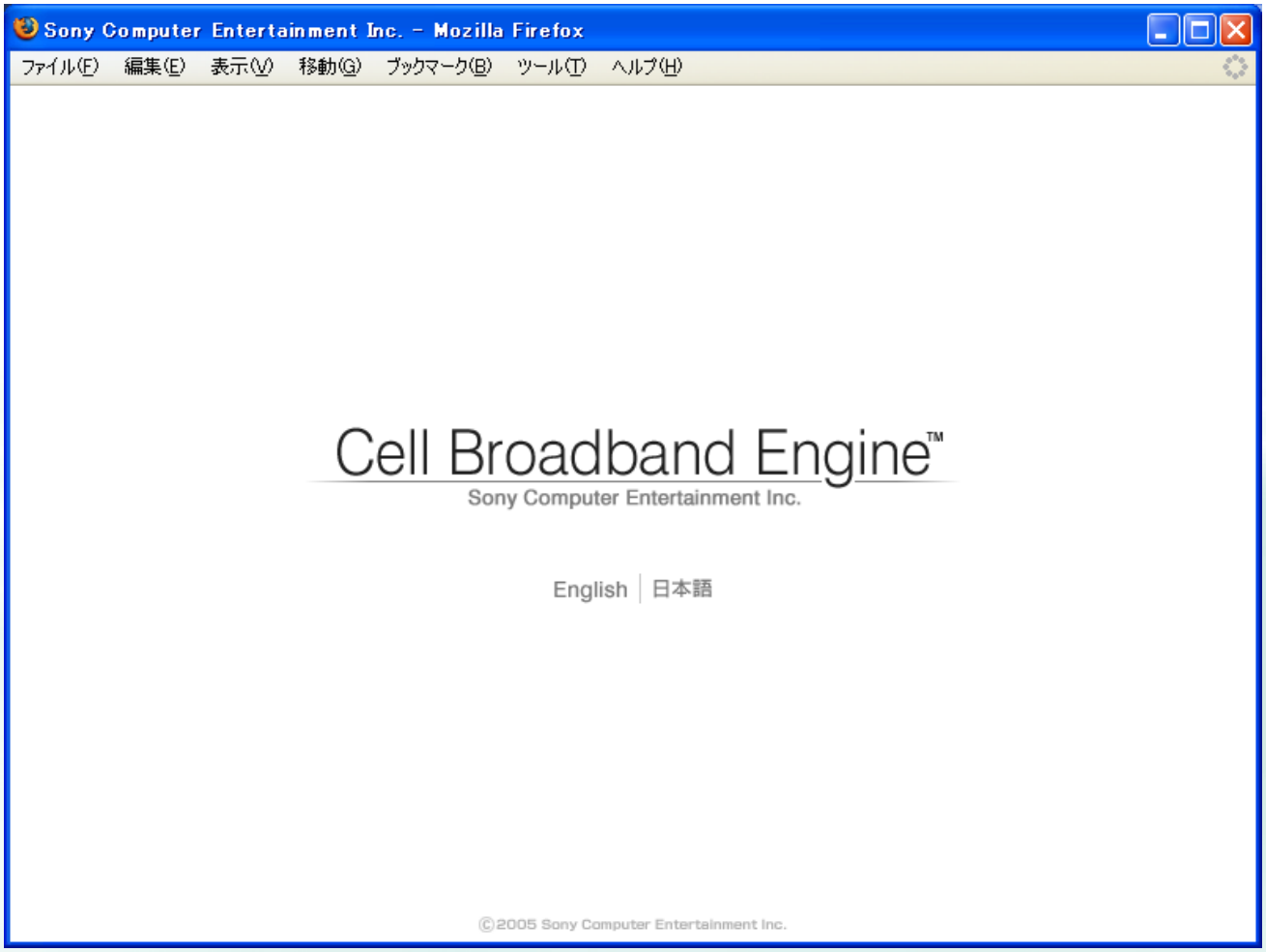
Cellプロセッサ向け実行環境
SPU Centric Execution Model

（株）ソニー・コンピュータエンタテイメント
井上敬介
Keisuke.Inoue@jp.sony.com
http://cell.scei.co.jp/

FAQ Cellの環境はいつ手に入りますか？

- 今すぐダウンロード
  - 各種日本語ドキュメント
  - Linux on Cell（SPE実行環境）
  - SPUコンパイラ（gcc、xlc）
  - フルシステムシミュレータ
- 各社の製品は各社にご確認ください

http://cell.scei.co.jp/

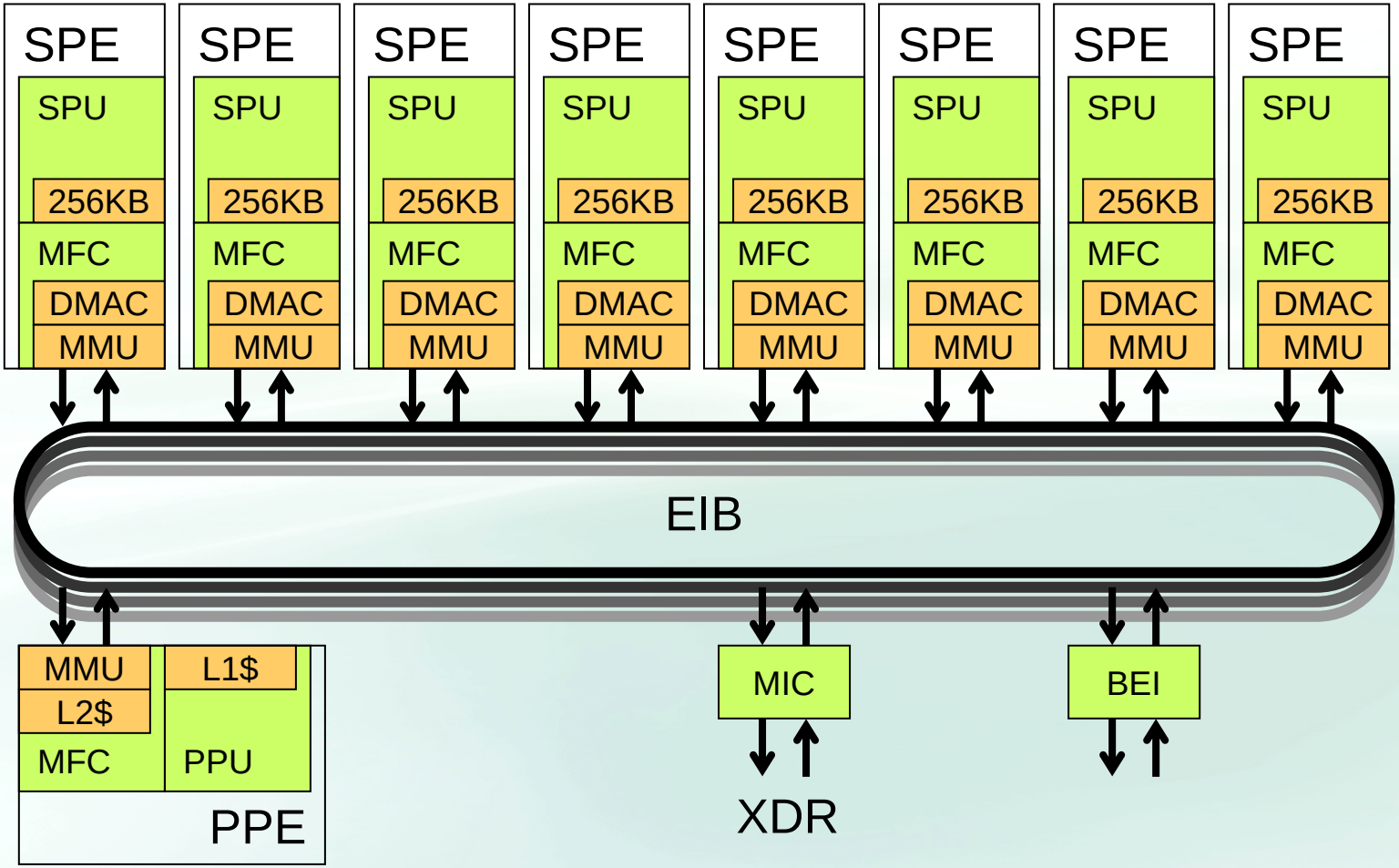


学生さんも安心の日本語ドキュメント！

内容

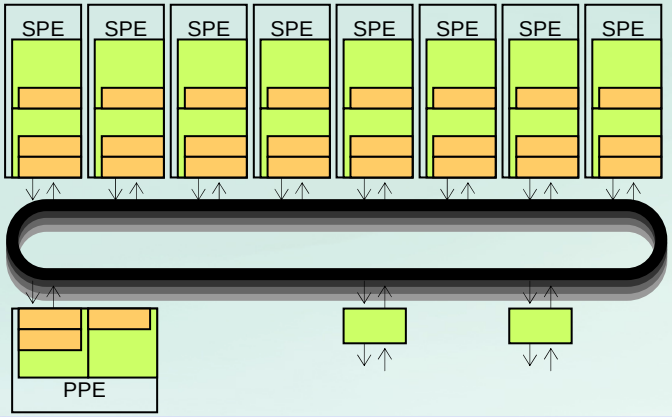
- Cell Broadband Engineの概要
- SPU Runtime Systemのコンセプト
- 並列実行モデル その1
- 並列実行モデル その2
- 実行モデル間での効果的なSPUの共有

Cell Broadband Engineのブロック図



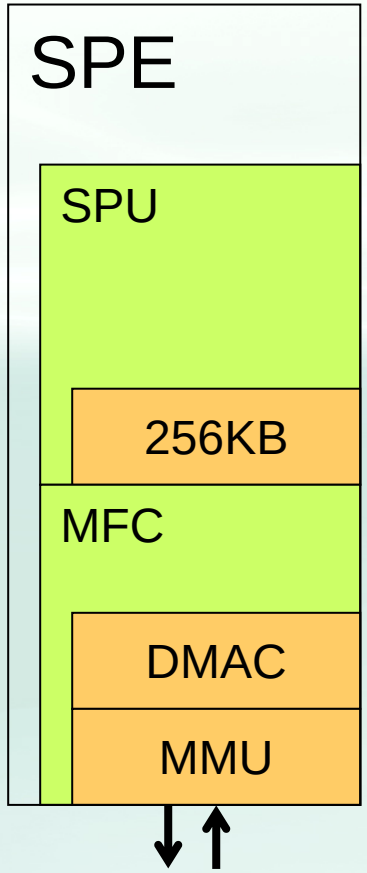
Cell Broadband Engineの並列処理的側面

- ヘテロジニアスなチップマルチプロセッサ
  - SPE視点ならシンメトリック
- キャッシュコヒーレンシ
  - PPEのキャッシュ
  - SPEのアトミックキャッシュ（同期用）
  - SPEのDMA



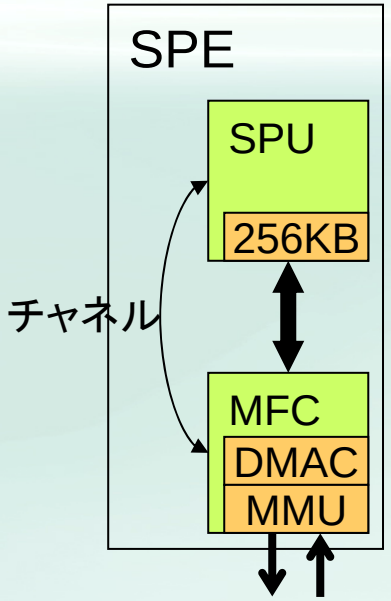
SPEの概要

- SPU - Synergistic Processing Unit
  - シンプルな構造の普通のプロセッサ
    - シンプルで静的な振る舞い
    - 動作の予測が容易
  - 128本の128ビットレジスタファイル
  - ほとんどSIMD命令
  - 256KBローカルストレージ（LS）
    - L1キャッシュ相当のスピード
    - 固定レイテンシ
    - コードとデータで共用
- MFC - Memory Flow Controller
  - DMAコントローラ
  - Memory Management Unit



SPEの並列処理的側面

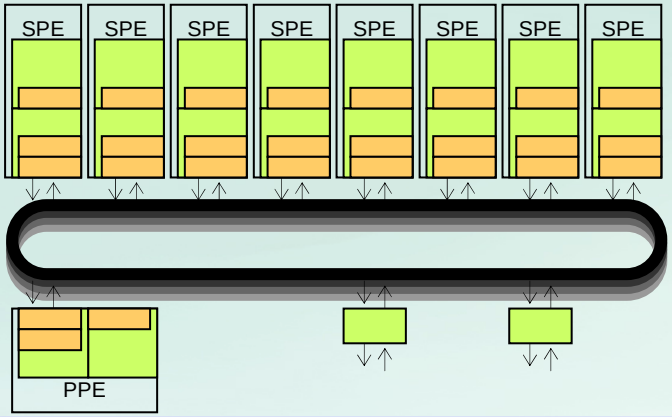
- SPUとMFCは別プロセッシングユニット?
  - メッセージパッシング
    - SPU⇔MFCはチャネルインタフェースを通して通信
      - リクエスト→ステータスリード(ブロッキング、ノンブロッキング)
  - LS⇔MFCは別パス
  - お互いに独立動作



Cellの性能を存分に生かすには？

- 当然SPEを効率よく使い切る
- あらゆるレベルで並列処理
  - Processing Element
  - XDR⇔LS転送と計算
  - LS⇔レジスタ転送と計算
  - Dual Issue
  - SIMD

ここが勝負



SPEを効率よく使い切るには？

- 高い独立性でSPUを実行
  - Do not stop SPUs!
  - Do not disturb SPUs!
  - 同期は最小限に
  - 排他（mutex）の利用は計画的に
    - プリエンプションの発生する環境で危険
  - Think about SMP (SPE centric Multi-Processing)

Pitfalls

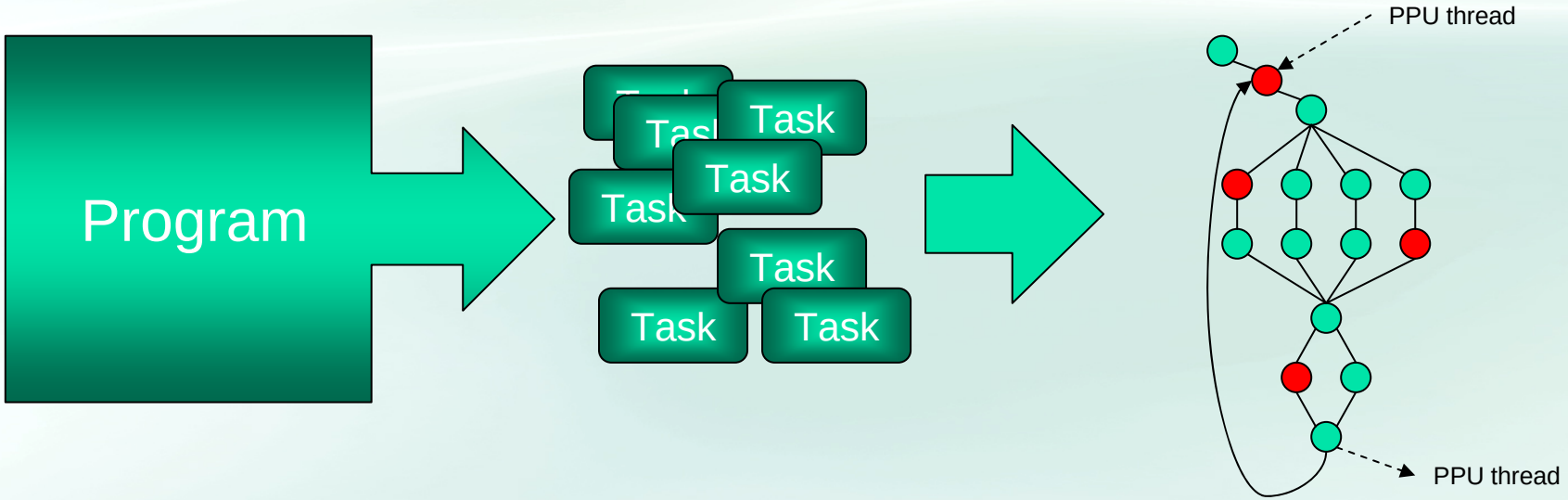
- OSによって仮想化されたSPEでmulti-threading
  - サービスコールはリモートPPUコール
    - SPUに特権モードはない
    - ブロッキングコールだとSPEは待ちぼうけ
  - SPEのプリエンプションコストは安くない
    - 参考文献：Cell Broadband Engine Architecture
- PPU centric multi-processing
  - SPUから使いづらいデータ構造になりがち
  - SPUをコプロセッサとして使いがち
    - 密結合でないコプロセッサは通信コストで負ける
  - SPUへ関数オフロードしがち
    - 判断基準がSPU向きの処理かどうかではアムダールの法則により効果薄

Philosophy of SPURS

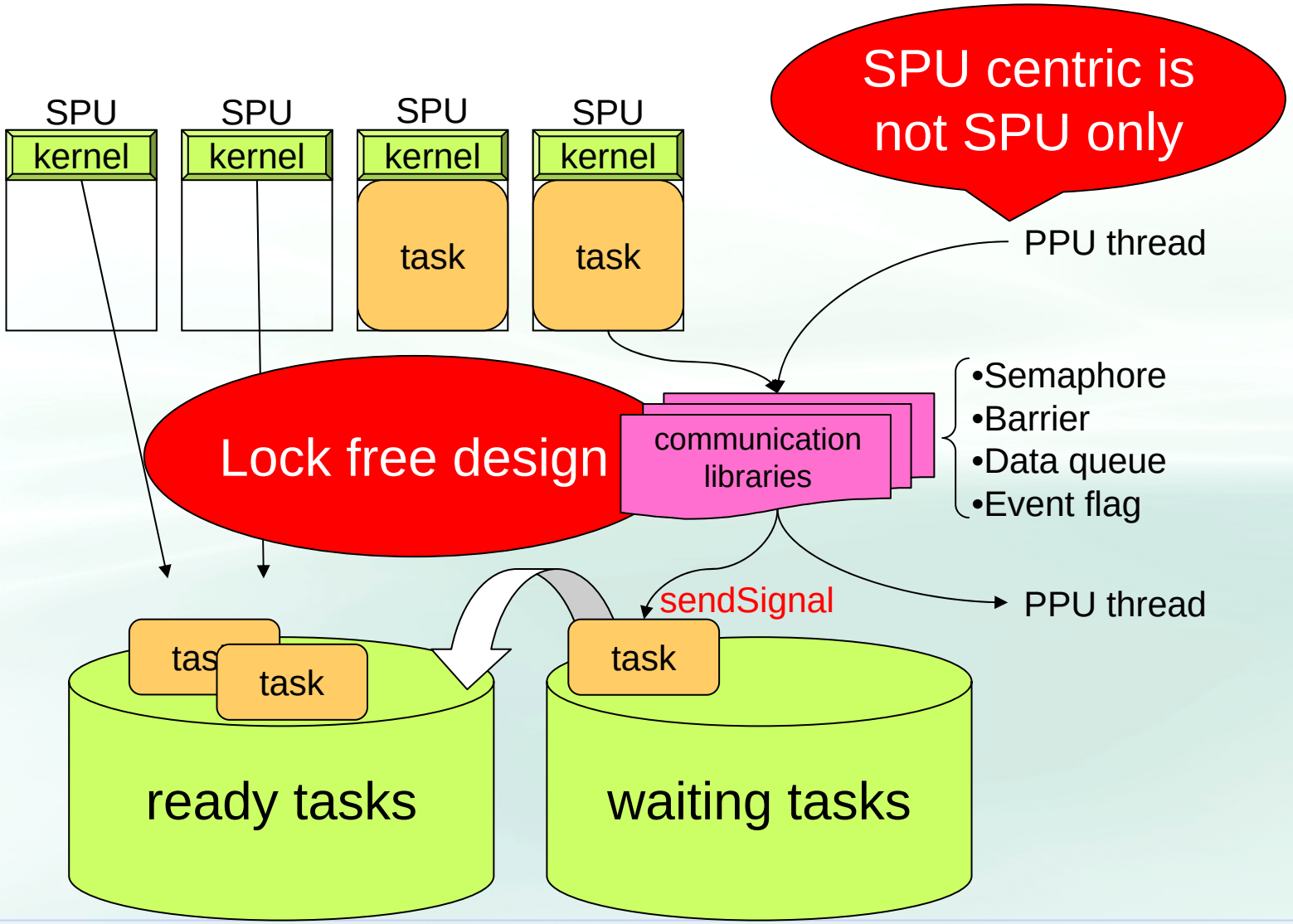
- 与えられたSPUを効率よく利用
- PPUに依存せずにSPUコードを選択／実行
- 機能は効率重視で割り切る
  - MFCのコンテキストは保存せずDMA完了を待つ
- 協調的プリエンプション
- PPUはサービスコプロセッサ
- KISS (Keep It Simple and Static behavior)

アプリケーションを複数SPUで実行 その1

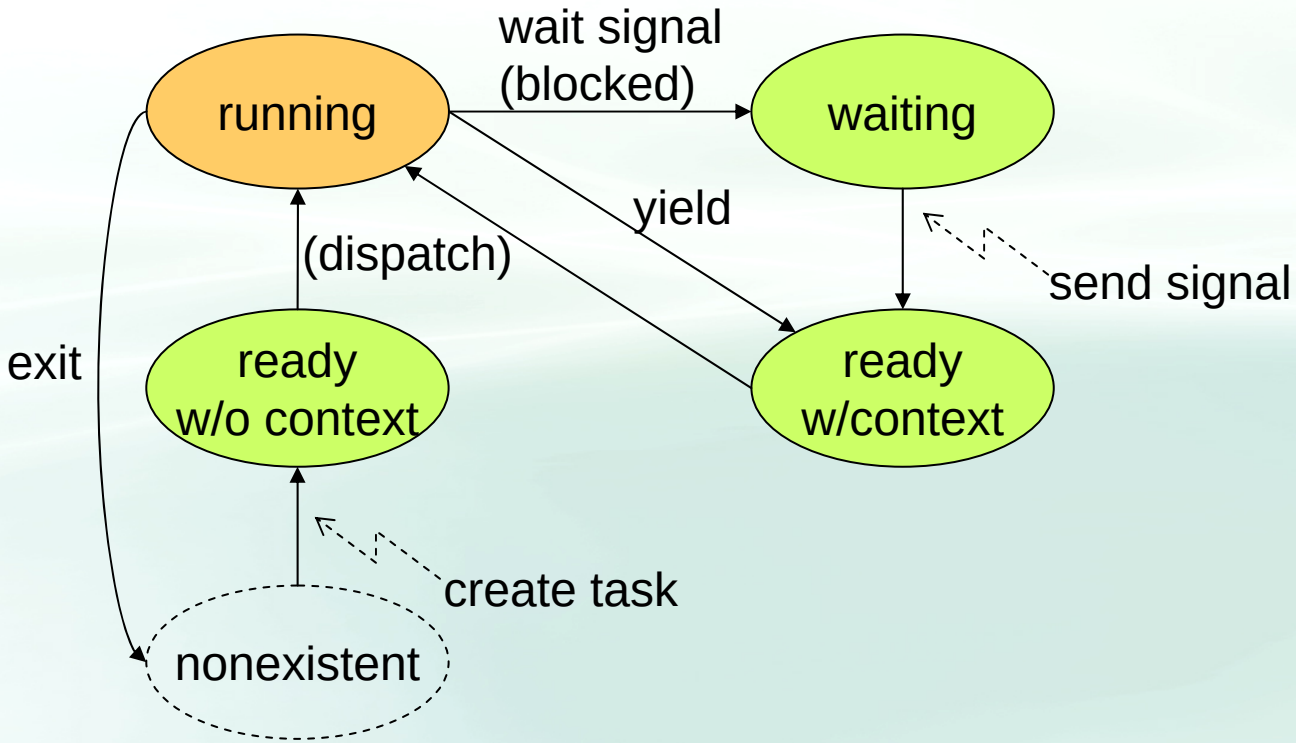
- プログラムを複数の小さなタスクに分割
- 通信ライブラリを用いてタスク間を依存関係で結合
- LS常駐のカーネルは実行可能なタスクを選んで実行



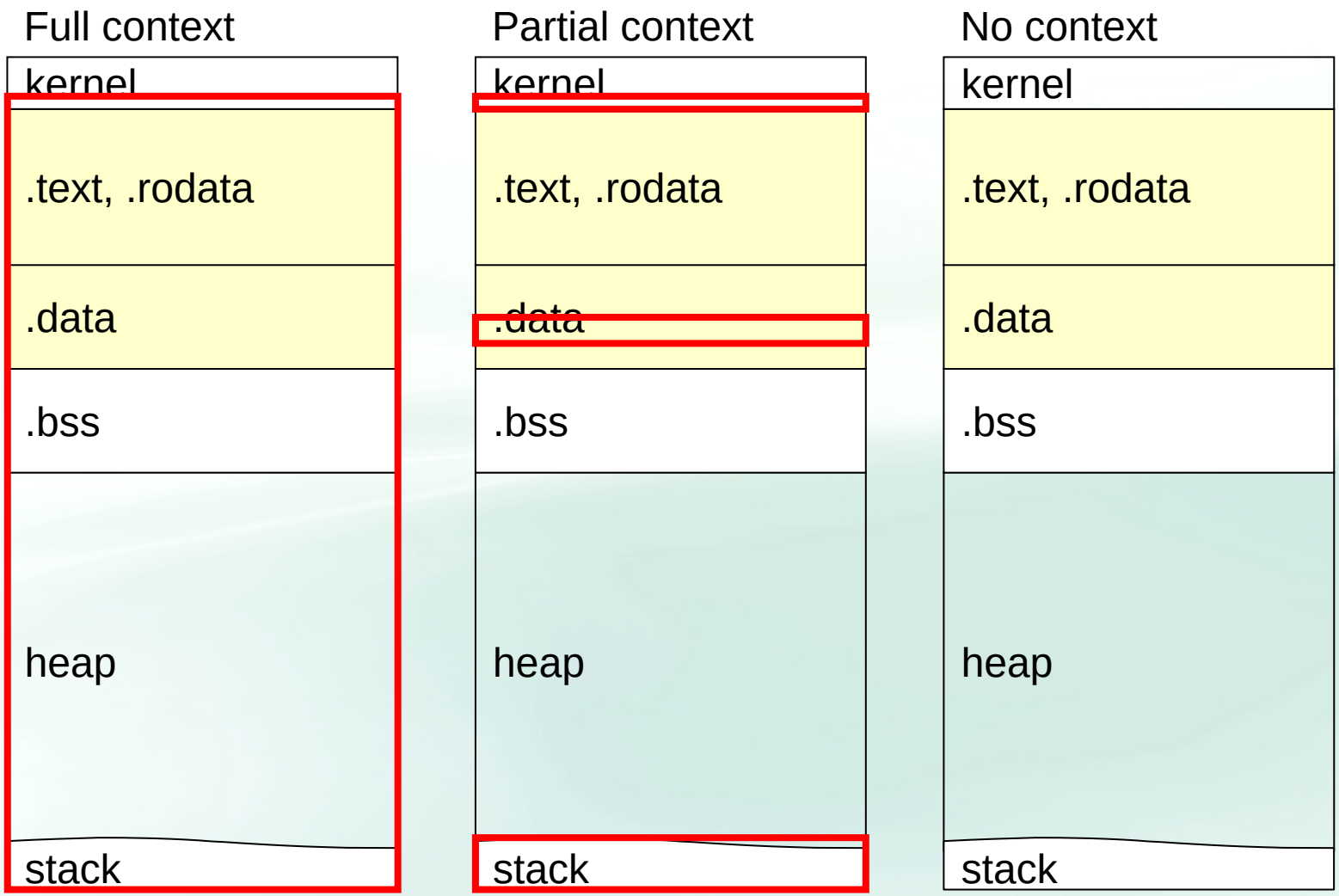
SPURSタスク実行モデル



SPURSタスクの状態遷移図



タスクのLSコンテキストサイズは任意

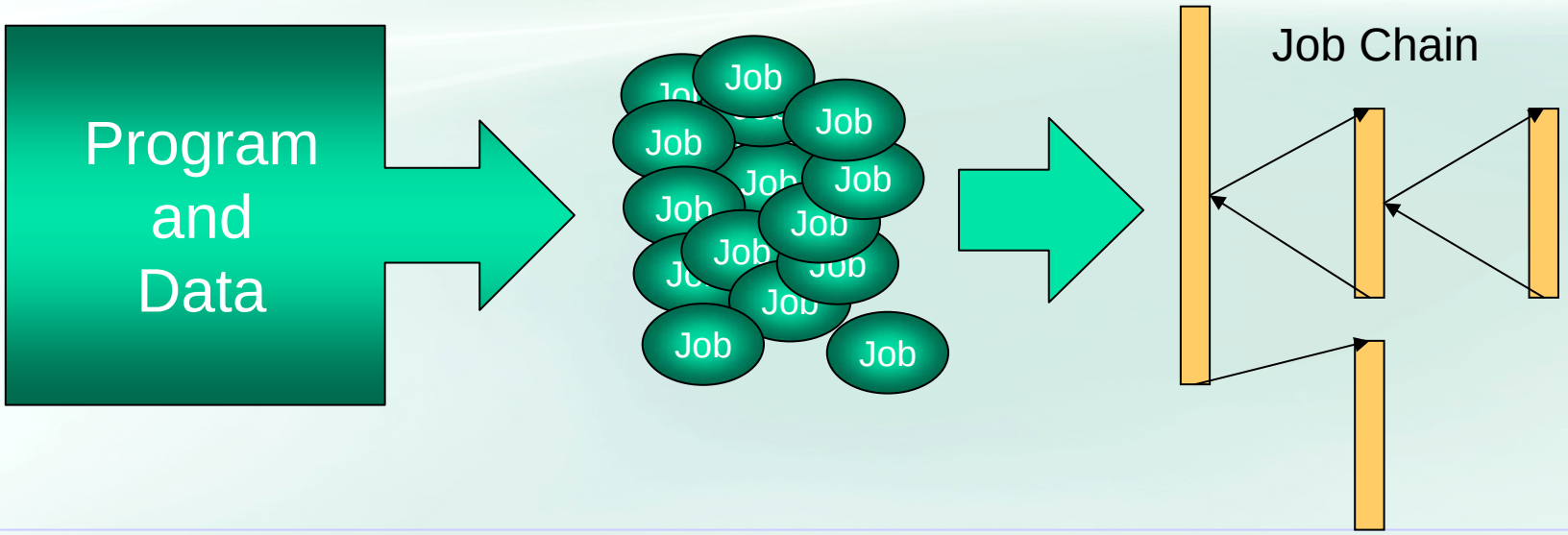


SPURSタスク実行モデルのまとめ

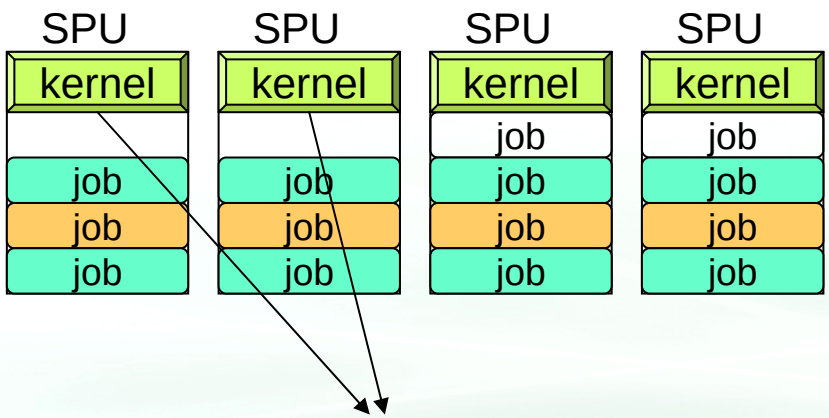
- Non preemptive multi-SPU multi-tasking
- 固定メモリレイアウト
  - ユーザメモリエリアはLSの95%以上
- ロックフリーデザイン
  - PPUスレッドもSPURSタスクと協調動作可能
  - 協調動作中のPPUスレッドがプリエンプトされてもSPU側は動作し続ける
- 軽い動作の同期・通信ライブラリ
  - PPUスレッドをブロックする以外にシステムコールは呼ばない

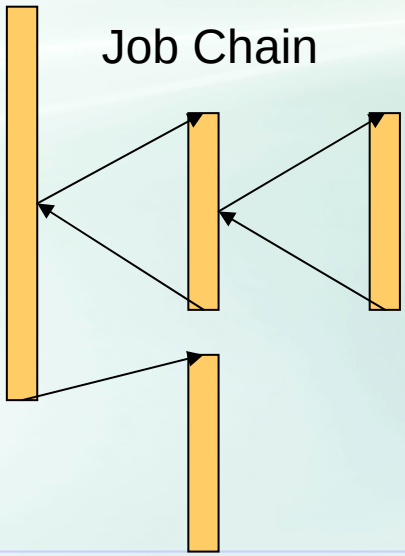
アプリケーションを複数SPUで実行 その2

1. プログラムとデータを小さいジョブに分割
2. ジョブを並べる（Job Chain）
3. LS常駐のカーネルはジョブリストからジョブを取ってきて実行



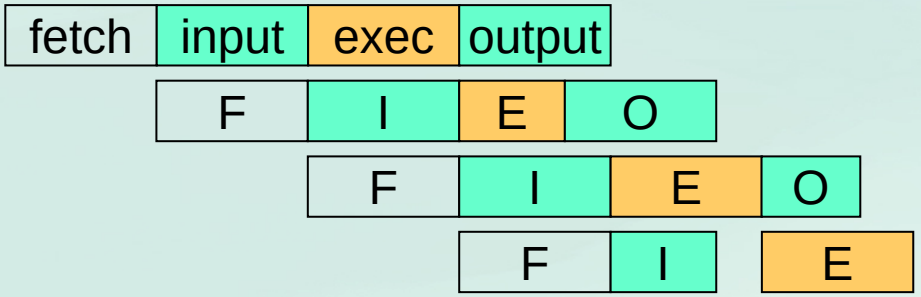
SPURSジョブ実行モデル



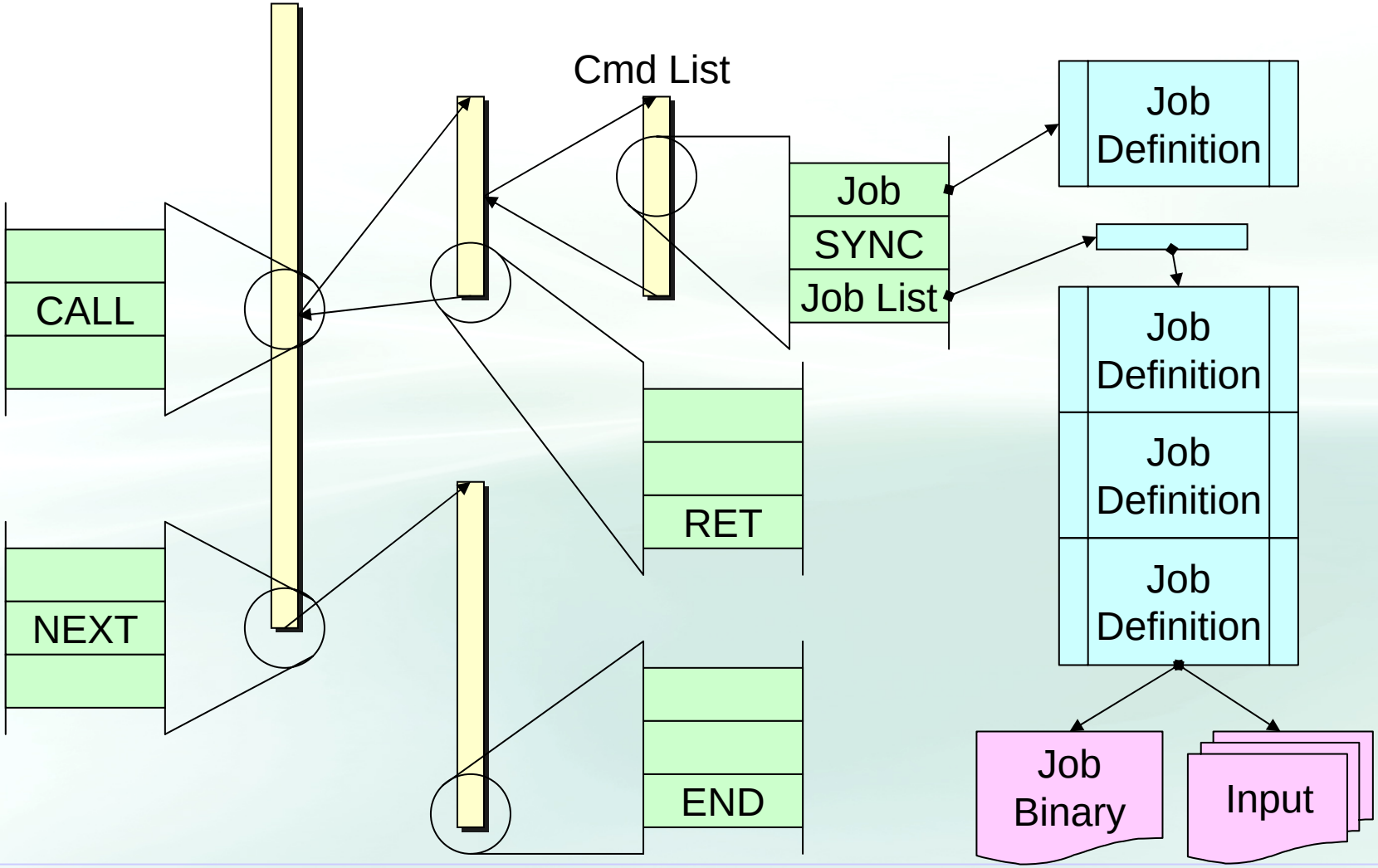


4 stage pipeline

- In order job fetch DMA
- Out of order input DMA
- Out of order execution
- Out of order output DMA



SPURSジョブチェイン



SPURSジョブ実行モデルのまとめ

- Multi SPU execution
  - ジョブチェインは複数のカーネルから共有される
- ジョブはSPE上で4ステージパイプライン実行
  - ジョブとDMAも並列実行
- 柔軟なメモリ管理（mallocは未使用）
  - ジョブはほぼ全メモリの使用も可能
    - パイプラインストールが発生
  - バイナリやroデータをキャッシュ可能
- 優先ジョブコマンド（Urgent Job）
- ジョブチェイン、ジョブリストは再利用可能

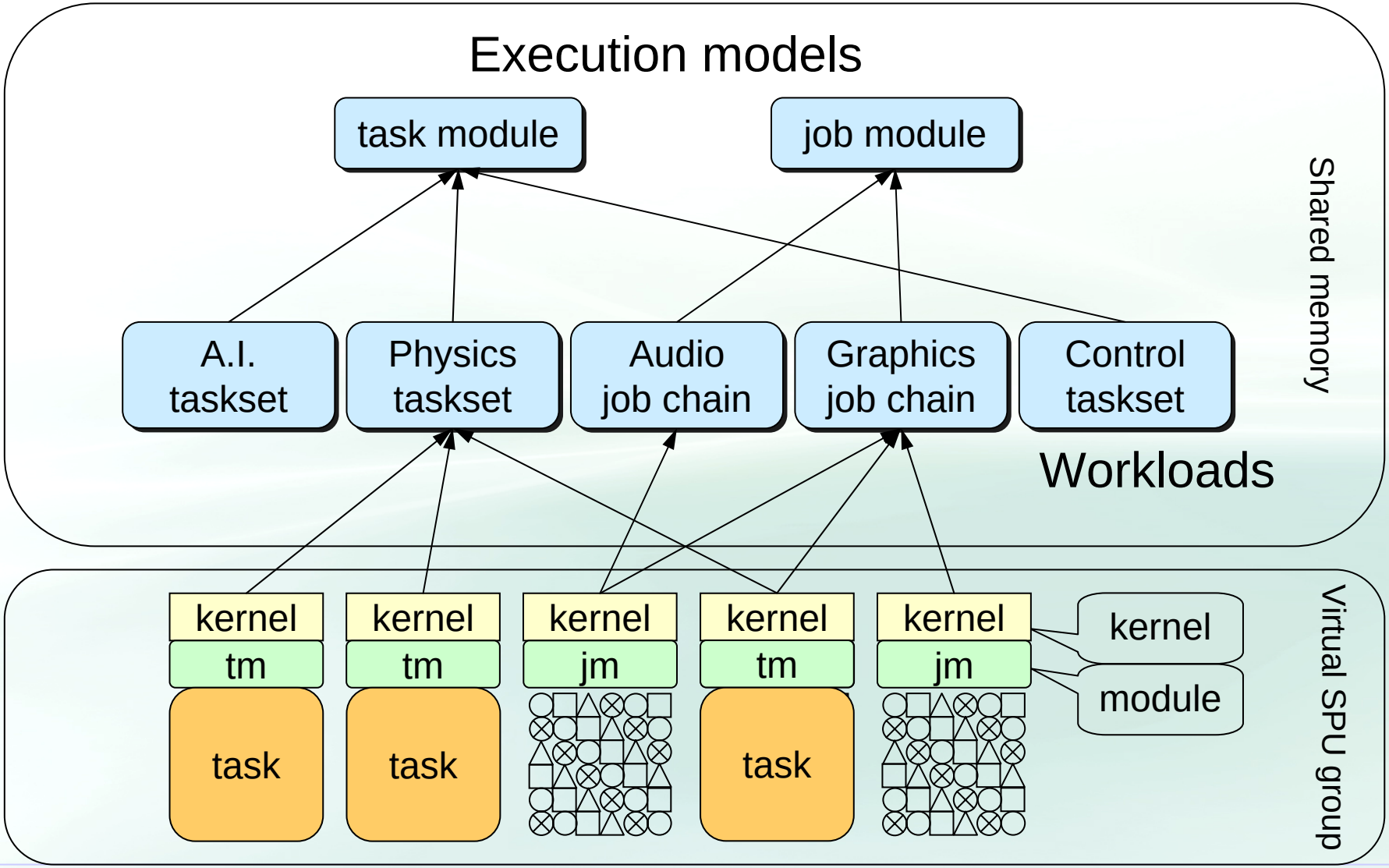
さらにSPUの利用効率を上げる

- 規模が大きいと統合が発生
  - 両方の実行モデルが同時に必要
    - タスク：固定デザインやコード中心のプログラムスタイル
    - ジョブ：データ志向での柔軟なプログラム構成
  - ミドルウェアや社内共通プログラムの使用
- 統合は利用効率問題につながる
  - SPUを機能割りしたくない
  - 仮想化したSPUのスイッチバック回数とオーバヘッド
  - Integrator’s controlability
- Integrator's control
  - No virtualization!
  - No unpredictable behavior!

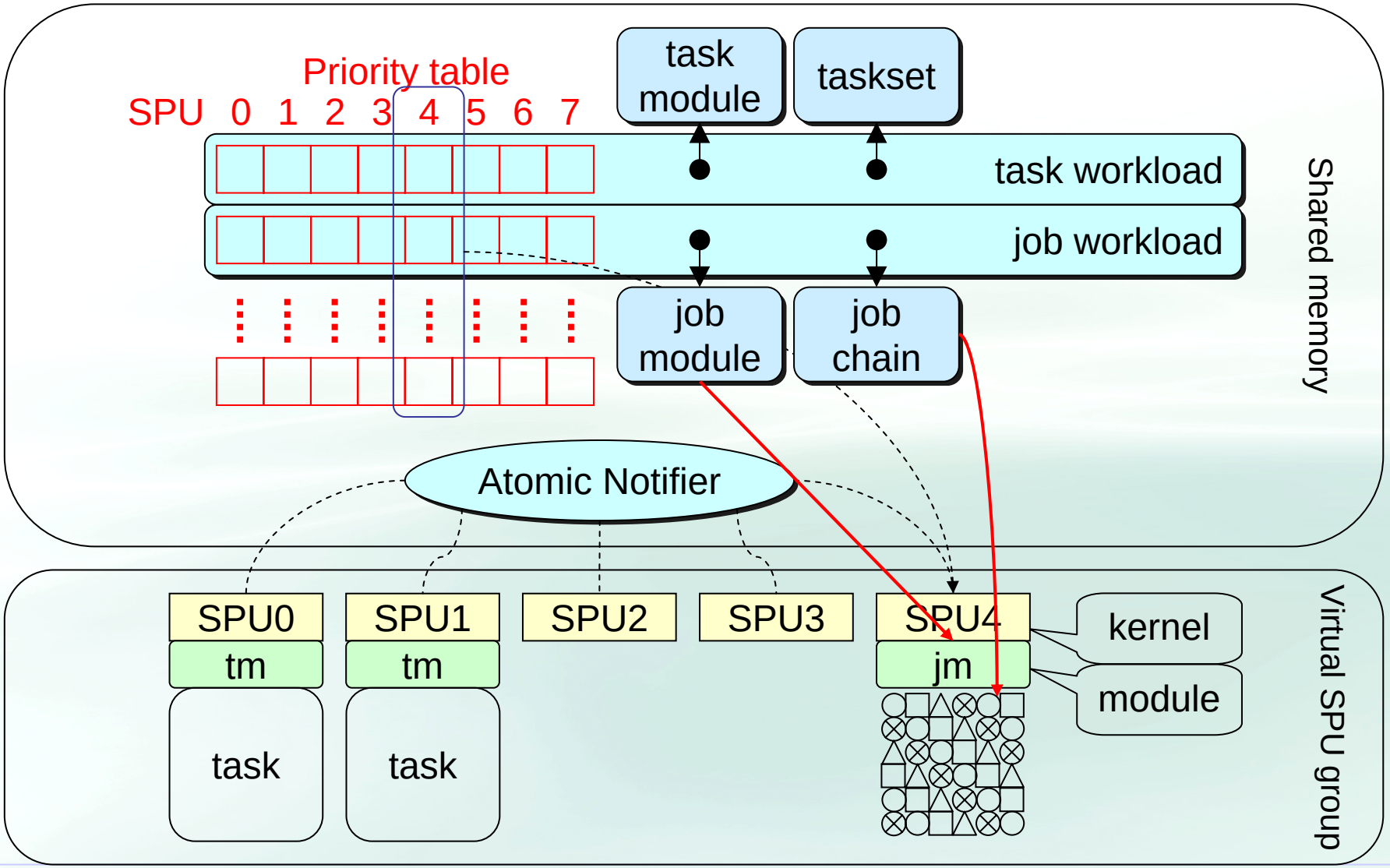
SPURS Minimum Integration Platform

- Full control of scheduling to integrator
  - Matrix style priority
- Small footprint
  - less than 1% of Local Storage
- Non virtualized static behavior
  - Non-preemptive
- Inter-workload balancing
  - Next chart

SPU Runtime System Overview



Scheduling And Priority Table



SPURSのまとめ

- SPURSは2レイヤ構成
  - 2KBのワークロードスケジュール層
  - ポリシーモジュール（実行モデル）層
- 効率の良い実行
- 協調的なSPUの共有
- インテグレータに嬉しい見通しの良い動作
  - No virtualization
  - Non-preemptive
  - Static behavior of scheduling