先
  - http://www.toppers.jp/fmp-kernel.html

# FMPカーネルの概要

# FMPカーネルの概要

- マルチプロセッサ用RTOS
- TOPPERSプロジェクトにおけるこれまでの取り組み
  - TOPPERS/FDMPカーネル
  - TOPPERS/SMPカーネル
- TOPPERS/FMPカーネル
  - 仕様・実装
  - 検証
  - 可視化ツール

# 組込みシステムにおけるマルチプロセッサの利用

組込みシステムにおいてもマルチプロセッサの利用が進んでいる

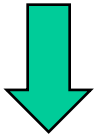
性能向上と消費電力削減の両立

- 低性能CPU複数個の消費電力 < 高性能CPU1個の性能
- 消極的なマルチプロセッサの利用
  - ソフトウェアエンジニアとしては高性能で低消費電力なプロセッサが使いたい

要件の異なるサブシステムの組み合わせ

- リアルタイム性と高機能性の両立
  - 携帯電話，カーナビ，NC工作機
- 積極的なマルチプロセッサの利用

マルチプロセッサに対応したRTOSが必要となる

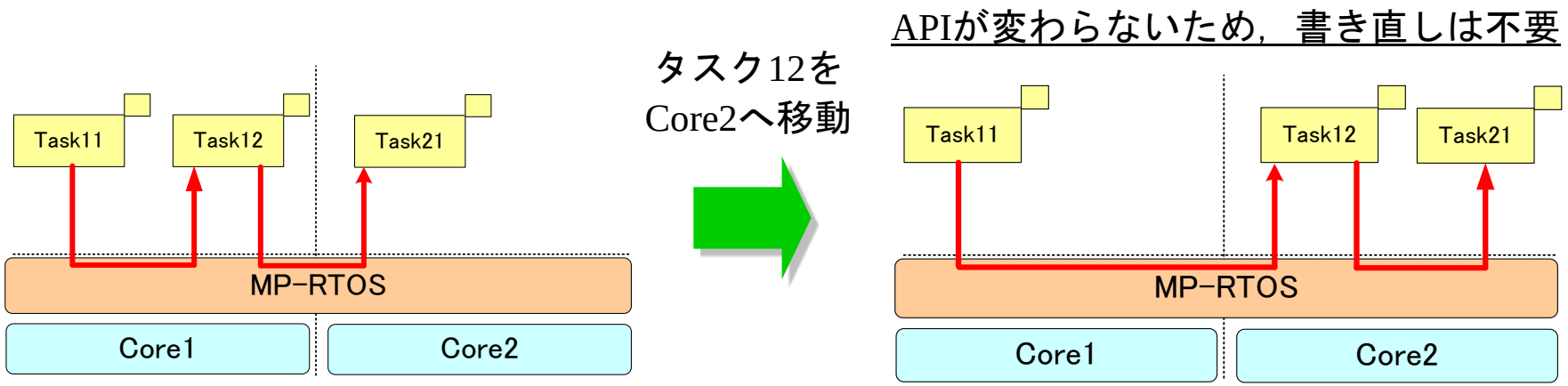


マルチプロセッサに対応したRTOSとは?

# マルチプロセッサ向けOSの機能(1/2)

## プロセッサ間通信のサポート

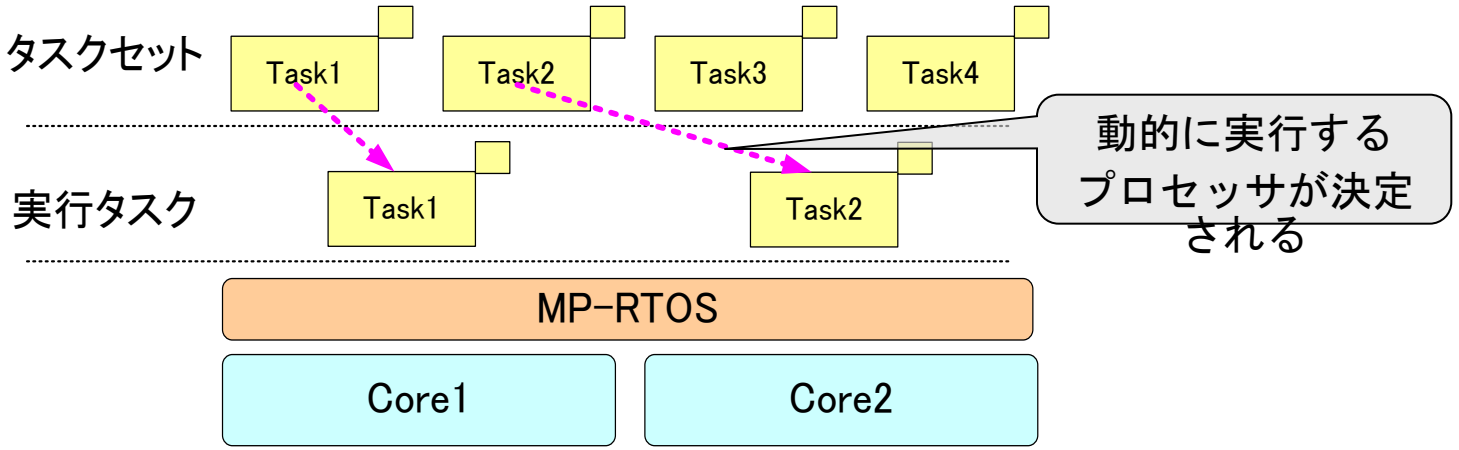
- アプリケーション毎にプロセッサ間の同期・通信を実現する必要がないため，開発工数が減少する
- プロセッサ内通信と互換の同期・通信機能
  - ある処理を別のプロセッサに(動的にも静的にも)容易に移動できる



# マルチプロセッサ向けOSの機能(2/2)

動的な処理の割り当て(場合によっては)

- 負荷に応じて，スループットが最大になるよう，処理(タスク)を動的にプロセッサに割り当てる
- タスクの途中状態での移動が可能
  - タスク切り替えと同じタイミングで可能
- 組込みシステムにおいては，リアルタイム性の観点からデメリットとなる場合があるので，理解した上で使用することが重要(後述)



# OSサポートの視点からのマルチプロセッサの分類

対称型マルチプロセッサOS（SMP-OS）

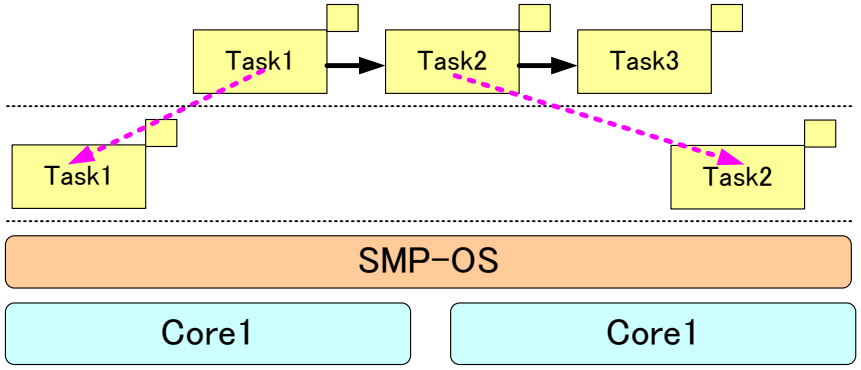
- プロセッサ内通信と互換のプロセッサ間通信
- 動的な処理（タスク）の割り当て
  - グローバルスケジューリング

非対称型マルチプロセッサOS（AMP-OS）

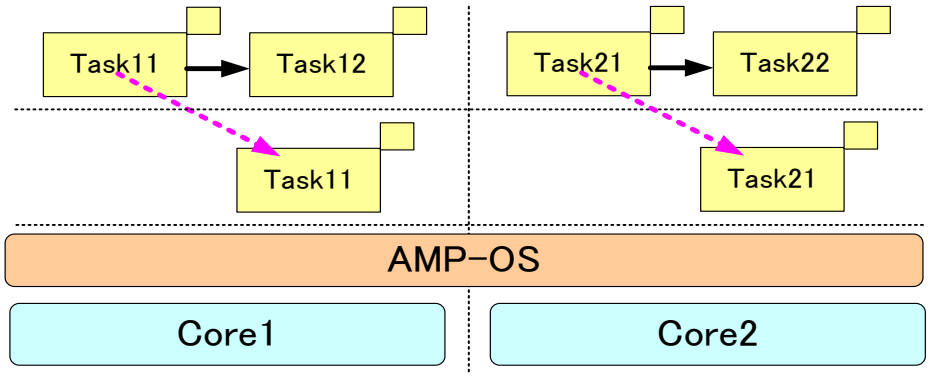
- プロセッサ内通信と互換のプロセッサ間通信
- 静的な処理（タスク）の割り当て
  - ローカルスケジューリング

OSの内部構造の違いによる性質から，AMP-OSの方が組込みに適している場合が多い

SMP-OS




AMP-OS



# SMP-OS：メリットとデメリット

メリット

- 動的な負荷分散が可能で，システムのスループットを向上させることが可能
- ソフトウェアの作りこみなしに性能を得られる可能性

デメリット

- 全てのリソースを各プロセッサからアクセス可能とするため，高速で複雑なバスやコヒーレントキャッシュが必要不可欠
  - ハードウェアのコストや消費電力は大きくなる傾向になる
- 動的な負荷分散を行うため，各タスクのリアルタイム性の保証が困難になる
  - 割り込みハンドラも動的にするとさらに困難に

# AMP-OS：メリットとデメリット

メリット

- 機能を固定することで，ソフトウェア(OS)やハードウェア構成を用途に応じて最適化することが可能
  - プロセッサ間でコードを共有しないため，コヒーレンシキャッシュ等の複雑な回路を必要としない
  - バス構成やメモリ階層を多段にして性能向上を図れる
- 個別のプロセッサの性質はシングルプロセッサに近く，対称型と比較してリアルタイム性の保証が容易

デメリット

- OSでは動的な負荷分散をサポートしていないため，アプリケーションレベルで実現する必要がある

# FMPカーネルの概要

- マルチプロセッサ用RTOS
- TOPPERSプロジェクトにおけるこれまでの取り組み
  - TOPPERS/FDMPカーネル
  - TOPPERS/SMPカーネル
- TOPPERS/FMPカーネル
  - 仕様・実装
  - 検証
  - 可視化ツール

# TOPPERSプロジェクトで開発したマルチプロセッサ用OS

非対称型マルチプロセッサ用リアルタイムOS

- TOPPERS/FDMPカーネル
  - FDMP（Function Distributed Multiprocessor）
- 2005年にカーネル仕様を公開
- 2006年4月にカーネル実装をオープンソースとして公開（1.1）

対称型マルチプロセッサ用リアルタイムOS

- TOPPERS/SMPカーネル
  - SMP（Symmetric MultiProcessor）
- EPSONと名古屋大学との共同研究
- カーネル仕様を会員向けに公開

# TOPPERS/FDMPカーネル

- μITRON4.0仕様を非対称型マルチプロセッサ向けに拡張
- マルチプロセッサ独自の機能は極力設けない

アプリケーション開発を効率化

- μITRON仕様のAPIでプロセッサ間の同期・通信が可能

  ⇒μITRON仕様OS向けのソフトウェア資産が活用可能

性能を重視した実装

- プロセッサ数に対するスケーラビリティ（ロック単位）
- リアルタイム性（特に割込みに対する応答性）を損なわないための工夫（ロック取得ルーチンの工夫）
  - プロセッサに閉じた処理に関しては，シングルプロセッサとほぼ同等の性能

# TOPPERS/SMPカーネル

- μITRON4.0仕様を対称型マルチプロセッサ向けに拡張
- マルチプロセッサ独自の機能は極力設けない

アプリケーション開発を効率化

- μITRON仕様のAPIでプロセッサ間の同期・通信が可能

  ⇒μITRON仕様OS向けのソフトウェア資産が活用可能

排他制御のための機能を追加

- シングルプロセッサ用の排他制御機構では排他制御が実現できないため

  ⇒アプリケーション開発の留意点で説明

- 既存の排他制御も互換のために残しているが，仕様が異なっているため，使用に関しては注意が必要

# FMPカーネルの概要

- マルチプロセッサ用RTOS
- TOPPERSプロジェクトにおけるこれまでの取り組み
  - TOPPERS/FDMPカーネル
  - TOPPERS/SMPカーネル
- TOPPERS/FMPカーネル
  - 仕様・実装
  - 検証
  - 可視化ツール

# TOPPERS/FMPカーネル：開発経緯

新世代カーネルに追従したマルチプロセッサ向けカーネルの必要性

- TOPPERS割込み処理モデルを実装

TOPPERS/FDMPカーネル，TOPPERS/SMPカーネルからの要望

- 動的なタスク移動のサポート
  - サポート範囲はリアルタイム性とのトレードオフ
- リアルタイム性
  - ピュアSMPカーネルではリアルタイム性の保障は困難
- アーキテクチャ最適化
  - 特にメモリアーキテクチャに対する最適化を可能に
- カーネルコード共有
  - カーネルのプログラム（バイナリコード）は，すべてのプロセッサで共有

⇒ TOPPERS/FMPカーネルの開発へ

# TOPPERS/FMPカーネル

リアルタイム性と動的なタスク移動との両立を目指す

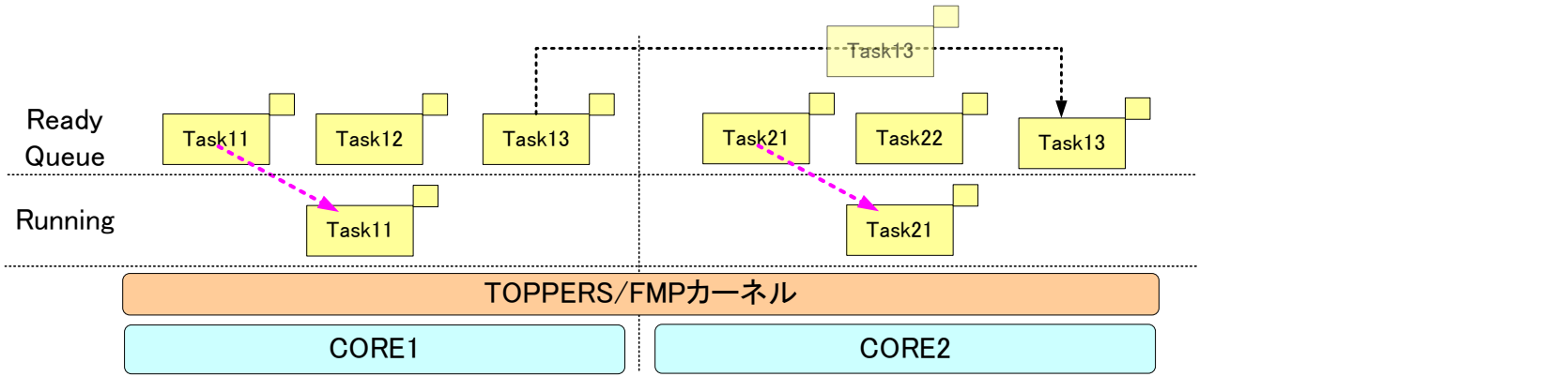
プロセッサ毎のタスクスケジューリング

- リアルタイム性の確保が(比較的)容易

タスクを移動させるAPIを追加

- 負荷変動への対応が可能
  - マイグレート可能なタイミングは要検討

ポリシとメカニズムの分離



# TOPPERS/FMPカーネル：特徴

この講演で示したい!!

- ASPカーネルとの互換性による移行容易性
  - ASPカーネルと互換のAPIでプロセッサ間の通信が可能

- タスクマイグレーションをサポート
  - APIによるマイグレーション
    - タスク実行時のマイグレーション
    - タスク起動時のマイグレーション

- プロセッサ数に対する性能・リアルタイム性のスケーラビリティを実現
  - AMP型OSと同等の性能を実現可能

- ハードウェアアーキテクチャに応じた最適化が可能
  - メモリ，タイマ，排他制御機構

# TOPPERS/FMPカーネル

開発状況

- 2009/5 : Release 1.0.0 オープンソースとして一般公開
- 2010/2 : Release 1.1.0 バグフィックス
- 2011/7 : Release 1.2.0 テストスイートによる検証済みコードの公開予定

仕様

- 新世代カーネル統合仕様書として一般公開中

サポートプロセッサ

- ARM社 MPCore (ARM11/Cortex-A9)
- ALTERA社 Nios II
- ルネサス社 SH4A-MULTI, SH2A-DUAL
- ARMプロセッサの命令セットシミュレータ SkyEye

# TOPPERS/FMPカーネル：利用事例

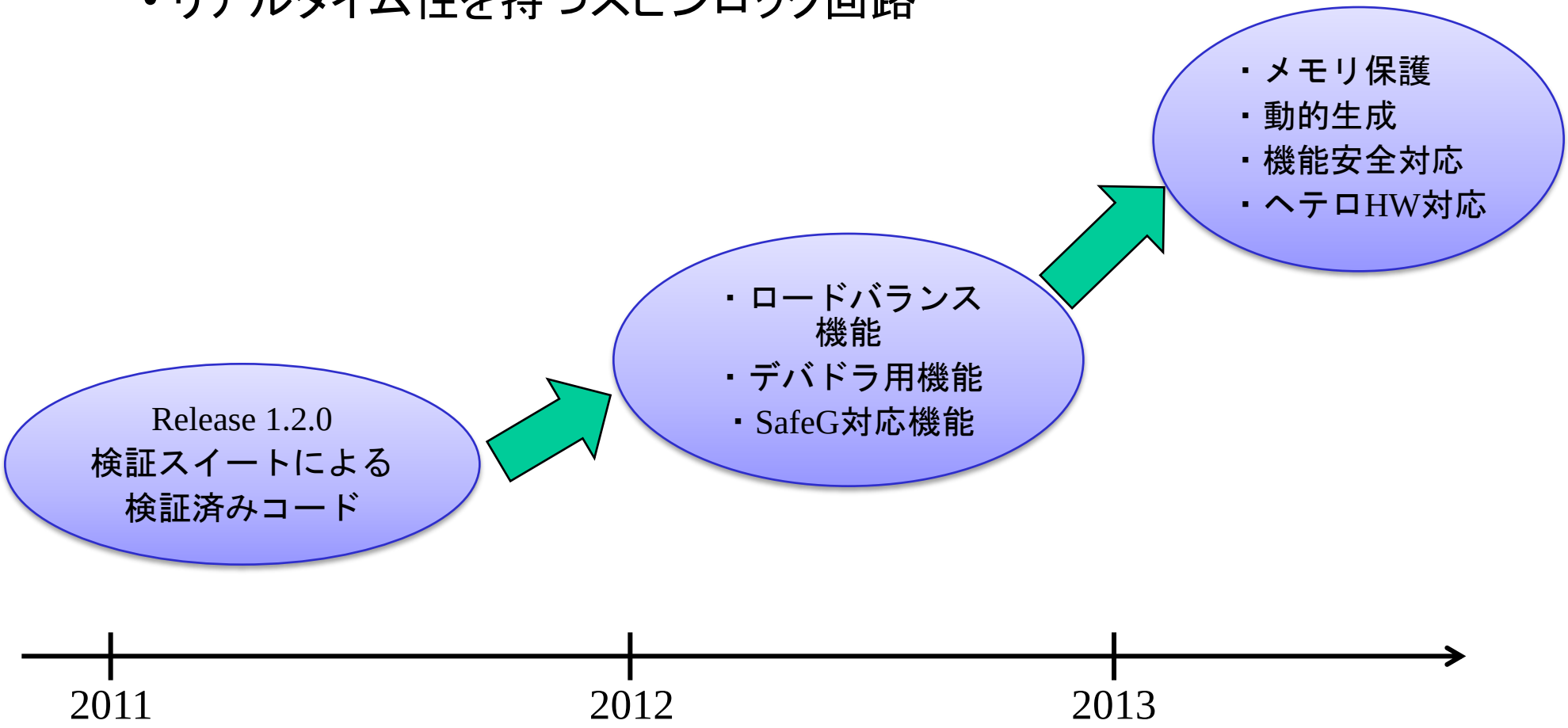
- シャープ株式会社 携帯電話945SH/002SH



出典: SoftBank webページ

# ロードマップ

- 名古屋大学での研究成果を順次反映
- 組込みシステム向けのハードウエア機構も提案
  - リアルタイム性を持つスピンロック回路



# TOPPERS/FMPカーネル仕様：タスクスケジューリング

プロセッサ毎のスケジューリング

- プロセッサ毎に優先度ベーススケジューリングを行う
  - 基本的にはAMP-OSと同等

➡ リアルタイム性の確保が容易

タスクマイグレーション

- タスクをプロセッサ間でマイグレート（移動）させるAPIを用意

➡ マイグレーションポリシーはユーザーに委ねる

マイグレート可能な条件

- システムコールの最悪実行時間を抑えるために制約を設ける
  1. タスクコンテキストから自プロセッサの他のタスクに対して
  2. 自タスクに対して

# TOPPERS/FMPカーネル仕様：クラス

クラス

- オブジェクトの属性を定義
- システムには複数のクラスが定義されている
- カーネルオブジェクト（タスク，セマフォなど）は，いずれかのクラスに属する

クラスの内容（ターゲット依存部で定義）

- オブジェクトコントロールブロックの配置場所
- タスクスタック等（データキューやメモリプール）の配置場所
- 処理単位以外のオブジェクトコントロールブロックの排他制御のために使用するロック
- 処理単位の初期割付けプロセッサ
- 処理単位のマイグレート制限

```
CLASS(TCL_1){
    CRE_TSK(TASK1, { TA_NULL, (VP_INT) 1, ..);
    CRE_TSK(TASK2, { TA_NULL, (VP_INT) 2, ..);
    ....
}
```

# TOPPERS/FMPカーネル仕様：割込み処理

割込み処理モデル

- TOPPERS標準割込み処理モデルに準拠

SMPに準拠

- 割込み要求を処理するプロセッサの指定方法や，特定のプロセッサで処理するか，いずれかのプロセッサで処理するかはターゲット定義とする（CFG_INTで定義）

割込み禁止（CPUロック）

- 発行したプロセッサでのみ有効

➡ 異なるプロセッサで動作するタスク/割込みハンドラ間の排他制御を，割込み禁止で実現することはできない

# TOPPERS/FMPカーネル仕様：追加API

タスクマイグレーション機能

- mig_tsk(ID tskid, ID prcid)
- mact_tsk(ID tskid, ID prcid)

タイムイベントハンドラの起動時プロセッサ指定

- msta_cyc(ID cycid, ID prcid)/msta_alm(ID cycid, ID prcid)

スピンロック機能

- loc_spn/iloc_spn,unl_spn/inul_spn
  - タスクと割込みハンドラ間の排他制御機構

プロセッサIDの取得

- get_pid(ID *p_prcid)

他プロセッサのタスクの優先順位の回転

- mrot_rdq(PRI tskpri, ID prcid)/imrot_rdq(PRI tskpri, ID prcid)

# TOPPERS/FMPカーネル仕様：マイグレーション関連API詳細

- mig_tsk(ID tskid, ID prcid)
  - tskid で指定したタスクを prcid で指定したプロセッサに移動させる
  - prcid にPRC_INIT（0）を指定すると，初期所属プロセッサを指定する
  - tskidに指定可能なタスク
    1. タスクコンテキストから自プロセッサの他のタスクに対して
    2. 自タスクに対して

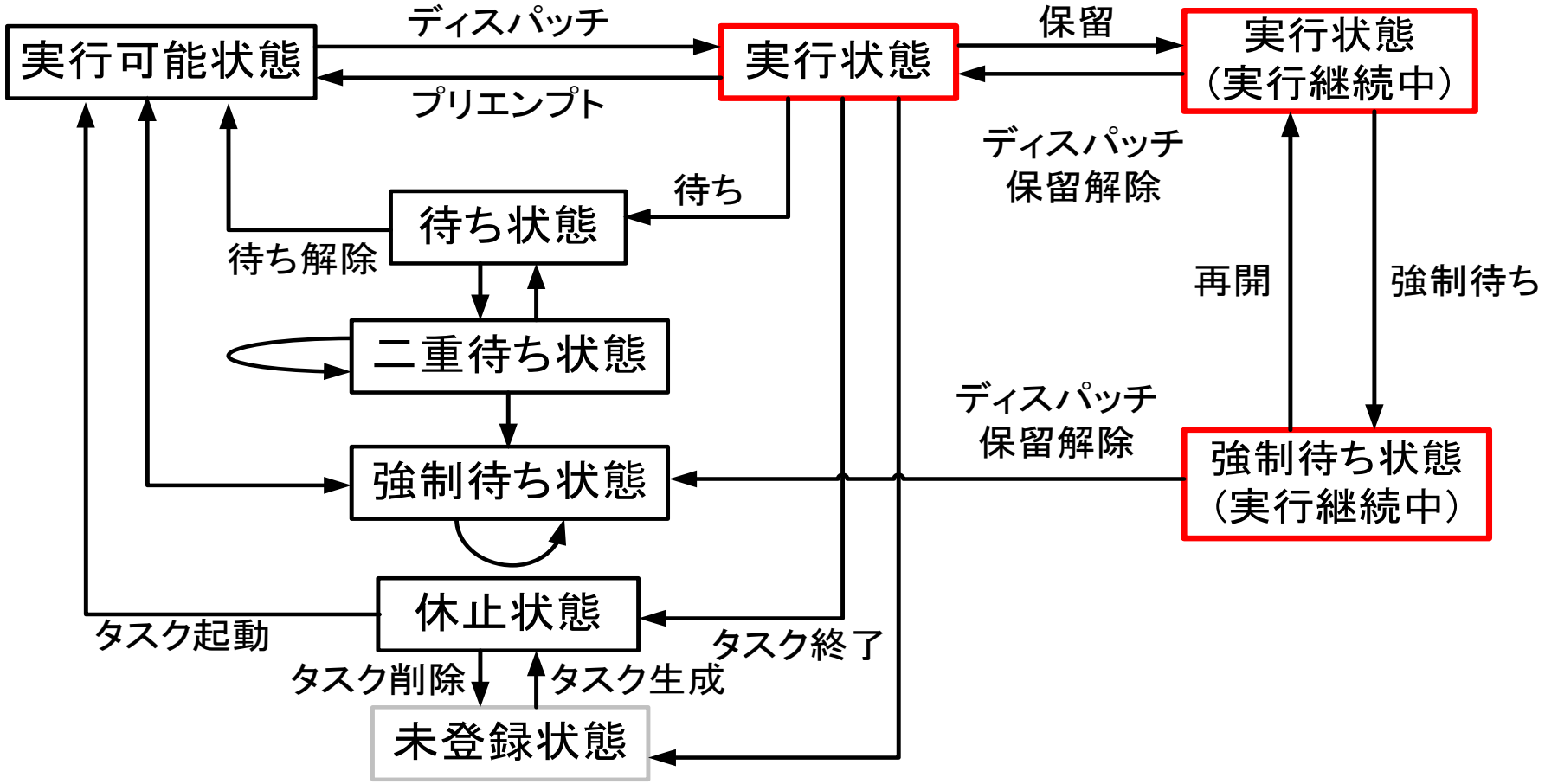
- mact_tsk(ID tskid, ID prcid)/imact_tsk(ID tskid, ID prcid)
  - tskid で指定したタスクを prcid で指定したプロセッサで起動する
  - prcid にPRC_INIT（0）を指定すると，初期所属プロセッサを指定する
  - タスクが休止状態以外なら，起動要求をキューイングする．キューイングされた場合は，タスク終了時に指定されたプロセッサで再起動する
  - tskidに指定可能なタスクに制限はない

# TOPPERS/FMPカーネル仕様：仕様変更API

- act_tsk(ID tskid)/iact_tsk(ID tskid)
  - タスクを所属プロセッサで実行する
  - キューイングした場合は，タスク終了時後に終了時に所属しているプロセッサでタスクを実行する
- ter_tsk(ID tskid)
  - 発行したタスクと同じプロセッサに所属しているタスクのみを指定可能とする
- ref_tsk(ID tskid, T_RTSK *pk_rtsk)
  - T_RTSK に所属プロセッサIDを追加
- 初期化ルーチンの実行
  - マスタプロセッサの初期化ルーチンを実行後，スレーブプロセッサの初期化ルーチンを実行する

# TOPPERS/FMPカーネル仕様：タスク状態

sus_tsk()でのみタスク状態不整合問題が発生する



# TOPPERS/FMPカーネル実装

ASPカーネルベース

- シングルプロセッサ向けのTOPPERS次世代カーネルであるASPカーネルをベースに開発

前提

- ROMは比較的大きなサイズを使用可能とする
  - ノンコヒーレントキャッシュが有効
  - 各プロセッサのプライベートメモリにコピーを持つ
- #ifdef は極力使用しない
  - 使用する場合は，大きな単位で使用する

開発環境

- ARMのISS（Skyeye）をマルチプロセッサ拡張
- ISSをプロセッサ個数分起動してデバイスマネージャにより接続
  - Windowsの共有メモリとMutexを使用

# TOPPERS/FMPカーネル実装：動作アーキテクチャ

- 各プロセッサで，プログラムや固定データに対して，同一アドレスでアクセス可能であること．それぞれのプロセッサがアクセスする物理的なメモリは異なっていてもよい
- 各プロセッサで，同一のアドレスでアクセス可能なRAMがあること
- 任意のプロセッサに割込み（プロセッサ間割込み）を発生可能であること
- プロセッサ間での排他制御のための機構を持つこと
  - 例：test & set 命令，Mutex回路
- プロセッサ間の排他制御機構を用いてロックを最低1個作成可能であること
  - 1個のみの場合はジャイアントロックのみサポート
  - プロセッサ数x2個作成可能であるとプロセッサロックもサポート
  - ロックの作成個数に上限がないと細粒度ロックもサポート
- 各プロセッサが自プロセッサを判別可能であること（プロセッサID等）

# TOPPERS/FMPカーネル実装：コンフィギュレーション

様々なハードウェアに適用可能とするため，
OSの実現方法を複数の方式から選択可能に

- ロック方式

  ジャイアントロック（G_LOCK）
  - 単一のロックで全てのカーネルデータを排他制御する

  プロセッサロック（P_LOCK）
  - プロセッサ毎にタスクロックとオブジェクトロック

  細粒度ロック（F_LOCK）
  - プロセッサ毎にタスクロック
  - オブジェクト毎にオブジェクトロック

# TOPPERS/FMPカーネル実装：コンフィギュレーション

- タイマ方式

  ローカルタイマ

  - プロセッサ毎にカーネルティックタイマを持つ
  - タスクが時間待ちになる場合は，所属するプロセッサのタイムイベントキューに接続される
  - タイムイベントハンドラのマイグレーション

  グローバルタイマ

  - システムで共通のカーネルティクタイマを持つ
  - ロック方式はジャイアントロック限定
  - どのプロセッサがカーネルティックを生成するかは，ターゲット依存部で決定する

- スピンロック方式

  ネイティブ方式

  - ハードウェアのスピンロック機構を直接呼び出す

  エミュレーション方式

  - 1個のハードウェアのスピンロック機構をOSにより抽象化する

# FMPカーネルの概要

- マルチプロセッサ用RTOS
- TOPPERSプロジェクトにおけるこれまでの取り組み
  - TOPPERS/FDMPカーネル
  - TOPPERS/SMPカーネル
- TOPPERS/FMPカーネル
  - 仕様・実装
  - 検証
  - 可視化ツール

詳細は裏のセッションで(?)

# TOPPERS新世代カーネルに対するテスト

包括的なテストは未実施

- RTOSのテストは規模が膨大で工数が大きい
- 大学では仕様検討/性能評価などの研究を中心としている
- RTOSを実製品へ組み込む際，利用者側でテストを行っている

FMPカーネルに対するテスト

- マルチプロセッサ対応RTOSのテスト開発に対する経験が少ない
- テスト開発/実施の工数がシングルプロセッサより肥大化する
- プロセッサ間の実行タイミングに依存したテストを考慮する必要がある

名古屋大学 附属組込みシステム研究センター（NCES）では，2009年度よりコンソーシアム型の研究組織を立ち上げ、上記の問題を解決するためのテストスイートの開発を行っている

# コンソーシアム型研究組織とは

名古屋大学 組込みシステム研究センターが
設定した研究テーマに対して，複数の企業・団体が
参加し共同で研究・開発を行う
(研究成果は一定期間後にオープンとする)

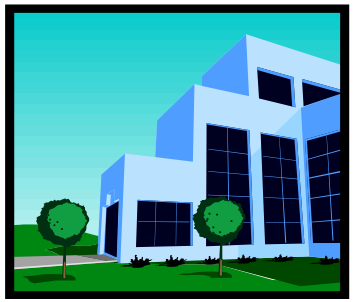
教員／研究者

コンソーシアム型
研究組織

研究者／技術者
(受託研究員)

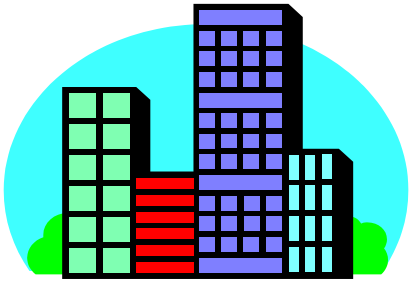
企業のRTOSに関する
経験／知見

・人材育成
・ビジネス発展



名古屋大学
組込みシステム研究センター

産学官連携による
技術課題の解決



参加企業

# マルチプロセッサ対応RTOSの研究・開発

マルチプロセッサの重要性が高まり，利用したい
企業が増えているが課題が多い

- マルチプロセッサ対応RTOSの自社開発は困難
  - 重複投資となり，製品の差別化に結びつかない
  - 各企業で個別に開発するよりも共同研究して
    共有した方が品質向上にも繋がる
  - マルチプロセッサの検証に関する経験が少ない
- 自社開発しなくてもRTOSを利用する上で，RTOSに精通した技術者は必要

2009年度より，メインの研究テーマを
「マルチプロセッサ対応RTOSの検証手法の開発」として，5社1機関の
参加を得てコンソーシアムを形成

# 開発成果物

TTSP(TOPPERS Test Suite Package)

- ASP/FMPカーネル APIテストスイート
  - ASPカーネル ：1786件
  - FMPカーネル： 2593件
- TTG(TOPPERS Test Generator)
  - テストプログラム生成ツール
- コンフィギュレーションツール
  - 環境設定ファイル
  - TTGを用いた実行モジュール生成ツール

# 入手方法


TOPPERSサイトから
ダウンロード可能

http://www.toppers.jp/ttsp.html

- ASPカーネル1.7.0に対応
  (SILテストは除く)
- SILテストおよび
  FMPカーネル用は,
  **2012年3月末**に公開予定

# FMPカーネルの概要

- マルチプロセッサ用RTOS
- TOPPERSプロジェクトにおけるこれまでの取り組み
  - TOPPERS/FDMPカーネル
  - TOPPERS/SMPカーネル
- TOPPERS/FMPカーネル
  - 仕様・実装
  - 検証
  - 可視化ツール

# 開発環境(デバッグ環境)の課題

## マルチプロセッサ環境でのデバッグ

- マルチプロセッサ環境では各プロセッサが独立に並列動作
  - ブレークポイントやステップ実行を用いたデバッグが困難
    - ➔実行後のトレースログの解析によるデバッグが有効

## トレースログの解析によるデバッグ

- RTOSやシミュレータ、エミュレータなどが出力するトレースログを解析することによって動作を確認する
- 開発者がトレースログを直接扱うのには限界がある
  - トレースログのサイズや処理の複雑さによっては解析不可能
  - ➔トレースログの解析を支援するツールの要求

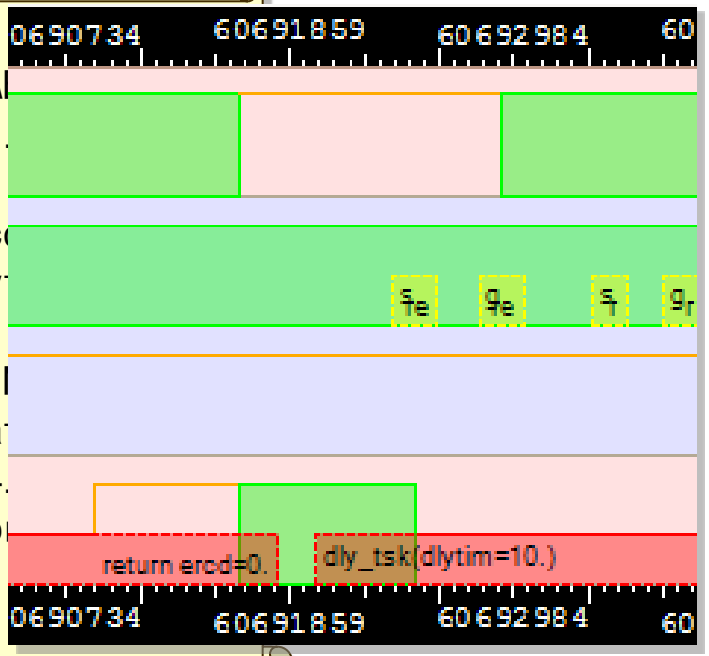
**➔トレースログを可視化表示するツールが重要となる**

# 2プロセッサ上で動くRTOSのトレースログの例

- ×時系列に各プロセッサの動作が分散
- ×膨大な量
  - ➘約1msの間に11個のログ

```
                    ⋮
[60690867]:[1]: task 4 becomes RUNNA
[60691406]:[1]: dispatch from task 1
[60691582]:[1]: dispatch to task 4.
[60691788]:[1]: leave to dly_tsk erc
[60691975]:[1]: enter to dly_tsk dly
[60692360]:[2]: enter to sns_ctx
[60692484]:[1]: task 4 becomes WAITI
[60692586]:[2]: leave to sns_ctx sta
[60692708]:[1]: dispatch from task 4
[60692798]:[2]: enter to get_pid p_p
[60692914]:[1]: dispatch to task 1.
                    ⋮
```

可視化

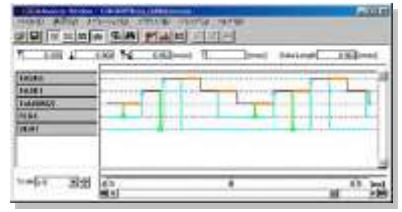



➔トレースログを可視化表示し解析を支援

# 既存の可視化ツール

## 組込みシステム向けデバッガソフトウェア

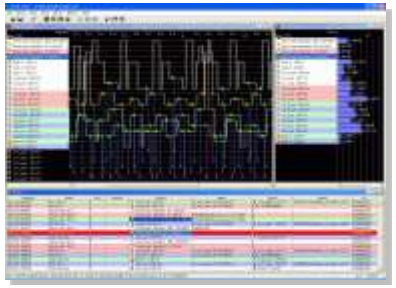
- PARTNER-Jetイベントトラッカー
- WatchPoint OSアナライザ



WatchPoint OSアナライザ
https://www.sophia-systems.co.jp/ice/products/watchpoint

## 組込みシステム向け統合開発環境

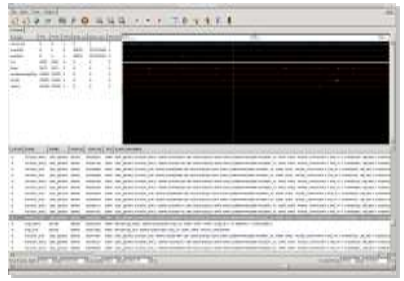
- EvenTrek
- QNX System Profiler



EvenTrek
http://www.esol.co.jp/embedded/eb_multicore2.html

## Unix系OSのトレースログプロファイラ

- Dtrace-Chime (Solaris)
- LTTV(Linux)



LTTV
http://ltt.polymtl.ca/screenshots/

×ログの形式が標準化されていない

➔汎用性に乏しい

×可視化表示項目が提供されているものに限られる

➔拡張性に乏しい

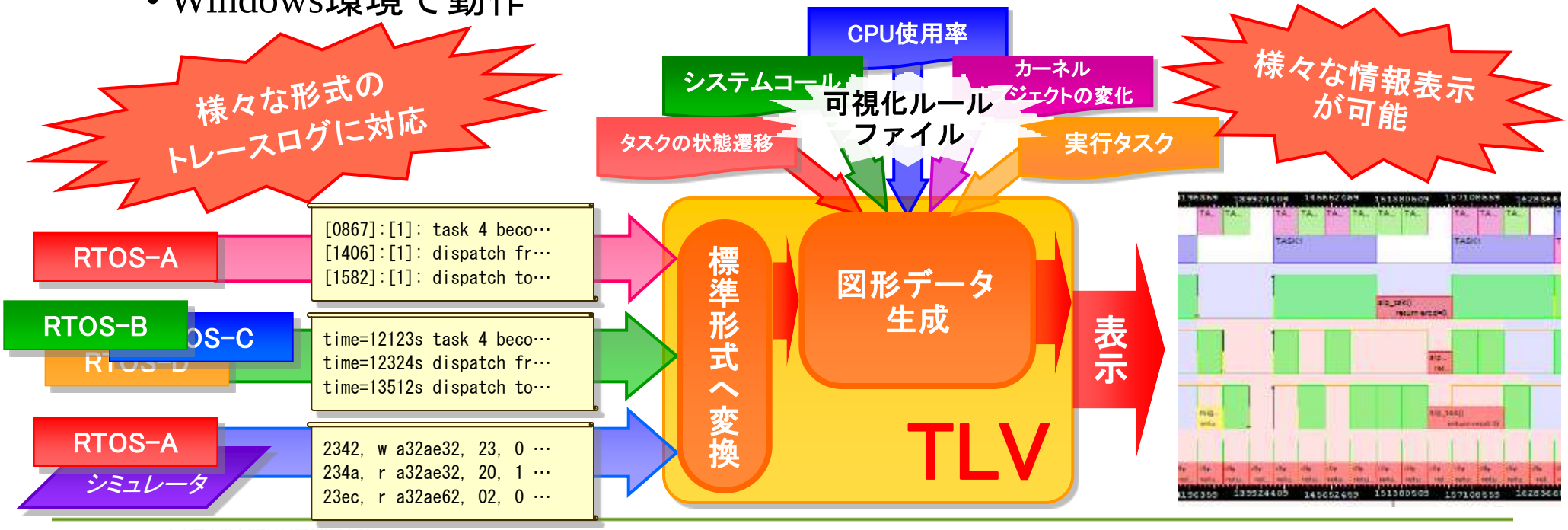
# トレースログを可視化表示するツール

## 開発目標

- 汎用性　…　ログの形式に非依存化
- 拡張性　…　可視化表示項目をプラグイン化

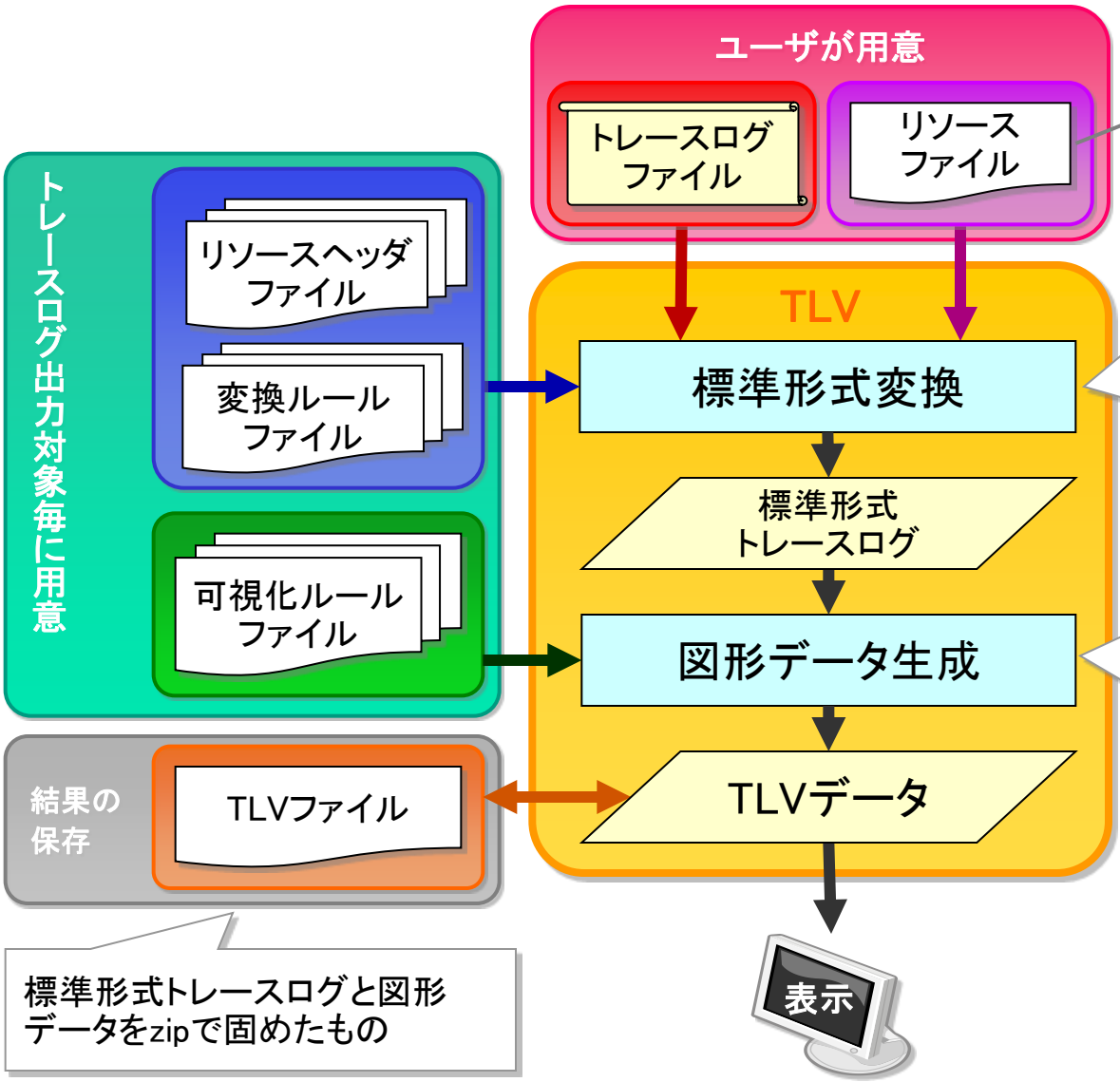
✔TOPPERSプロジェクトからオープンソースでリリース

- Windows環境で動作



# TLVのスクリーンショット

# TLV全体像




読み込むトレースログの形式や
標準形式トレースログで扱う
リソースなどを定義したファイル

読み込んだトレースログを
変換ルールファイルに従い
標準形式のトレースログに変換
➔ ***汎用性の実現***

可視化ルールファイルを
標準形式トレースログに適用し
図形データを生成
➔ ***拡張性の実現***

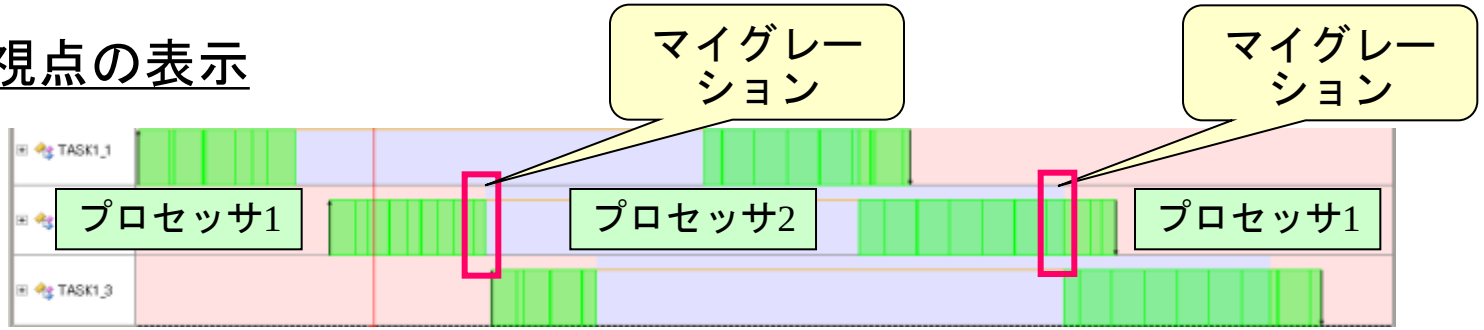
標準形式トレースログと図形
データをzipで固めたもの

✔トレースログファイル以外のファイ
ルはJson形式で記述

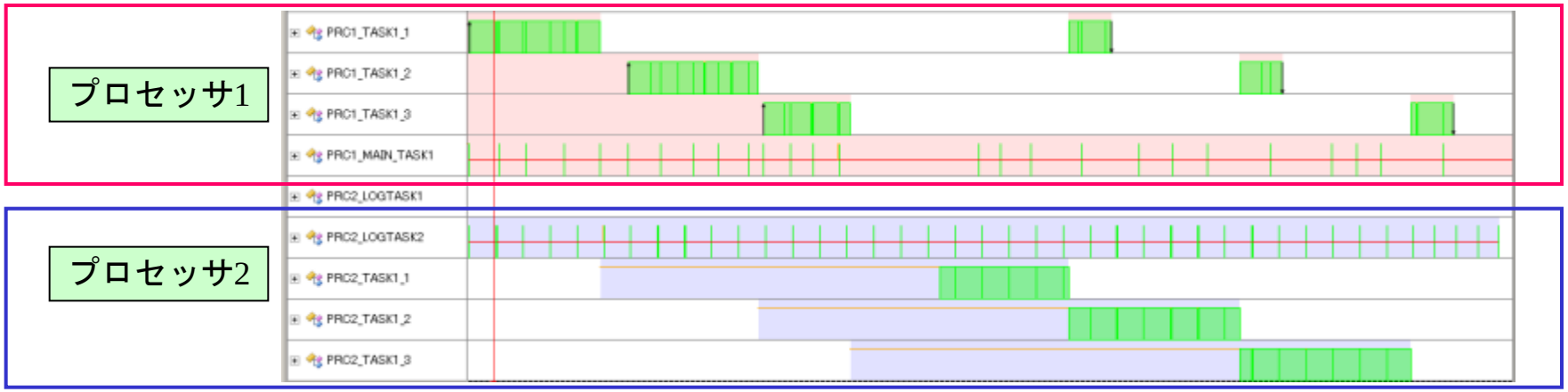
# FMPの可視化方式

- TOPPERS/ASPカーネルのカーネルログ表示
- TOPPERS/FMPカーネルのカーネルログ表示
  - マイグレーションの表現方法で2種類用意

タスク視点の表示




プロセッサ視点の表示



# FMPカーネルアプリケーションプログラミング

# 概要

- 目的
  - ASPカーネル用のサンプルプログラム（sample1）をFMPカーネル上へ移行する手順の習得
  - FMPカーネルの特徴的なAPIの利用方法の習得

- 演習の流れ

  ステップ1. 開発環境について
  ステップ2. ASPカーネルのサンプルプログラムの解説
  ステップ3. ASP用サンプルプログラムを1プロセッサで動作させる
  ステップ4. ASP用サンプルプログラムを2プロセッサで動作させる
  ステップ5. タスク間の排他制御の方法
  ステップ6. タスクとハンドラ間の排他制御の方法
  ステップ7. 起動時タスクマイグレーションAPI(mact_tsk)の使い方
  ステップ8. タスクマイグレーションAPI(mig_tsk)の使い方

# アジェンダ

ステップ1. 開発環境について
ステップ2. ASPカーネルのサンプルプログラムの解説
ステップ3. ASP用サンプルプログラムを1プロセッサで動作させる
ステップ4. ASP用サンプルプログラムを2プロセッサで動作させる
ステップ5. タスク間の排他制御の方法
ステップ6. タスクとハンドラ間の排他制御の方法
ステップ7. 起動時タスクマイグレーションAPI(mact_tsk)の使い方
ステップ8. タスクマイグレーションAPI(mig_tsk)の使い方

# 開発環境：GCC

- GNUプロジェクトにより開発されているオープンソースのコンパイラ
- GCCは「GNU Compiler Collection」の略であり、名前が示すように多くの言語をサポート
  - C、C++、Objective-C、FORTRAN、Java、Ada
- 多くの種類のプロセッサをサポート
  - Alpha、ARM、AVR、H8、IA64、M32R、M68K、MIPS、SH、SPARC、V850
- GCCはアセンブラやリンカとしてbinutilsを呼び出す
  - アセンブラ(gas)、リンカ(ld)、オブジェクトダンプ（objdump）
- デバッガとしてはGNUプロジェクトより同じくオープンソースのソフトウェアとしてgdbが提供されている

# 開発環境：Cygwin

- 一般的なGNU開発プログラムを含むUNIXのプログラムをWindows上で動作させるための環境
- Cygwinライブラリ（Cygwin.dll）によりUNIXのシステムコールを提供（バーチャルマシンではない）
- ほぼ全てがGPL/X11ライセンスのフリーソフトウェア
- Cygnus Solution社（現在はRed Hat社の一部）が開発
- インストールはCygwinのホームページからダウンロードできるインストーラを用いる
- インストール方法は書籍を参考のこと
  - Cygwin+CygwinJE-Windowsで動かすUNIX、佐藤 竜一、アスキー
  - Cygwin—Windowsで使えるUNIX環境、川井 義治、米田 聡、ソフトバンクパブリッシング

# 開発環境：SkyEye

ARMベースのプロセッサをシミュレートする
オープンソースの命令セットシミュレータ

- TOPPERSカーネル向けにオリジナルのSkyEye(1.2.4)を拡張
  - マルチプロセッサ対応（プロセッサ数の上限なし）
  - 外部デバイスモデル対応
  - 実行タイミング制御
  - GCOV対応（カバレッジ取得）

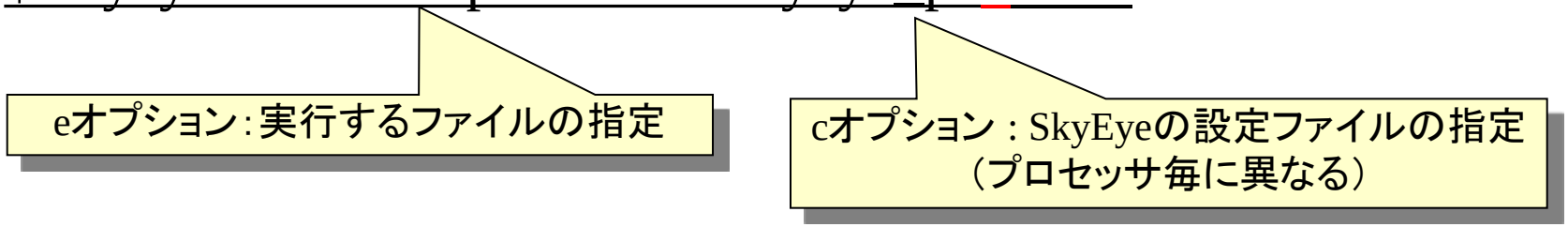
- 最新リリース
  - 1.0.5（2011/03/18）
  - Web : http://www.toppers.jp/sim.html
  - 解説論文 : http://www.jstage.jst.go.jp/article/jssst/27/4/4_24/_pdf/-char/ja/

# SkyEye : 2プロセッサでの実行方法

- Cygwinを2つ起動して，それぞれでSkyEyeを実行する

- プロセッサ1

  $ skyeye.exe –e fmp.exe –c ../skyeye_pe1.conf




- プロセッサ2

  $ skyeye.exe –e fmp.exe –c ../skyeye_pe2.conf

# SkyEye：デバッグ方法（参考情報）

- SkyEyeをデバッグモードで起動
  $ skyeye -d
- Cygwinを新たに開き、実行ファイルのある場所へ移動。移動後、以下のコマンドでGDBを実行する
  $ arm-elf-gdb [ファイル名]
  FMPの場合はgdb.iniが用意されている
  $  arm-elf-gdb -x gdb.ini fmp.exe
- デバッガで以下のコマンドを入力してターゲットと接続、ダウンロード後、プログラムをスタート
  (gdb) target remote localhost:[ポート番号]
  (gdb) load
- プログラムスタート
  (gdb) continue　（cだけでも可）
- 終了
  - デバッガで quit と入力
  - SkyEyeを起動したターミナルで Ctrl-C を押す

ポート番号はskyeye.confで指定した番号と同じにする

# GDB主要コマンド一覧（参考情報）

| | |
|---|---|
| **continue** | 停止しているプログラムを続行する |
| **next** | 次の1行を実行。関数呼び出しを1行として扱う |
| **step** | 次の1行を実行。呼び出された関数の行も1行として扱う |
| **list** | ソースファイルのリストを10行分表示 |
| **break linenum** | 指定行linenumにブレイクポイントを設定<br>指定行を実行する直前で停止する |
| **delete breakp** | 番号breakpのブレークポイントを削除 |
| **print expr** | exprの値を表示 |
| **info locals** | 現在の関数内の局所変数の名前と値を全て表示す |
| **help** | gdb全体のヘルプを表示 |

頭文字だけでも実行可能

# アジェンダ

ステップ1. 開発環境について
ステップ2. ASPカーネルのサンプルプログラムの解説
ステップ3. ASP用サンプルプログラムを1プロセッサで動作させる
ステップ4. ASP用サンプルプログラムを2プロセッサで動作させる
ステップ5. タスク間の排他制御の方法
ステップ6. タスクとハンドラ間の排他制御の方法
ステップ7. 起動時タスクマイグレーションAPI(mact_tsk)の使い方
ステップ8. タスクマイグレーションAPI(mig_tsk)の使い方

# サンプルプログラム (sample1)

- カーネルの基本的な動作を確認するためのプログラム
  - TOPPERS/ASPの配布パッケージに含まれる

- 構成ファイル
  - sample1.h　：ヘッダーファイル
  - sample1.c　：Cソースファイル
  - sample1.cfg　：コンフィギュレーションファイル
  - skyeye.conf　：skyeye設定ファイル

# ASPカーネルのsample1の実行手順

- Cygwinを立ち上げる
  - homeディレクトリがカレントディレクトリ
- ディレクトリを移動する

  $ cd asp_sample
- 設定ファイル（skyeye.conf）を引数にして実行

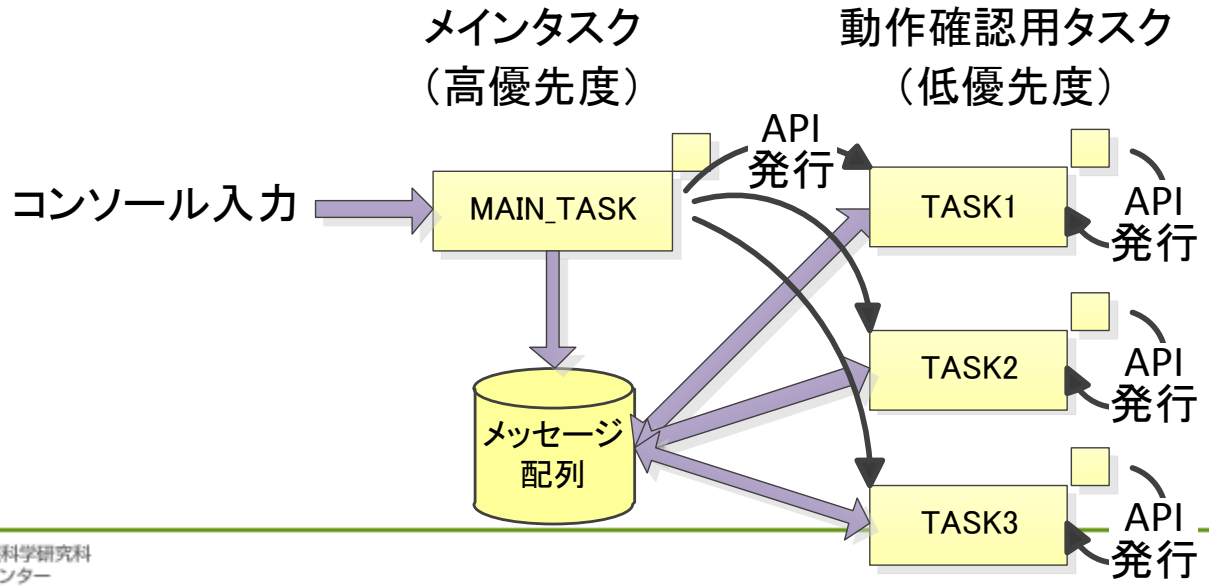
  $ skyeye.exe –e asp.exe –c skyeye.conf

```
~/asp_sample

students@PC-20 ~
$ cd asp_sample/

students@PC-20 ~/asp_sample
$ skyeye.exe -e asp.exe -c skyeye.conf
```

```
~/asp_sample
start addr is set to 0x0000003c by exec fi

TOPPERS/ASP Kernel Release 1.6.0 for AT91S
Copyright (C) 2000-2003 by Embedded and Re
                            Toyohashi Univ
Copyright (C) 2004-2010 by Embedded and Re
            Graduate School of Information

System logging task is started on port 1.
Sample program starts (exinf = 0).
task1 is running (001).   |
task1 is running (002).   |
task1 is running (003).   |
task1 is running (004).   |
task1 is running (005).   |
task1 is running (006).   |
task1 is running (007).   |
task1 is running (008).   |
task1 is running (009).   |
```

# タスク構成

- メインタスク(MAIN_TASK)（1個）
  - 実行可能状態で起動
  - コンソールからの入力を受け付ける
  - 入力に応じてAPI発行などを行う
- 動作確認用タスク(TASK1, TASK2, TASK3)（3個）
  - 休止状態で起動（メインタスクから起動）
  - 一定回数空ループを実行する度に、タスクが実行中であることをあらわすメッセージを表示する

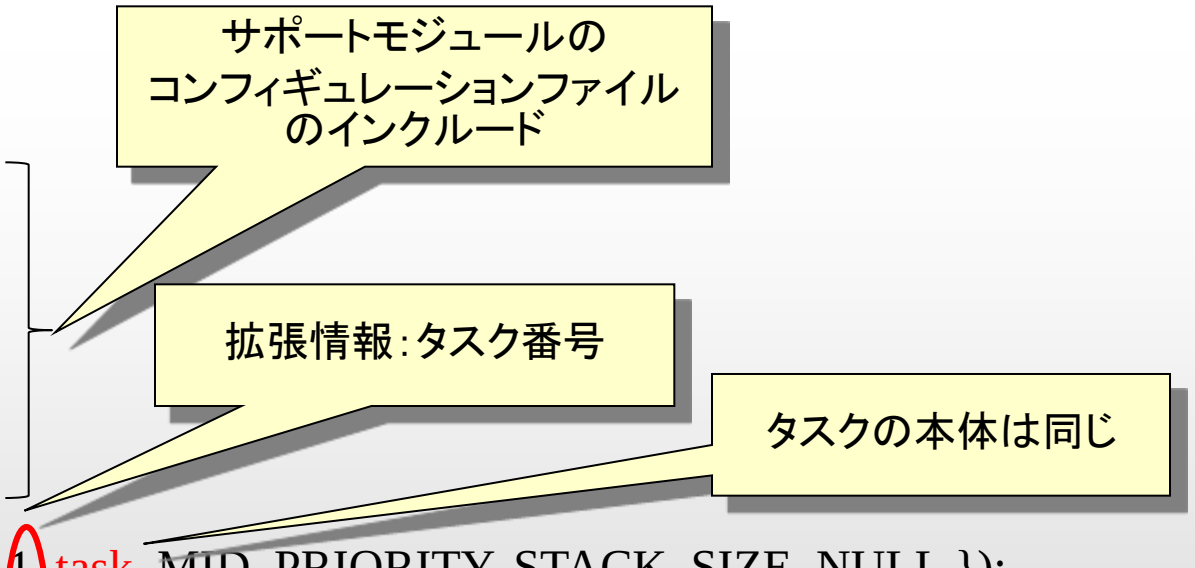


# コンフィギュレーションファイル（sample1.cfg）

- 静的APIを記述
- メインタスク
  - main_tsk関数から開始
  - 実行可能状態で起動
- 動作確認用タスク
  - task関数から開始
  - 休止状態で起動

sample1.cfg

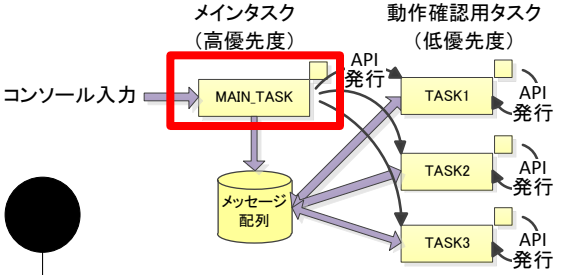
```
#include "sample1.h”
INCLUDE("target_timer.cfg");
INCLUDE("syssvc/syslog.cfg");
INCLUDE("syssvc/banner.cfg");
INCLUDE("syssvc/serial.cfg");
INCLUDE("syssvc/logtask.cfg");

CRE_TSK(TASK1, { TA_NULL, 1, task, MID_PRIORITY, STACK_SIZE, NULL });
CRE_TSK(TASK2, { TA_NULL, 2, task, MID_PRIORITY, STACK_SIZE, NULL });
CRE_TSK(TASK3, { TA_NULL, 3, task, MID_PRIORITY, STACK_SIZE, NULL });
CRE_TSK(MAIN_TASK, { TA_ACT, 0, main_task, MAIN_PRIORITY, STACK_SIZE, NULL });
```

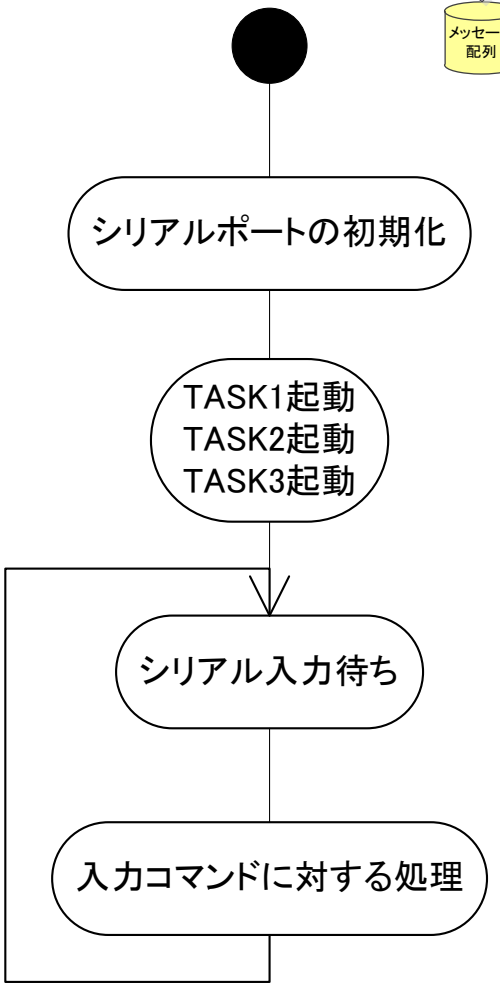


# メインタスクの処理(1/2)

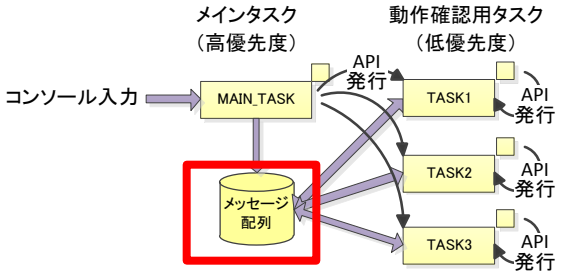
- タスクの中で最高優先度
- 実行可能状態で起動

- 処理内容
  - シリアルポートの初期化
  - TASK1，TASK2，TASK3を起動
  - ループ処理
    - シリアル入力待ち
    - 入力コマンドに対する処理



# メインタスクの処理(2/2)

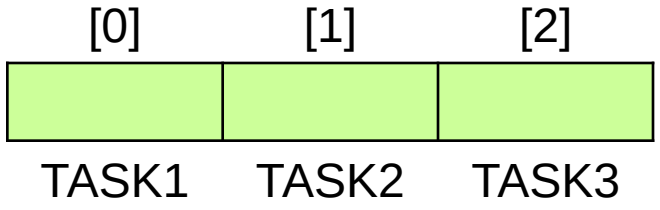



- 入力コマンドに対する処理
  - メインタスクが発行するAPIの場合は発行

    <例>  act_tsk(tskid), chg_pri(tskid,pri)
  - 動作確認用タスク自身が発行するAPIの場合
    - メッセージ配列の該当場所へ，該当メッセージを格納

<u>メッセージ配列</u>

- メインタスクと並列実行されるタスクの間は，グローバル配列でメッセージを伝える
  - char_t　message[3]
  - INDEX:タスク番号-1



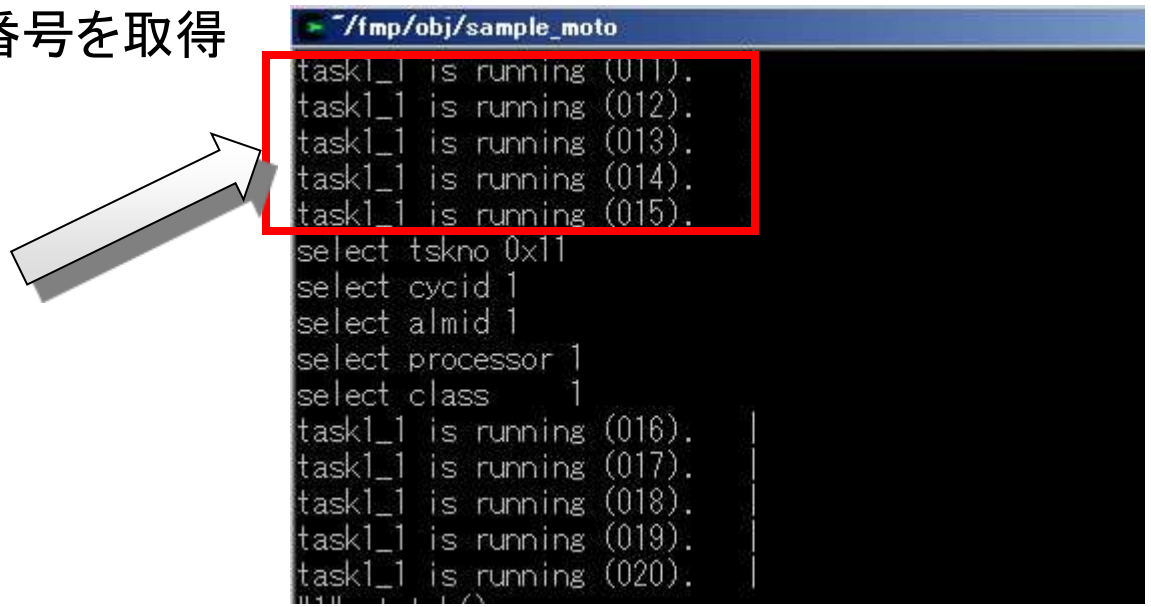

<例>

TASK1に対してext_tsk()　⇒message[0]=“e”

TASK2に対してslp_tsk()　⇒message[1]=“s”

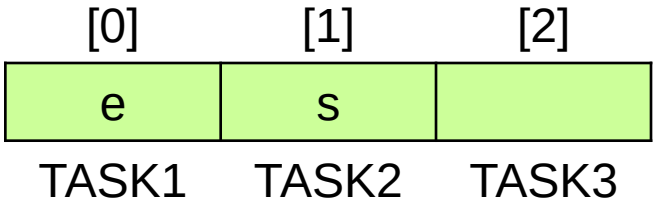
# 動作確認用タスクの処理

- 拡張情報から，自分のタスク番号を取得




- ループ処理
  - 実行ログを出力

- メッセージ配列を確認
  - 自分自身宛のメッセージがあれば受け取り，メッセージ配列をクリアする
  - APIを発行
    - ext_tsk(),slp_tsk()など



# サンプルプログラムの実行 ：コマンド一覧

- '1','2','3' : 以降のコマンドは TASK1/2/3 に対して行う.
- 'a' : タスクを act_tsk(ID tskid) により起動する.
- 'A' : タスクに対する起動要求を can_act(ID tskid) によりキャンセルする.
- 'e' : タスクに ext_tsk() を呼び出させ，終了させる.
- 't' : タスクを ter_tsk(ID tskid) により強制終了する.
- '>' : タスクの優先度を chg_pri(ID tskid, PRI tskpri)によりHIGH_PRIORITY にする
- '=' : タスクの優先度を chg_pri(ID tskid, PRI tskpri)によりMID_PRIORITY にする
- '<' : タスクの優先度を chg_pri(ID tskid, PRI tskpri)によりLOW_PRIORITY にする
- 'G' : タスクの優先度を get_pri(ID tskid, PRI *p_tskpri) で読み出す.
- 's' : タスクに slp_tsk() を呼び出させ，起床待ちにさせる.
- 'S' : タスクに tslp_tsk(TMO tmout) を呼び出させ，起床待ちにさせる.
- 'w' : タスクを wup_tsk(ID tskid) により起床する.
- 'W' : タスクに対する起床要求を can_wup(ID tskid) によりキャンセルする.
- 'c' : sta_cyc(ID cycid)により，周期ハンドラを動作させる.
- 'C' : stp_cyc(ID cycid)により，周期ハンドラを停止させる.
- 'b' : sta_alm(ID almid, RELTIM almtim)により，アラームハンドラを動作開始させる
- 'B' : stp_alm(ID almid)により，アラームハンドラを動作停止させる.

# サンプルプログラムの実行：sample1のコマンド例

- サンプルプログラムを動かしコマンドを入力する

  <例>

  - ‘s’コマンド(slp_tsk)でタスク1を起床待ちにすると、task2が実行
  - ‘2’コマンド で対象をタスク2に変え、‘s’コマンド(slp_tsk)で起床待ちにすると、task3が実行
  - ‘3’コマンド で対象をタスク3に変え、‘s’コマンド(slp_tsk)で起床待ちにすると、実行状態のタスクはいなくなる.
  - ‘2’コマンド で対象をタスク2に変え、‘w’コマンド(wup_tsk)で実行を再開させると、task2が実行

# アジェンダ

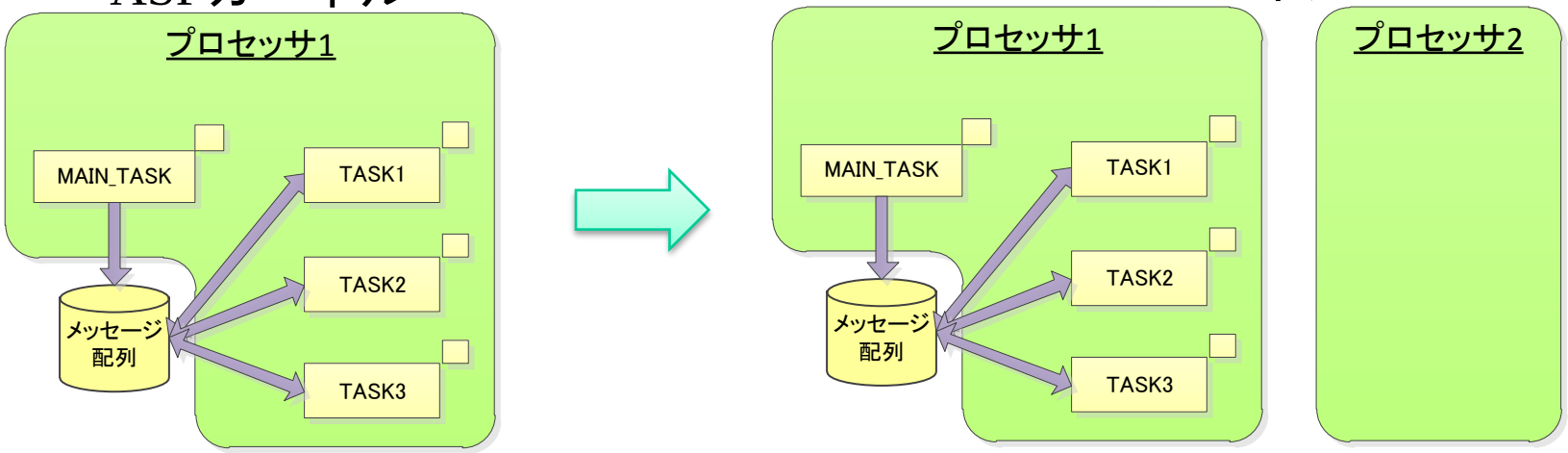
ステップ1. 開発環境について
ステップ2. ASPカーネルのサンプルプログラムの解説
ステップ3. ASP用サンプルプログラムを1プロセッサで動作させる
ステップ4. ASP用サンプルプログラムを2プロセッサで動作させる
ステップ5. タスク間の排他制御の方法
ステップ6. タスクとハンドラ間の排他制御の方法
ステップ7. 起動時タスクマイグレーションAPI(mact_tsk)の使い方
ステップ8. タスクマイグレーションAPI(mig_tsk)の使い方

# 【ステップ3】:概要

ASPカーネル用サンプルプログラムを1プロセッサで実行する

- 学習内容
  - シミュレータを用いたFMPカーネルの実行手順の確認
  - コンフィギュレーションについて，ASPカーネルと異なる部分の理解

ASPカーネル　　FMPカーネル



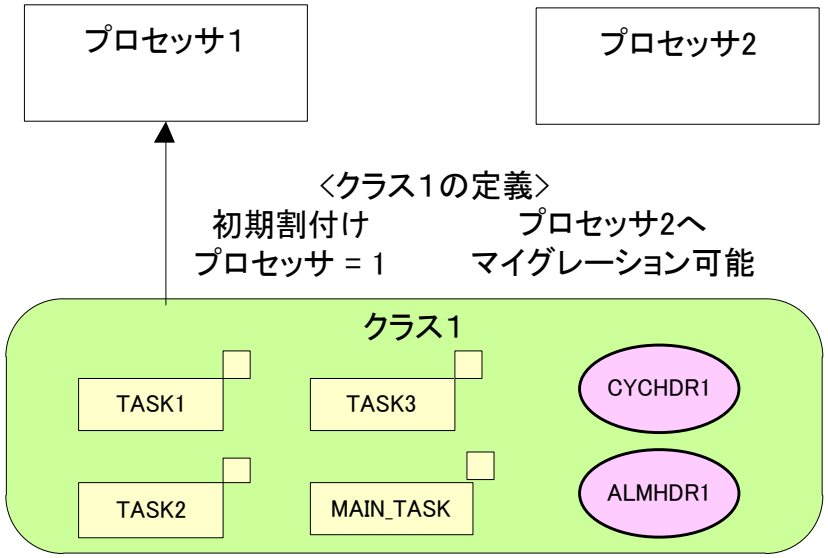

FMPカーネルでは，プロセッサという概念が増えるため
コンフィギュレーションファイルでプロセッサ（クラス）の
指定が必要

# 【ステップ3】:sample1.cfgの変更点 (1/2)

```
INCLUDE("target_ipi.cfg");
CLASS(TCL_1){
   CRE_TSK(TASK1, { TA_NULL, 1, task, MID_PRIORITY, STACK_SIZE, NULL });
   CRE_TSK(TASK2, { TA_NULL, 2, task, MID_PRIORITY, STACK_SIZE, NULL });
   CRE_TSK(TASK3, { TA_NULL, 3, task, MID_PRIORITY, STACK_SIZE, NULL });
   CRE_TSK(MAIN_TASK, { TA_ACT, 0, main_task, MAIN_PRIORITY, STACK_SIZE, NULL });
   CRE_CYC(CYCHDR1, { TA_NULL, 0, cyclic_handler, CYC_TIME, 0 });
   CRE_ALM(ALMHDR1, { TA_NULL, 0, alarm_handler });
}
```

クラス１（プロセッサ1で動作するクラス）

- オブジェクトが属するクラス
  - コンフィギュレーションファイルで指定
- クラスの囲みにオブジェクト生成の静的APIを記述する
- オブジェクトはいずれかのクラスに属する必要がある



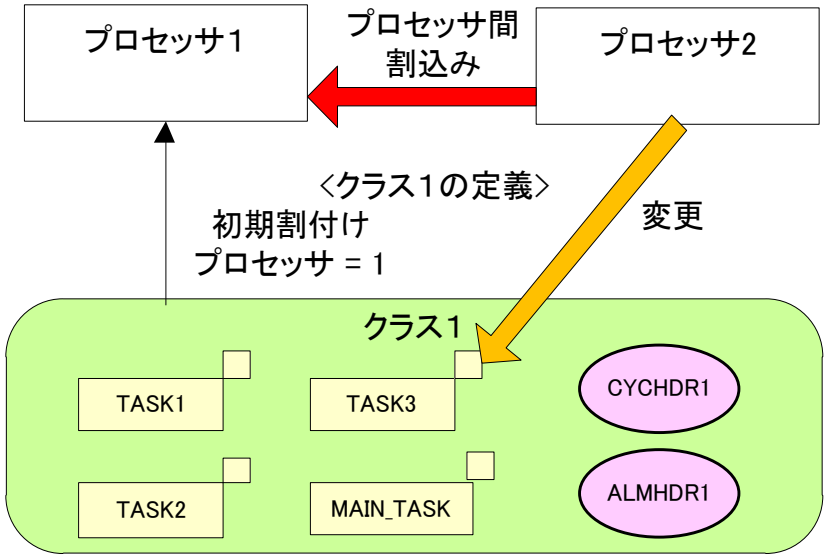
# 【ステップ3】:sample1.cfgの変更点 (2/2)

インクルードファイルの追加

```
INCLUDE("target_ipi.cfg");
CLASS(TCL_1){
   CRE_TSK(TASK1, { TA_NULL, 1, task, MID_PRIORITY, STACK_SIZE, NULL });
   CRE_TSK(TASK2, { TA_NULL, 2, task, MID_PRIORITY, STACK_SIZE, NULL });
   CRE_TSK(TASK3, { TA_NULL, 3, task, MID_PRIORITY, STACK_SIZE, NULL });
   CRE_TSK(MAIN_TASK, { TA_ACT, 0, main_task, MAIN_PRIORITY, STACK_SIZE, NULL });
   CRE_CYC(CYCHDR1, { TA_NULL, 0, cyclic_handler, CYC_TIME, 0 });
   CRE_ALM(ALMHDR1, { TA_NULL, 0, alarm_handler });
}
```

- 別のプロセッサに割り付けられたオブジェクトが変更されたことを通知する必要がある

  ⇨プロセッサ間割込みで通知する




!!追加しなくてもコンパイルエラーにならないので注意!!

# 【ステップ3】:実行手順

- Cygwinをプロセッサの数(今回は2つ)起動

- ディレクトリを移動

  $ cd fmp/obj/step3/

- コンパイル（どちらか一方のCygwinで）

  $ make depend

  $ make

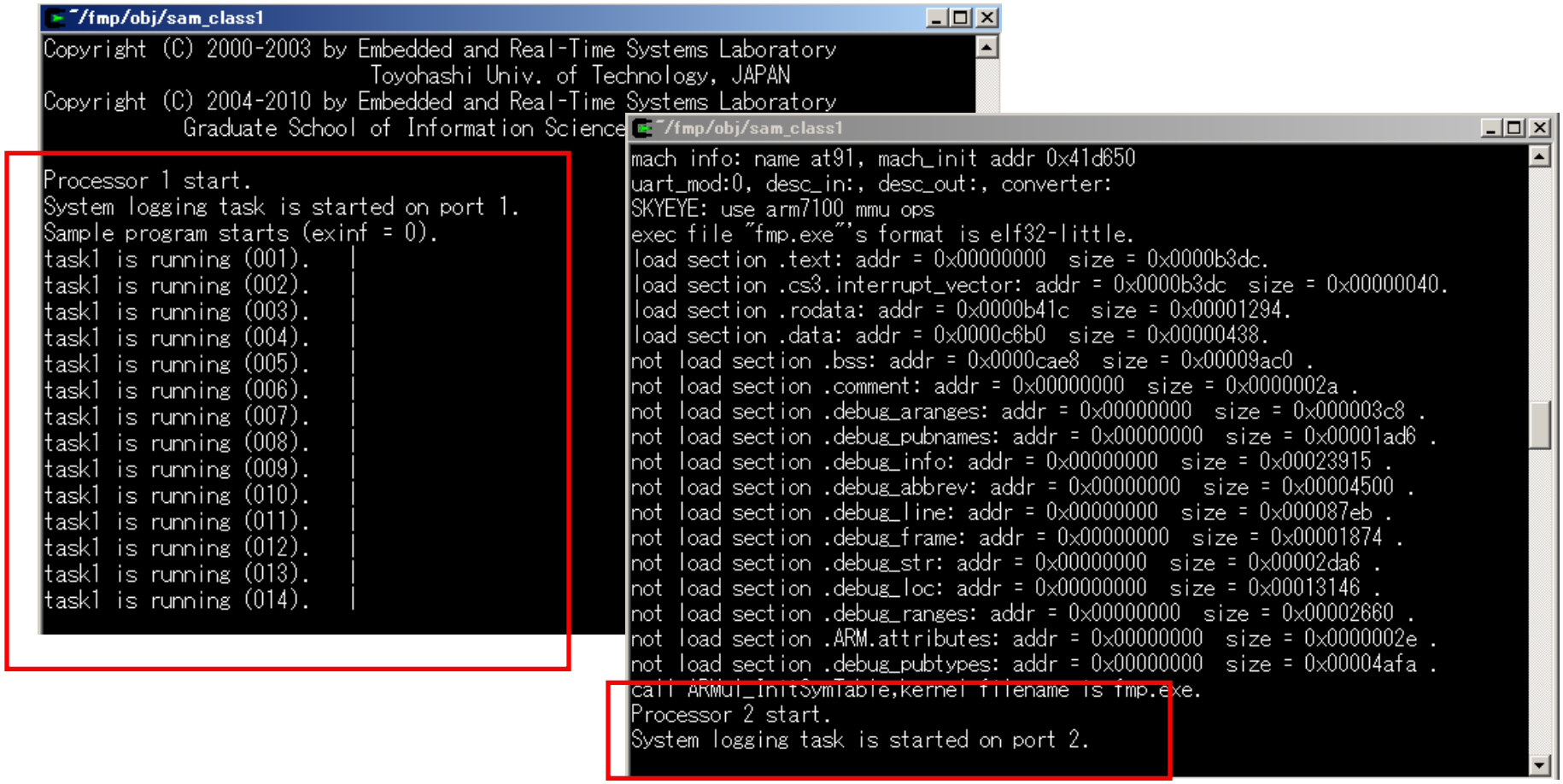
- それぞれskyeyeを実行

  $ skyeye.exe -e fmp.exe -c ../skyeye_pe1.conf

  $ skyeye.exe -e fmp.exe -c ../skyeye_pe2.conf

# 【ステップ3】:実行，動作確認

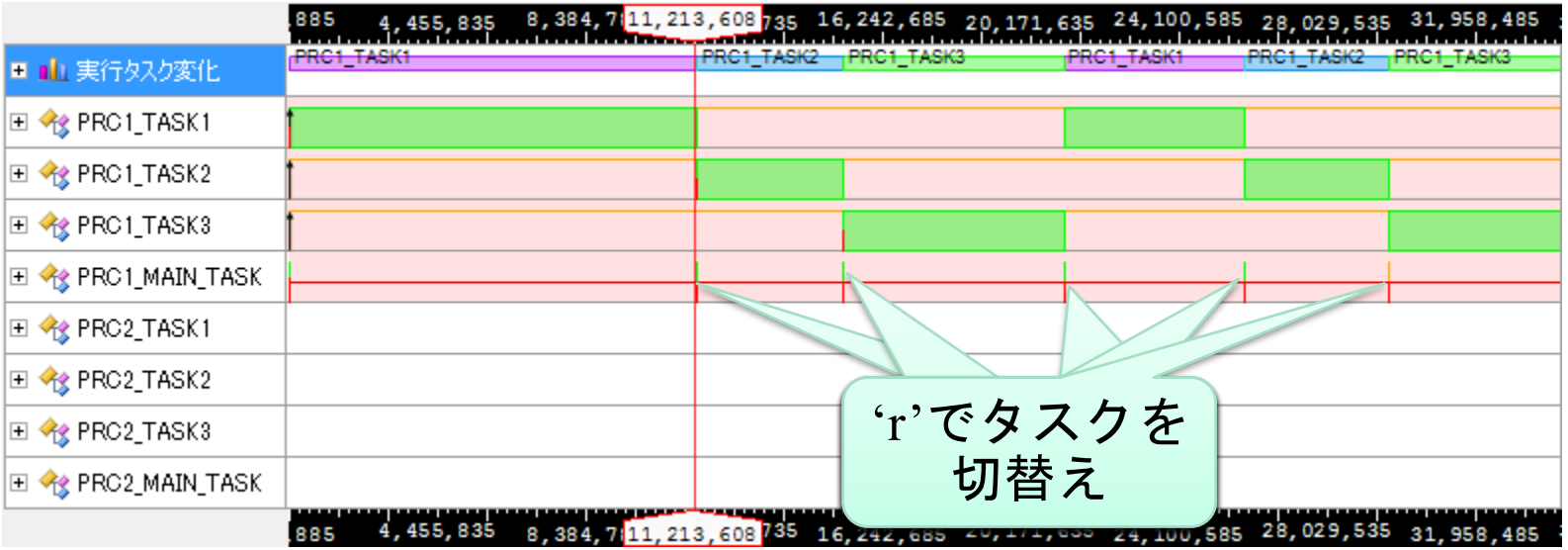
- プロセッサ1のみ動作確認用タスクが実行
- プロセッサ1からコマンドを入力し動作確認




ソースコード変更なしでASPから移行が可能!!

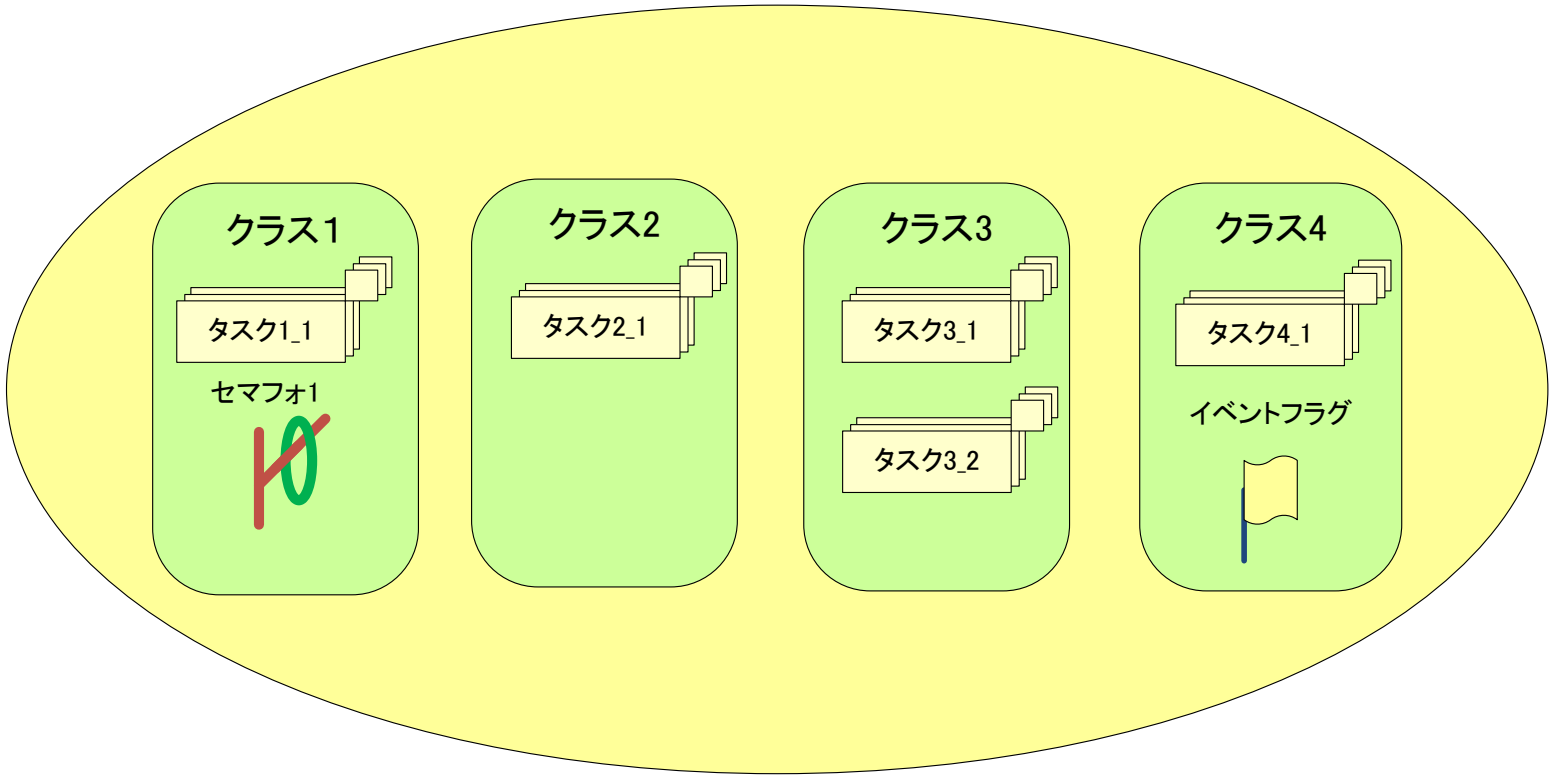
# 【ステップ3】: TLVでの可視化

- ‘Q’でサンプルプログラムを終了させる
  - ログファイル‘kernel.log’を出力するので，正常終了するまで待つ
- ‘kernel.log’と‘kernel.res’をTLVにドラッグアンドドロップする
  - TLVがログを解析した後，表示される



# クラスとは

- マルチプロセッサに対応するために用いるカーネルオブジェクトの集合
- クラスごとに特徴を定義することができる



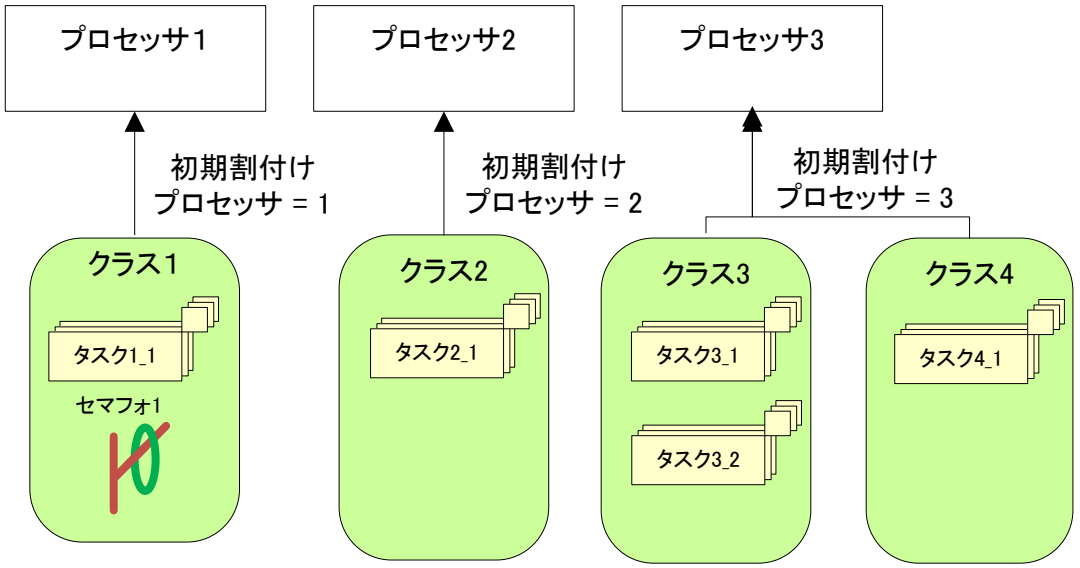
# クラスの定義内容

- 初期割付けプロセッサ
- 割付け可能プロセッサ
- コントロールブロックの配置場所
  - タスク管理ブロックの配置場所
  - セマフォ管理ブロックの配置場所
  - イベントフラグ管理ブロックの配置場所
    ・・・など
- コントロールブロック以外のメモリ領域の配置場所
  - タスクスタックの配置場所
  - データキュー管理領域の配置場所
    ・・・など
- オブジェクトが使用するオブジェクトロック

アーキテクチャに合わせて
ターゲット依存部開発者が定義する

# クラスの特徴

- システムには複数のクラスが定義されている.
- オブジェクトはいずれかのクラスに属する必要がある.
- オブジェクトが属するクラスは静的に指定する(動的に変更できない)
- １つのプロセッサを，複数のクラスが初期割り付けプロセッサに指定することができる
  - 同じプロセッサに特徴の違う2つのクラスを所属させ，使い分ける



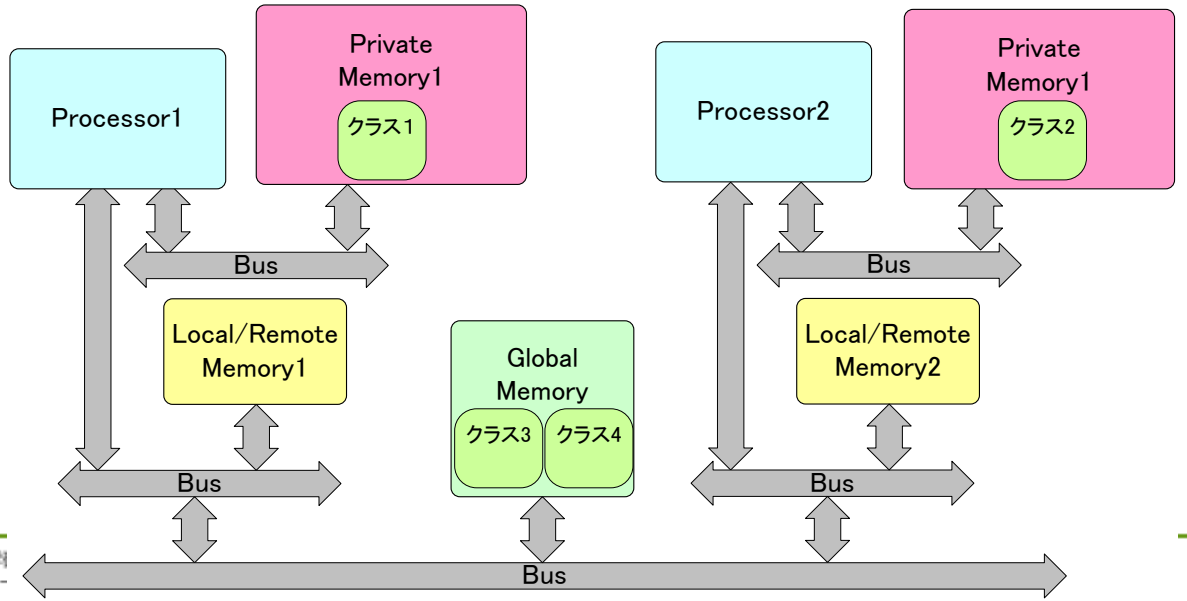

プロセッサ及びクラスは1から連続するID番号で管理される

プロセッサ　≠　クラス

# クラスの使用方法

- リアルタイム性が求めらる処理を行うクラスをプライベートメモリに
- 他プロセッサからのアクセスが必要なクラスはグローバルメモリに

| | Processor1に属する | Processor2に属する |
|---|---|---|
| リアルタイム性求められる | クラス1 | クラス2 |
| 他プロセッサからのアクセスあり | クラス3 | クラス4 |



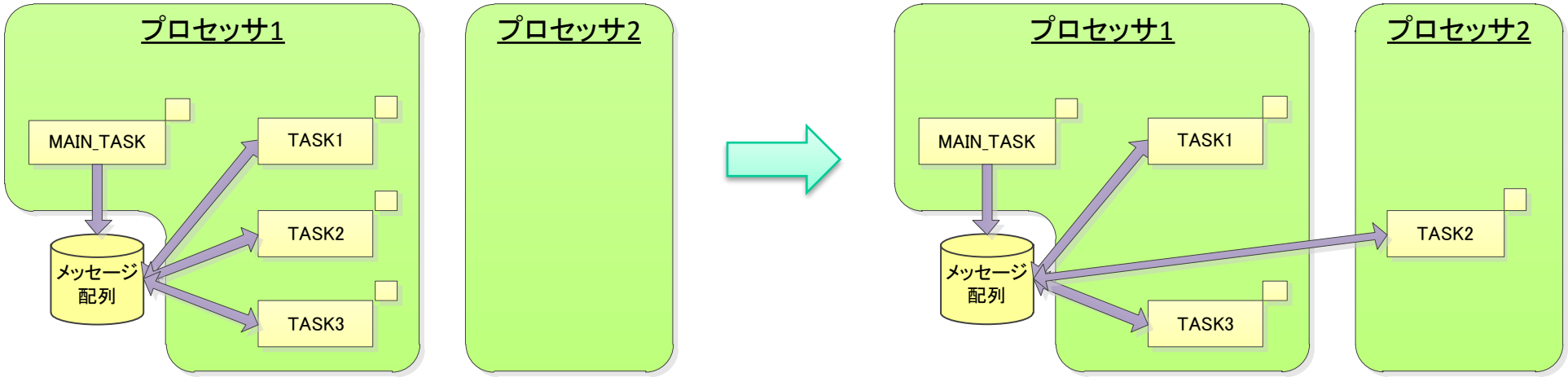
# アジェンダ

ステップ1. 開発環境について
ステップ2. ASPカーネルのサンプルプログラムの解説
ステップ3. ASP用サンプルプログラムを1プロセッサで動作させる
ステップ4. ASP用サンプルプログラムを2プロセッサで動作させる
ステップ5. タスク間の排他制御の方法
ステップ6. タスクとハンドラ間の排他制御の方法
ステップ7. 起動時タスクマイグレーションAPI(mact_tsk)の使い方
ステップ8. タスクマイグレーションAPI(mig_tsk)の使い方

# 【ステップ4】:概要

TASK2をクラス2に属するオブジェクトとする

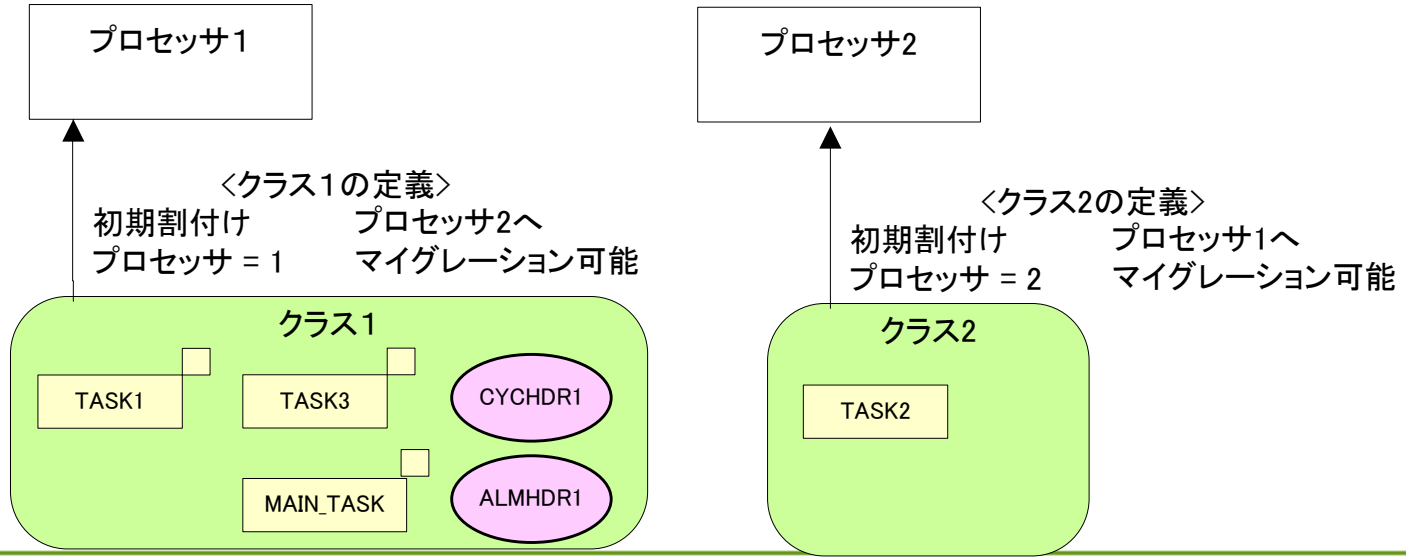
- 学習内容
  - TASK2の所属変更
  - プロセッサ間の排他制御の必要性および利用方法について理解する



# 【ステップ4】: sample1.cfg

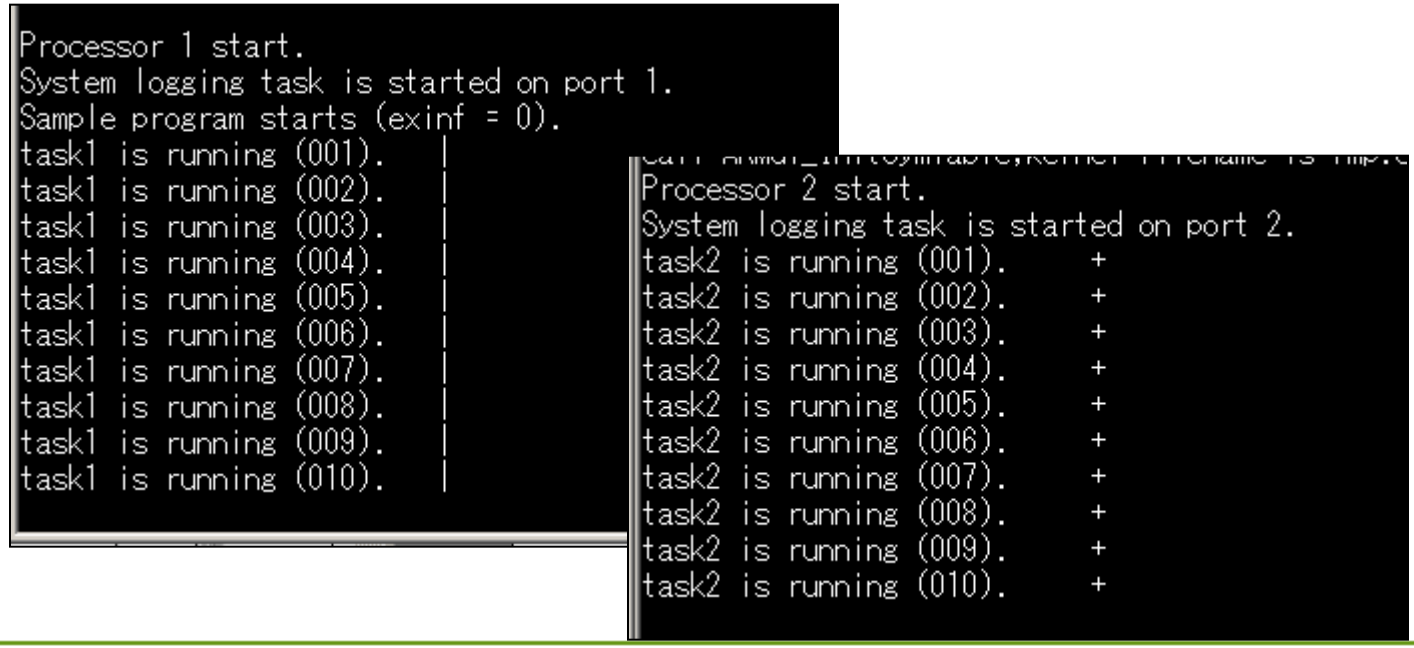
タスク2を移動

```
CLASS(TCL_1){
   CRE_TSK(TASK1, { TA_NULL, 1, task, MID_PRIORITY, STACK_SIZE, NULL });
   CRE_TSK(TASK3, { TA_NULL, 3, task, MID_PRIORITY, STACK_SIZE, NULL });
   CRE_TSK(MAIN_TASK1, { TA_ACT, 0, main_task, MAIN_PRIORITY, STACK_SIZE, NULL });
   CRE_CYC(CYCHDR1, { TA_NULL, 0, cyclic_handler, CYC_TIME, 0 });
   CRE_ALM(ALMHDR1, { TA_NULL, 0, alarm_handler });
}
CLASS(TCL_2){
   CRE_TSK(TASK2, { TA_NULL, 2, task, MID_PRIORITY, STACK_SIZE, NULL });
}
```



# 【ステップ4】:実行，動作確認

- プロセッサ1，2とも動作確認用タスクが実行
  - プロセッサ1 : TASK1/TASK3
  - プロセッサ2 : TASK2
- プロセッサ1からコマンドを入力し，動作確認
- プロセッサ2では，コマンド入力を受け付けない
  - メインタスクがいないため．



# ステップ3と4 : まとめ

- ASPカーネル用サンプルアプリケーションをFMPカーネル上で動作させた
  - クラスの追加
  - クラス指定の変更による異なるプロセッサへの割り付け
  - インクルードファイルの追加

- 異なるプロセッサで同時にタスクが動作するので、排他制御をしないと期待通りに動作しないことがある
  - 例 : タスク間で共有する変数を操作

# アジェンダ

ステップ1. 開発環境について
ステップ2. ASPカーネルのサンプルプログラムの解説
ステップ3. ASP用サンプルプログラムを1プロセッサで動作させる
ステップ4. ASP用サンプルプログラムを2プロセッサで動作させる
ステップ5. タスク間の排他制御の方法
ステップ6. タスクとハンドラ間の排他制御の方法
ステップ7. 起動時タスクマイグレーションAPI(mact_tsk)の使い方
ステップ8. タスクマイグレーションAPI(mig_tsk)の使い方

# 排他制御に関する留意点

シングルプロセッサで用いている排他制御手法は，
マルチプロセッサでは機能しない場合がある

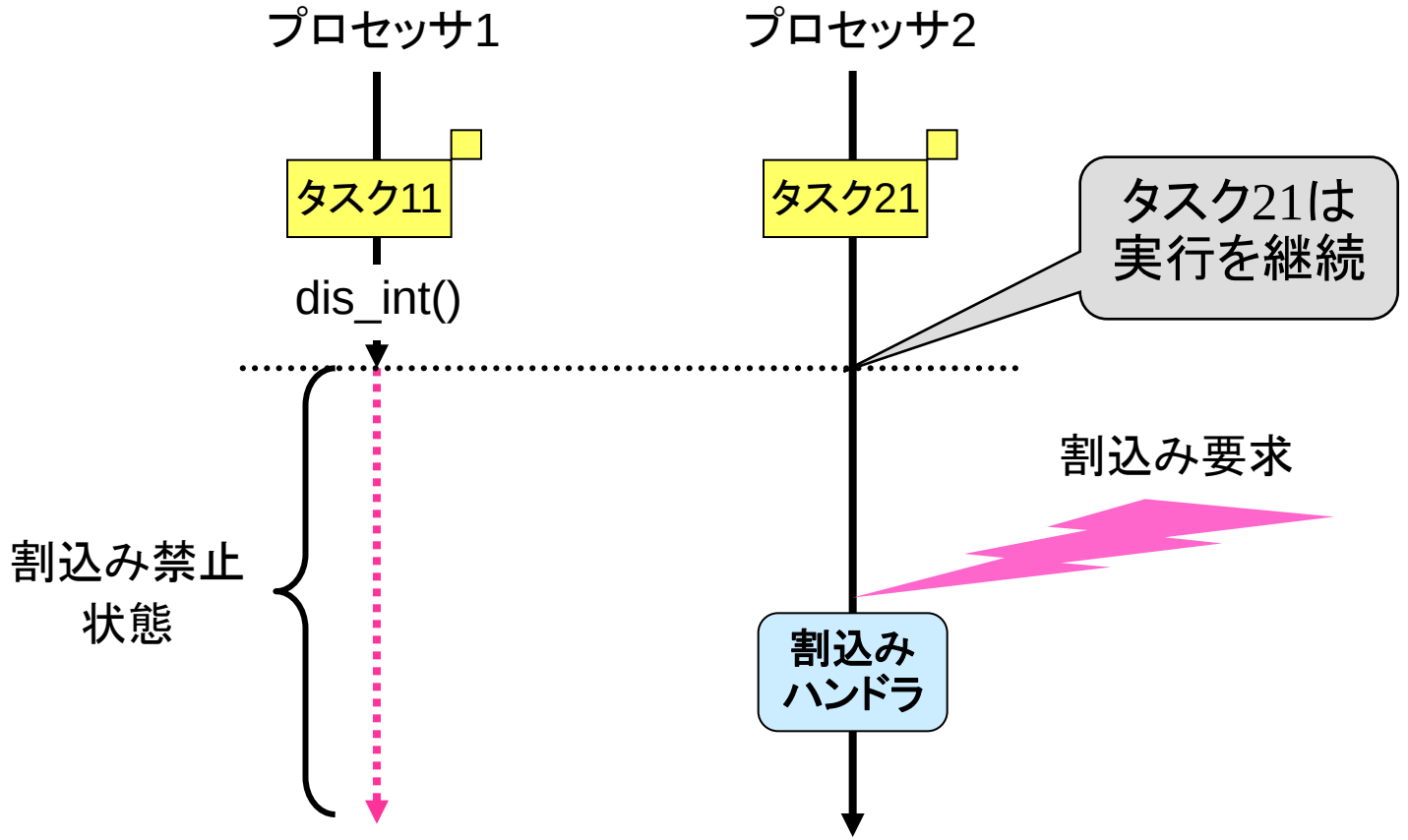
➡該当部分の書き換えが必要となる

ITRON（シングルプロセッサ）の排他関連機能

- 割り込み禁止状態
  - タスク間，割り込みハンドラ間，タスク-割込みハンドラ間の排他制御
  - 実行オーバヘッドが小さく，手軽なので多用される
- ディスパッチ禁止状態
  - タスク間の排他制御
- セマフォ・ミューテックス
  - タスク間の排他制御

# マルチプロセッサ環境下での割込み禁止

## 割込みの禁止・許可はプロセッサ毎に独立に管理されている




プロセッサ1が割込み禁止状態になったとしても，プロセッサ2はその影響を受けず，
タスクの実行は継続され，割込みの受付と割込みハンドラの実行が可能である

# マルチプロセッサ環境下での排他制御

割込み禁止ではプロセッサ間の排他制御は実現できない

シングルプロセッサ用のアプリケーション

- 割込み禁止により排他制御を行う場合がある

⇒マルチプロセッサ環境では，排他する対象により変更する必要がある

- タスク優先度を用いて排他制御を実現している箇所も同様
  - 「最高優先度のタスクが実行中に，他のタスクは動作しない」という考えはマルチプロセッサ環境では成り立たない

タスク間の排他制御

⇒セマフォやミューテックスを用いた排他制御に書き換える

- タスクの挙動が変化するので注意

# プロセッサ間排他制御の実現

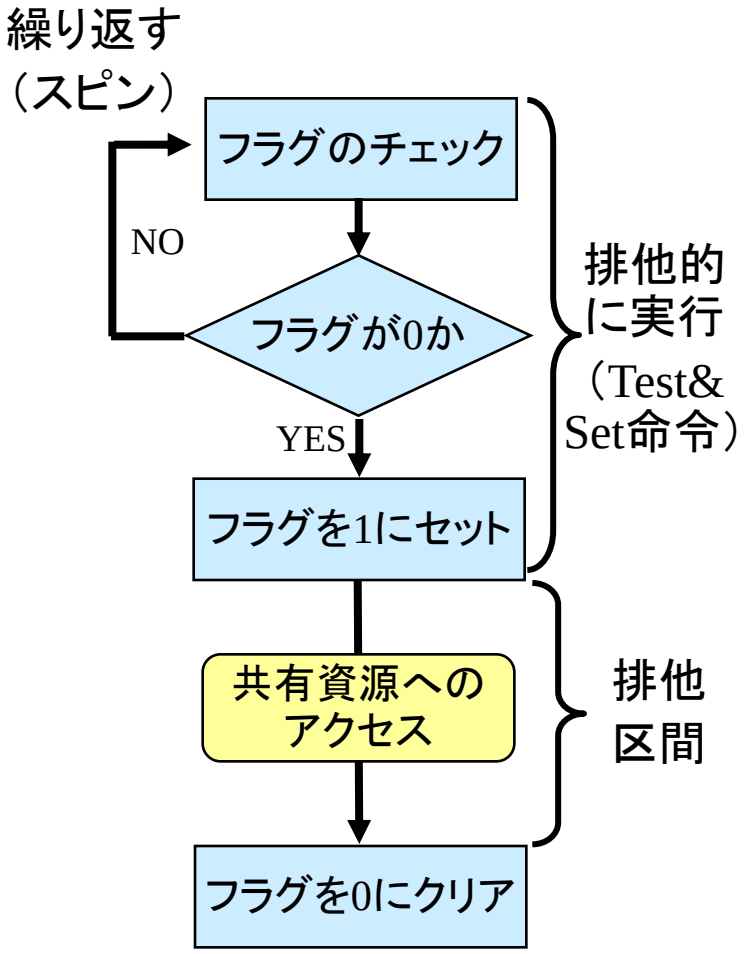
## プロセッサ間排他制御にはスピンロック機能を使用する

スピンロック機能

- プロセッサ間で共有するフラグを持つ
- 排他区間に入る前にフラグをチェックする
  - ‘0’なら‘1’として，排他区間に入る
  - ‘1’ならフラグのチェックを続ける
- 排他区間が終われば フラグを0とする

スピンロック取得時は割込みを禁止

- 割込みハンドラとのデッドロックを避けるため
- スピンロック取得状態の長さを短くするため



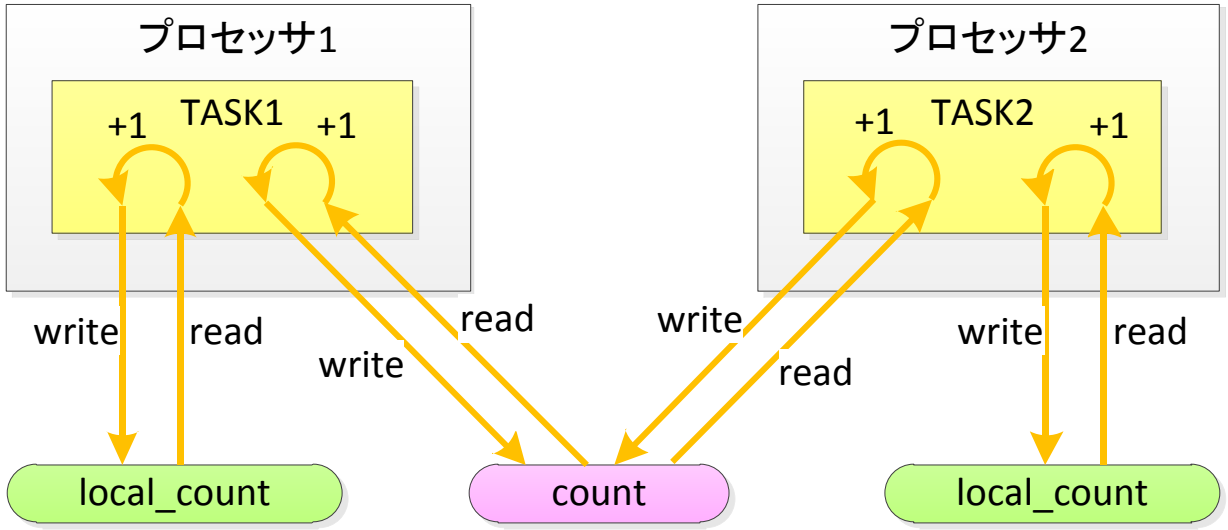
# 【ステップ5】: プロセッサ間排他制御(タスクvsタスク)

## タスクとタスク間のプロセッサ間排他制御

- 学習内容
  - プロセッサ間排他制御の方法
  - タスクとタスク間の排他制御をセマフォで実現

# 【ステップ5】:概要

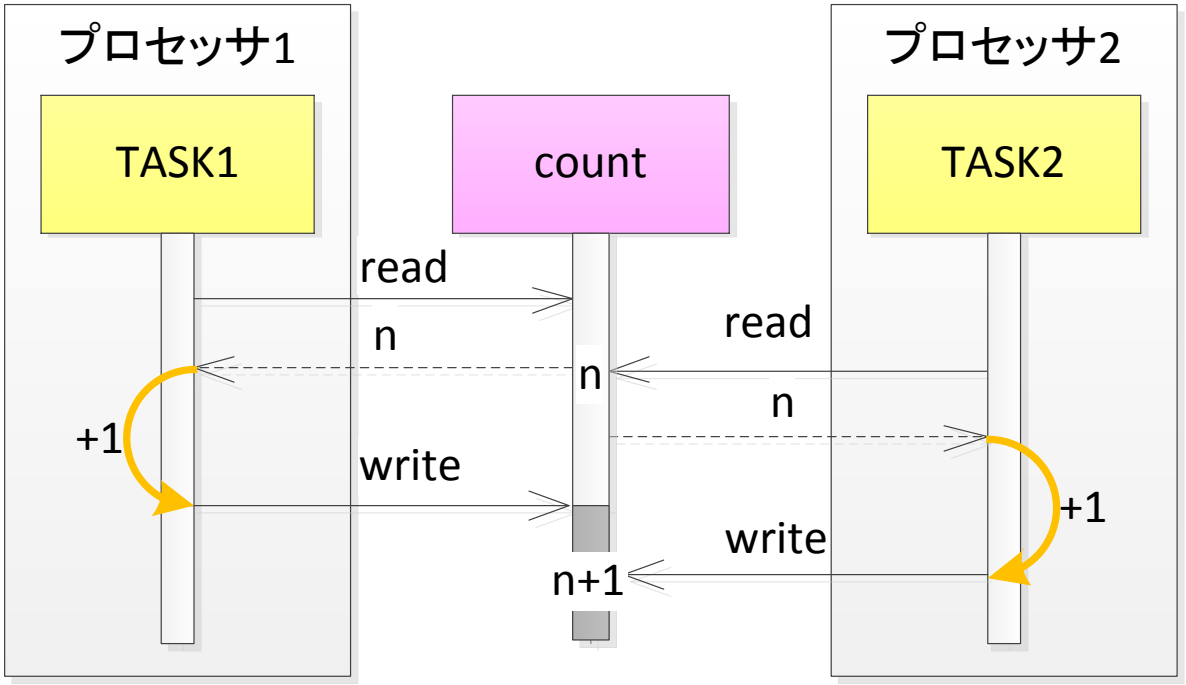
- プロセッサ1に所属するタスク1と，プロセッサ2に所属するタスク2が，共有変数countと，ローカル変数local_countをインクリメントする
- 共有変数の排他制御をセマフォで行う




countがlocal_countの2倍になることを期待

# 【ステップ5】: 排他制御の必要性

- 排他制御をしないと. . .
  - 2つのタスクがインクリメントしても+1にしかならない場合がある




共有変数countについて排他制御する必要がある

➡ セマフォで排他制御を行う

# 【ステップ5】: タスクの詳細

- タスク

```
int count = 0;
void
task(intptr_t exinf){
    int local_count = 0,i;

    for (i = 0 ; i < 100 ; i++) {
        local_count = busy_wait_inc(local_count);
        wai_sem(SEM1);
        count = busy_wait_inc(count);
        syslog(LOG_NOTICE, "Task %d : count = %d, local_count = %d", exinf, count, local_count);
        sig_sem(SEM1);
    }
}
```

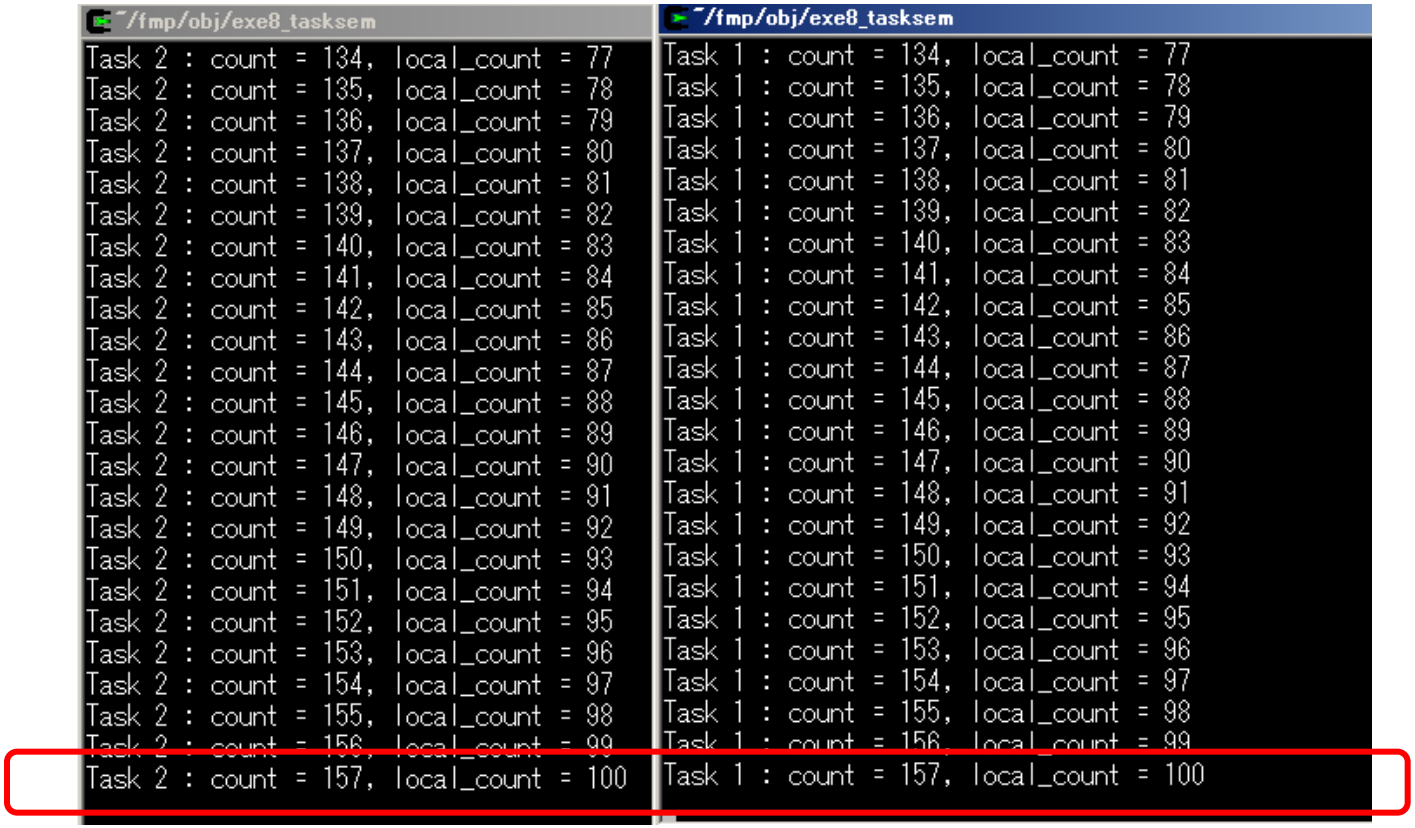
- busy_wait_inc関数
  - 引数をインクリメント
  - 時間待ち（実行状態のまま）
  - インクリメントした変数を戻り値とする

```
int
busy_wait_inc(int e){
    e++;
    sil_dly_nse(1000);
    return e;
}
```

微小時間待ち

# 【ステップ5】: 結果

- 排他制御しないと

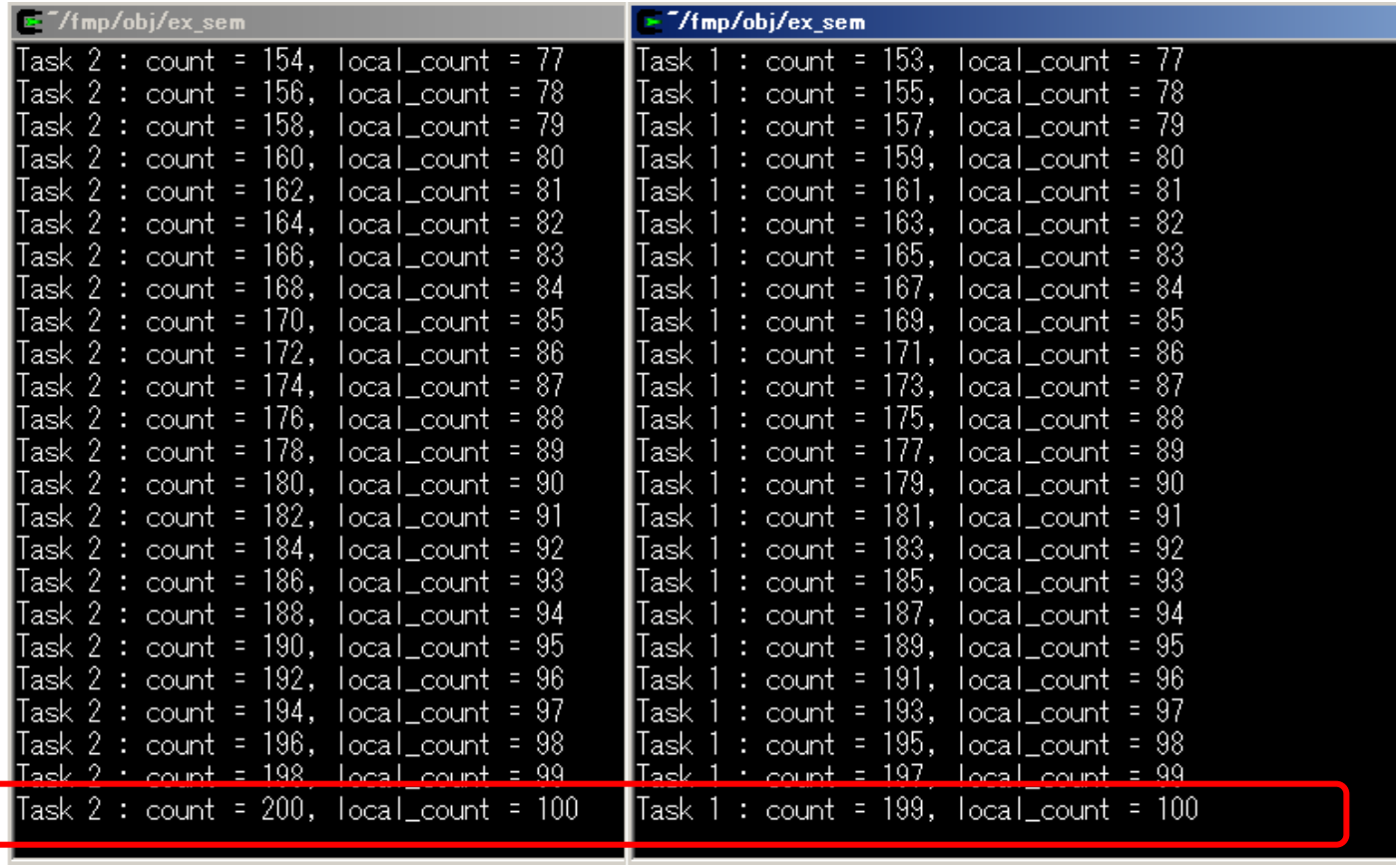



countはlocal_countの2倍になっていない

count（共有変数）のインクリメント途中に，他方のタスクでアクセスしている

# 【ステップ5】: 結果

- セマフォで排他制御後




countはlocal_countの2倍になっている

# アジェンダ

ステップ1. 開発環境について
ステップ2. ASPカーネルのサンプルプログラムの解説
ステップ3. ASP用サンプルプログラムを1プロセッサで動作させる
ステップ4. ASP用サンプルプログラムを2プロセッサで動作させる
ステップ5. タスク間の排他制御の方法
ステップ6. タスクとハンドラ間の排他制御の方法
ステップ7. 起動時タスクマイグレーションAPI(mact_tsk)の使い方
ステップ8. タスクマイグレーションAPI(mig_tsk)の使い方

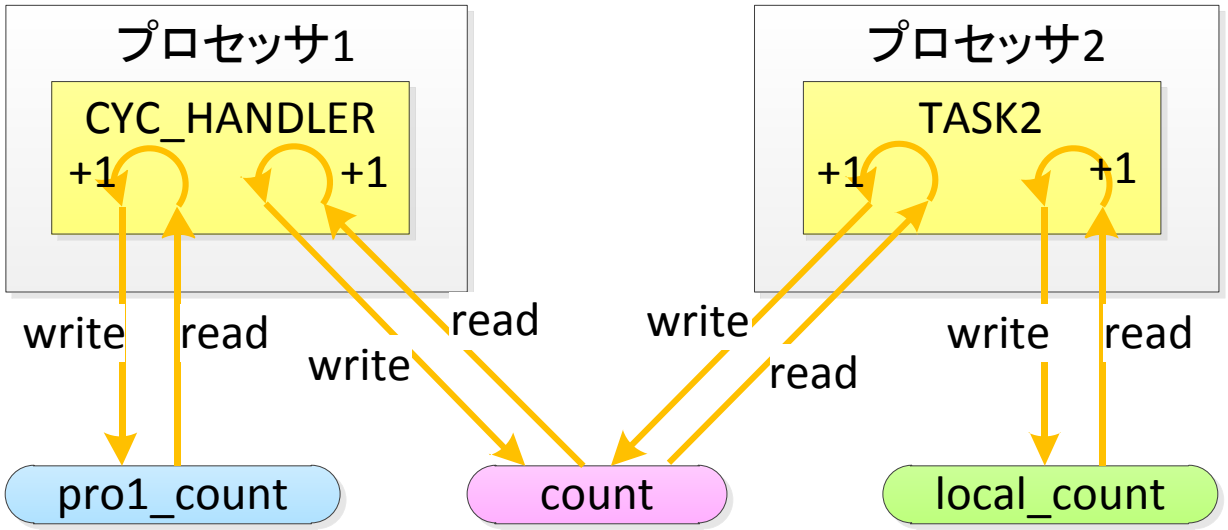
# 【ステップ6】: プロセッサ間排他制御(タスクvsハンドラ)

タスクとハンドラ間のプロセッサ間排他制御

- 学習内容
  - プロセッサ間排他制御の方法
  - タスクとハンドラ間の排他制御をスピンロックで実現

# 【ステップ6】: 概要

- プロセッサ1に所属する周期ハンドラ (CYC_HANDLER)
  - 共有変数countと，共有変数pro1_countをインクリメントする
- プロセッサ2に所属するタスク2 (TASK2)
  - 共有変数countと，ローカル変数local_countをインクリメントする
- 共有変数の排他制御を行う



# 【ステップ6】: 周期ハンドラとタスク

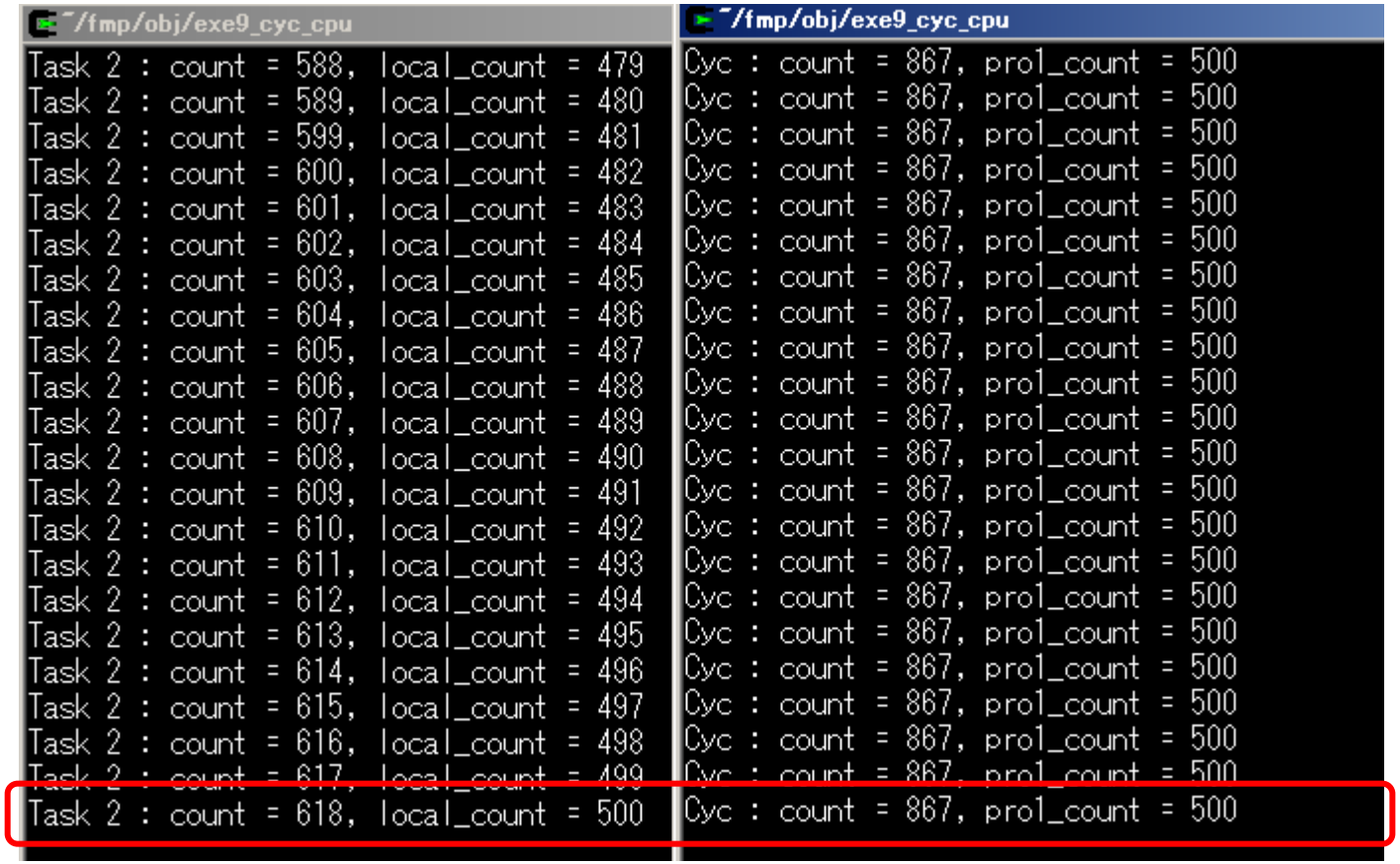
- 周期ハンドラ

```
void
cyc_handler(void){
    if(pro1_count < 500){
        pro1_count = busy_wait_inc(pro1_count);
        iloc_cpu();
        count = busy_wait_inc(count);
        iunl_cpu();
        syslog(LOG_NOTICE, "Cyc : count = %d, pro1_count = %d",  count, pro1_count);
    }
}
```

- プロセッサ2のタスク2ではCPUロックにより排他制御を試みる

```
void
task(intptr_t exinf){
    int local_count = 0,i;

    for (i = 0 ; i < 500 ; i++) {
        local_count = busy_wait_inc(local_count);
        loc_cpu();
        count = busy_wait_inc(count);
        unl_cpu();
        syslog(LOG_NOTICE, "Task %d : count = %d, local_count = %d", exinf, count, local_count);
    }
}
```

# 【ステップ6】: CPUロックで排他制御した例



```
Task 2 : count = 588, local_count = 479
Task 2 : count = 589, local_count = 480
Task 2 : count = 599, local_count = 481
Task 2 : count = 600, local_count = 482
Task 2 : count = 601, local_count = 483
Task 2 : count = 602, local_count = 484
Task 2 : count = 603, local_count = 485
Task 2 : count = 604, local_count = 486
Task 2 : count = 605, local_count = 487
Task 2 : count = 606, local_count = 488
Task 2 : count = 607, local_count = 489
Task 2 : count = 608, local_count = 490
Task 2 : count = 609, local_count = 491
Task 2 : count = 610, local_count = 492
Task 2 : count = 611, local_count = 493
Task 2 : count = 612, local_count = 494
Task 2 : count = 613, local_count = 495
Task 2 : count = 614, local_count = 496
Task 2 : count = 615, local_count = 497
Task 2 : count = 616, local_count = 498
Task 2 : count = 617, local_count = 499
Task 2 : count = 618, local_count = 500
```

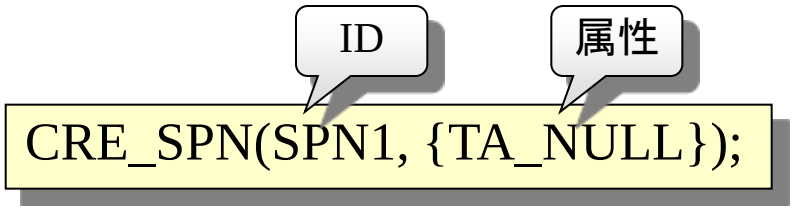
```
Cyc : count = 867, pro1_count = 500
Cyc : count = 867, pro1_count = 500
Cyc : count = 867, pro1_count = 500
Cyc : count = 867, pro1_count = 500
Cyc : count = 867, pro1_count = 500
Cyc : count = 867, pro1_count = 500
Cyc : count = 867, pro1_count = 500
Cyc : count = 867, pro1_count = 500
Cyc : count = 867, pro1_count = 500
Cyc : count = 867, pro1_count = 500
Cyc : count = 867, pro1_count = 500
Cyc : count = 867, pro1_count = 500
Cyc : count = 867, pro1_count = 500
Cyc : count = 867, pro1_count = 500
Cyc : count = 867, pro1_count = 500
Cyc : count = 867, pro1_count = 500
Cyc : count = 867, pro1_count = 500
Cyc : count = 867, pro1_count = 500
Cyc : count = 867, pro1_count = 500
Cyc : count = 867, pro1_count = 500
Cyc : count = 867, pro1_count = 500
Cyc : count = 867, pro1_count = 500
```

countはlocal_countの2倍になっていない

異なるプロセッサ間のタスクと周期ハンドラは，
CPUロックでは排他制御できない

# 【ステップ6】: スピンロックによる排他制御

- スピンロックでプロセッサ間排他制御を行う
  - スピンロックの生成




  - スピンロックの取得

```
loc_spn(SPN1)
```

  - スピンロックの返却

```
unl_spn(SPN1)
```

# 【ステップ6】: 周期ハンドラとタスク

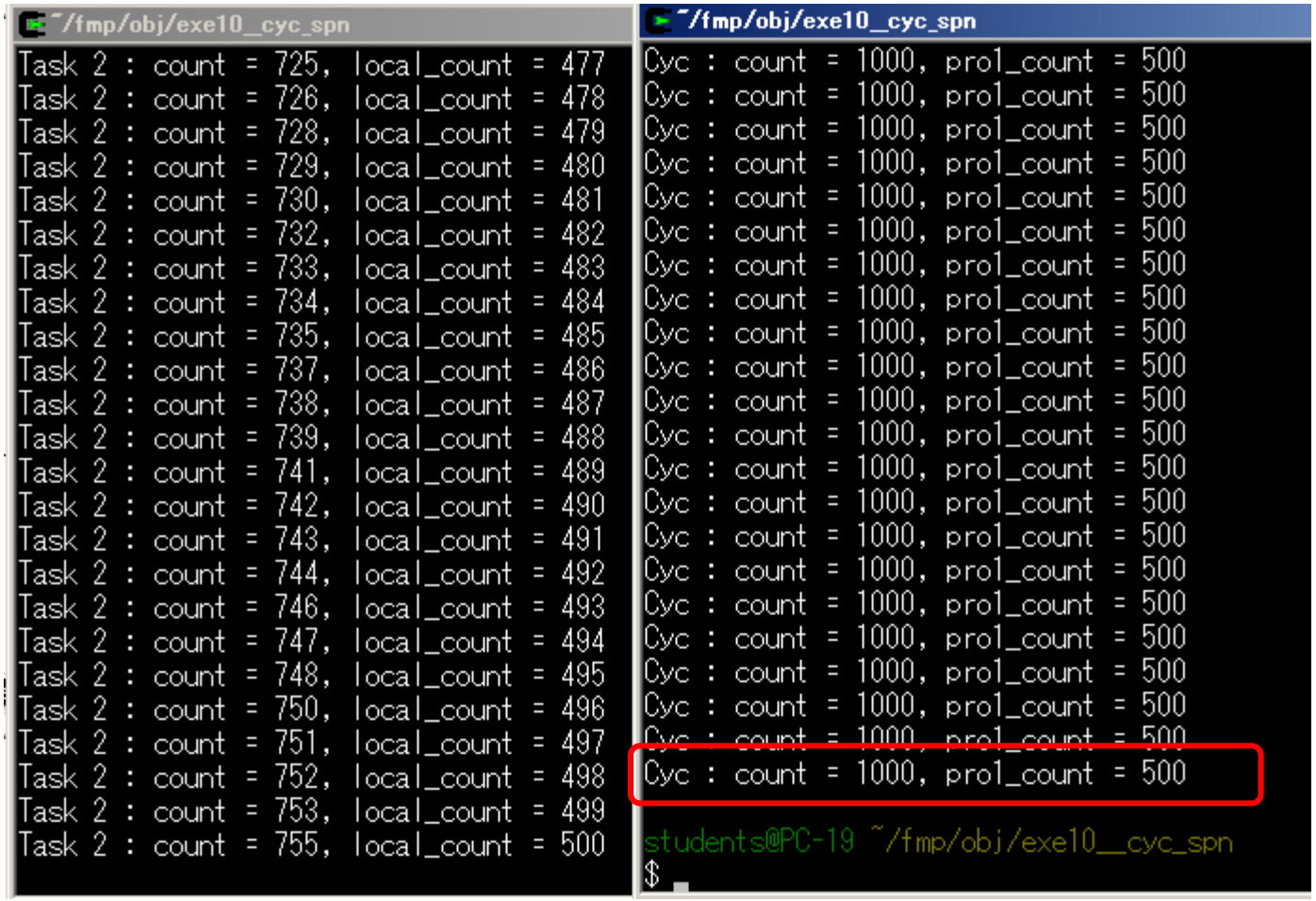
- 周期ハンドラ

```
void
cyc_handler(void){
    if(pro1_count < 500){
        pro1_count = busy_wait_inc(pro1_count);
        iloc_spn(SPN1);
        count = busy_wait_inc(count);
        iunl_spn(SPN1);
       syslog(LOG_NOTICE, "Cyc : count = %d, pro1_count = %d",  count, pro1_count);
    }
}
```

- プロセッサ2のタスク2ではCPUロックにより排他制御を試みる

```
void
task(intptr_t exinf){
    int local_count = 0,i;

    for (i = 0 ; i < 500 ; i++) {
        local_count = busy_wait_inc(local_count);
        loc_spn(SPN1);
        count = busy_wait_inc(count);
        unl_spn(SPN1);
        syslog_3(LOG_NOTICE, "Task %d : count = %d, local_count = %d", exinf, count, local_count);
    }
}
```

# 【ステップ6】: 周期ハンドラとタスク



```
Task 2 : count = 725, local_count = 477
Task 2 : count = 726, local_count = 478
Task 2 : count = 728, local_count = 479
Task 2 : count = 729, local_count = 480
Task 2 : count = 730, local_count = 481
Task 2 : count = 732, local_count = 482
Task 2 : count = 733, local_count = 483
Task 2 : count = 734, local_count = 484
Task 2 : count = 735, local_count = 485
Task 2 : count = 737, local_count = 486
Task 2 : count = 738, local_count = 487
Task 2 : count = 739, local_count = 488
Task 2 : count = 741, local_count = 489
Task 2 : count = 742, local_count = 490
Task 2 : count = 743, local_count = 491
Task 2 : count = 744, local_count = 492
Task 2 : count = 746, local_count = 493
Task 2 : count = 747, local_count = 494
Task 2 : count = 748, local_count = 495
Task 2 : count = 750, local_count = 496
Task 2 : count = 751, local_count = 497
Task 2 : count = 752, local_count = 498
Task 2 : count = 753, local_count = 499
Task 2 : count = 755, local_count = 500
```

```
Cyc : count = 1000, pro1_count = 500
Cyc : count = 1000, pro1_count = 500
Cyc : count = 1000, pro1_count = 500
Cyc : count = 1000, pro1_count = 500
Cyc : count = 1000, pro1_count = 500
Cyc : count = 1000, pro1_count = 500
Cyc : count = 1000, pro1_count = 500
Cyc : count = 1000, pro1_count = 500
Cyc : count = 1000, pro1_count = 500
Cyc : count = 1000, pro1_count = 500
Cyc : count = 1000, pro1_count = 500
Cyc : count = 1000, pro1_count = 500
Cyc : count = 1000, pro1_count = 500
Cyc : count = 1000, pro1_count = 500
Cyc : count = 1000, pro1_count = 500
Cyc : count = 1000, pro1_count = 500
Cyc : count = 1000, pro1_count = 500
Cyc : count = 1000, pro1_count = 500
Cyc : count = 1000, pro1_count = 500
Cyc : count = 1000, pro1_count = 500
Cyc : count = 1000, pro1_count = 500
Cyc : count = 1000, pro1_count = 500

students@PC-19 ~/fmp/obj/exe10_cyc_spn
$
```

<u>countはlocal_countの2倍になっている</u>

# ステップ5と6：まとめ

- タスクとタスク間の排他制御
  - セマフォを利用する
- タスクとハンドラ間の排他制御
  - 異なるプロセッサ間のタスクと周期ハンドラは，CPUロックでは排他制御できない
  - スピンロックを利用する
- 排他制御の方法

| 排他制御対象 / 排他制御の方法 | 同一プロセッサ間 | | 他プロセッサ間 | |
|---|---|---|---|---|
| | 他タスク | 割込み | 他タスク | 割込み |
| ディスパッチ禁止 | ○ | × | × | × |
| CPUロック | ○ | ○ | × | × |
| セマフォ・ミューテックス | ○ | × | ○ | × |
| CPUロック＋プロセッサ間ロック | ○ | ○ | ○ | ○ |

# アジェンダ

ステップ1. 開発環境について
- GCC, Cygwin, SkyEye

ステップ2. ASPカーネルのサンプルプログラムの解説

ステップ3. ASP用サンプルプログラムを1プロセッサで動作させる

ステップ4. ASP用サンプルプログラムを2プロセッサで動作させる

ステップ5. タスク間の排他制御の方法

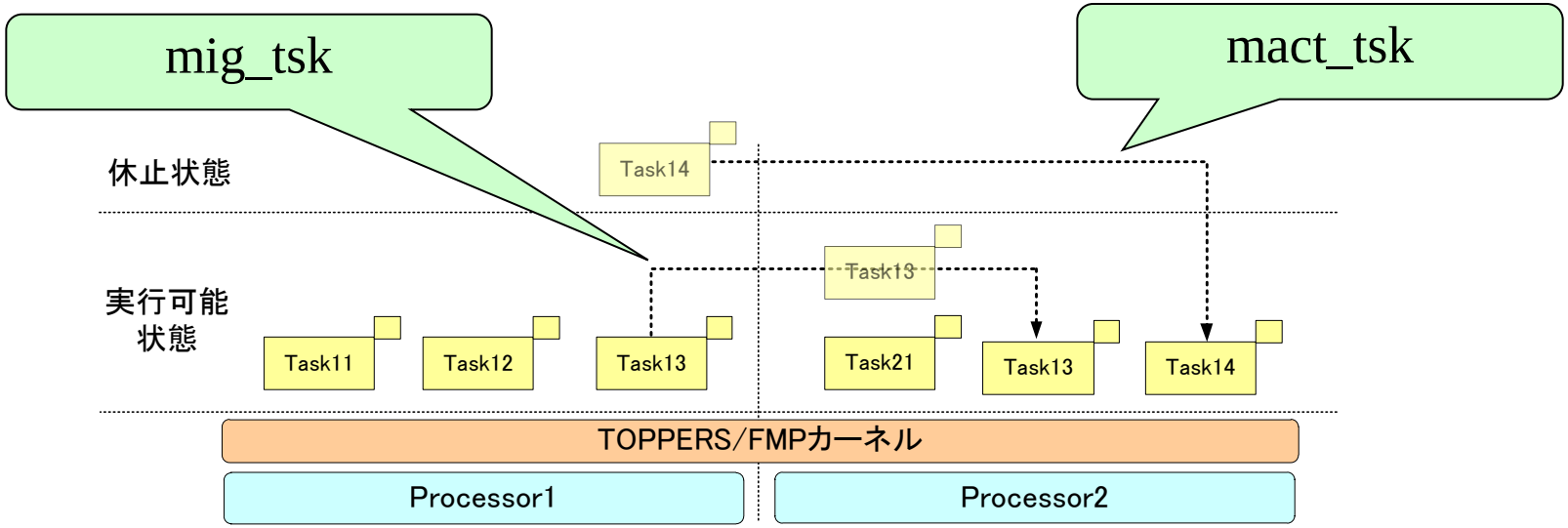
ステップ6. タスクとハンドラ間の排他制御の方法

ステップ7. 起動時タスクマイグレーションAPI(mact_tsk)の使い方

ステップ8. タスクマイグレーションAPI(mig_tsk)の使い方

# タスクマイグレーション

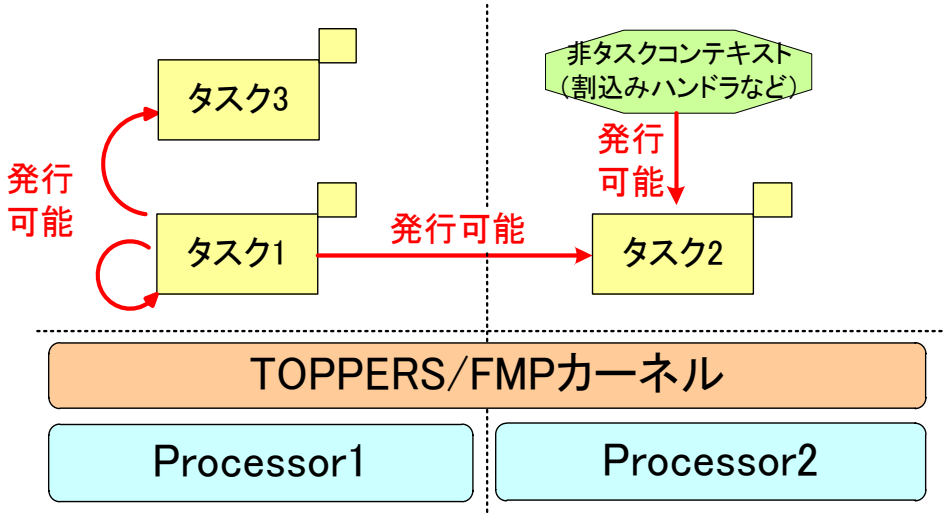
- FMPで用意されているタスクマイグレーションAPIは2つ
  1. mact_tsk
     - 次回起動するプロセッサを変更
  2. mig_tsk
     - 実行するプロセッサを変更
- 2つのAPIは使用する状況が異なる



# 【ステップ7】: 概要

起動時タスクマイグレーションAPIについて学ぶ

- 学習内容
  - 起動時タスクマイグレーションAPIの発行
- mact_tsk(ID tskid, ID prcid) / imact_tsk(ID tskid, ID prcid)
  - tskid で指定したタスクを prcid で指定したプロセッサで起動する
  - tskidに指定可能なタスクに制限はない



# 【ステップ7】: sample1.c

- メインタスクがmact_tskを発行する
  - ‘f’：指定したタスクをプロセッサ1で起動する
  - ‘F’：指定したタスクをプロセッサ2で起動する

```
void main_task(intptr_t exinf)
{
    do {
        ・・・・・・
        switch (c) {
        case 'f':
            syslog(LOG_INFO, "#mact_tsk(0x%x, 1)", tskno);
            SVC_PERROR(mact_tsk(tskid, 1));
            break;
        case ‘F':
            syslog(LOG_INFO, "#mact_tsk(0x%x, 2)", tskno);
            SVC_PERROR(mact_tsk(tskid, 2));
            break;

        ・・・・・・
    } while (c != '¥003' && c != 'Q');
```

タスク番号を表示

タスクIDで指定

# アジェンダ

ステップ1. 開発環境について
- GCC, Cygwin, SkyEye

ステップ2. ASPカーネルのサンプルプログラムの解説

ステップ3. ASP用サンプルプログラムを1プロセッサで動作させる

ステップ4. ASP用サンプルプログラムを2プロセッサで動作させる

ステップ5. タスク間の排他制御の方法

ステップ6. タスクとハンドラ間の排他制御の方法

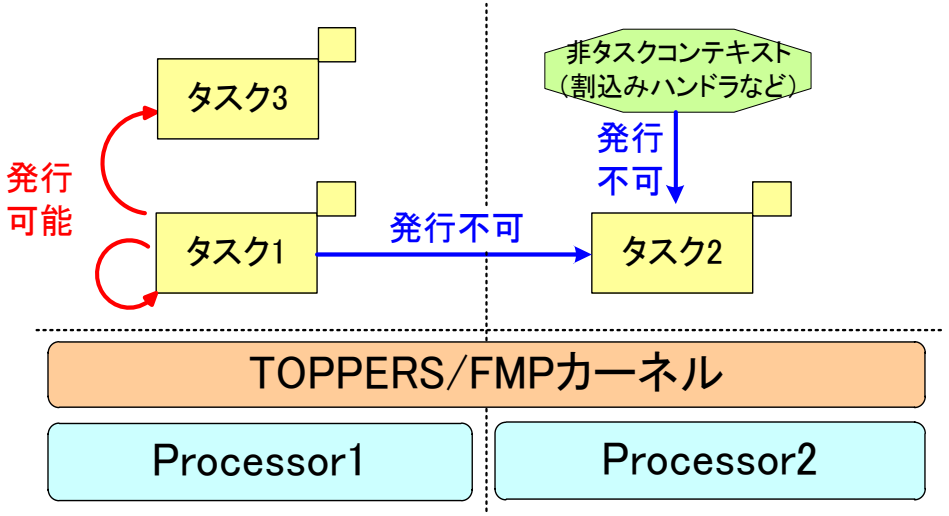
ステップ7. 起動時タスクマイグレーションAPI(mact_tsk)の使い方

ステップ8. タスクマイグレーションAPI(mig_tsk)の使い方

# 【ステップ8】: 概要

タスクマイグレーションAPIについて学ぶ

- 学習内容
  - タスクマイグレーションAPIの発行
- mig_tsk(ID tskid, ID prcid)
  - tskid で指定したタスクを prcid で指定したプロセッサに移動させる
  - tskidに指定可能なタスク
    - タスクコンテキストから自プロセッサの他のタスクに対して
    - 自タスクに対して



# 【ステップ8】: sample1.c (1/2)

- 動作確認用タスク自身がmig_tskを発行する
  - ‘g’ : 実行しているプロセッサとは別のプロセッサへ

```
void task(intptr_t exinf)
{
      ID prcid, mprcid;
      while(1) {
            ・・・・・・
            switch (c) {
            case ‘g':
                  get_pid(&prcid);
                  mprcid = ( prcid == 1 ) ? 2 : 1;
                  syslog(LOG_INFO, "#mig_tsk(0x%x, 0x%x)", tskno, mprcid);
                  SVC_PERROR(mig_tsk(tskid, mprcid));
                  break;
            ・・・・・・
      }
```

プロセッサIDの取得

実行しているプロセッサとは別のプロセッサを指定

# 【ステップ8】: sample1.c (2/2)

- メインタスクがmig_tskを発行する
  - ‘i’ : 指定したタスクをプロセッサ1で起動する
  - ‘I’ : 指定したタスクをプロセッサ2で起動する

```
void main_task(intptr_t exinf)
{
    do {
        ・・・・・・
        switch (c) {
        case ‘i':
            syslog(LOG_INFO, "#mig_tsk(0x%x, 1)", tskno);
            SVC_PERROR(mig_tsk(tskid, 1));
            break;
        case ‘I':
            syslog(LOG_INFO, "#mig_tsk(0x%x, 2)", tskno);
            SVC_PERROR(mig_tsk(tskid, 2));
            break;

        ・・・・・・
    } while (c != '¥003' && c != 'Q');
```

# ステップ7と8：まとめ

- タスクをマイグレーションさせるAPIは2つある
- mact_tsk
  - 次回起動するプロセッサを変更する
- mig_tsk
  - 実行可能状態でもプロセッサを変更できる
  - 指定可能なタスクに制約がある
    - タスクコンテキストから自プロセッサの他のタスクに対して
    - 自タスクに対して

# おわりに

- 比較的容易にASPカーネルからFMPカーネルへの移行が可能
- ただし，異なるプロセッサ間の排他制御については注意が必要

FMPカーネルのより詳しい内容を知りたい方は
名古屋大学の公開講座NEPで解説しています


http://www.nces.is.nagoya-u.ac.jp/NEP/

FMPカーネルを用いた製品・研究開発を検討したい方は
名古屋大学の組込みシステム研究センターまでご連絡下さい

NCES

http://www.nces.is.nagoya-u.ac.jp/