コンピュータシステムII（7）

情報基盤センター
天野　浩文

1

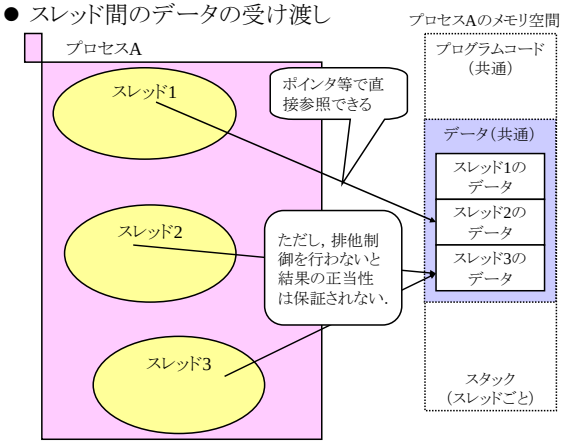
前回のおさらい（1）

- 並列処理においては，どうしても他のプロセス（スレッド）とデータを受け渡さなければならないことがある．
- データ受け渡しの方法
  - スレッドどうしでデータを受け渡すには…
    - メモリ空間を共有しているので，他のスレッドの変数に直接アクセスできる．
    - 他のスレッドの変数をメモリ上の番地により直接参照できる．
      - ただし，書き込み時の排他制御は必要
  - プロセスどうしでデータを受け渡すには…
    - メモリ空間が共有されないので，他のプロセスのデータに直接アクセスすることはできない．

2

前回のおさらい（2）

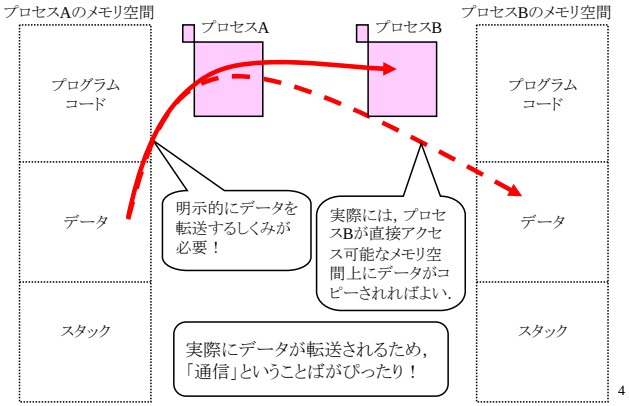
- スレッド間のデータの受け渡し



3

前回のおさらい（3）

- プロセス間のデータの受け渡し



4

前回のおさらい（4）

- プロセス間通信の実現法
  - **「プロセスどうしはメモリ空間を共有しない」という原則を守る場合**
    - 異なるメモリ空間の間のデータのコピーとして実現
    - ただし，データのコピーにはそれだけの処理時間がかかる．
  - データのコピーを避けるための方策
    - プロセスのメモリ空間の**一部または全部が重なっていてもよいことにする**．
    - コピーは不要になる．
    - しかし，プロセスのメモリ空間の「保護」は難しくなったり，不可能になったりする．

5

前回のおさらい（5）

- 共有メモリを経由したデータ受け渡し
  - 物理的な共有メモリがある場合には特に有効
    - 実記憶
      - 仮想記憶を用いない
    - 単一仮想記憶
      - 仮想記憶空間の全体が共有される
    - 多重仮想記憶
      - 仮想記憶空間の一部が共有される
- メッセージ通信
  - 送り手と受け手がそれぞれ固有の操作を行うようにプログラムを記述
    - 受け手（のプロセスID）を指定してメッセージを送る（SEND）
    - 送り手（のプロセスID）を指定してメッセージを受け取る（RECEIVE）
  - ただし，メッセージ通信の実現法としては，共有メモリを使用してもよい．

6

前回のおさらい（6）

- **分散共有メモリ**（distributed shared memory）
  - ハードウェア的な共有メモリが存在しなくても，ソフトウェアでそれを実現する．
  - 仮想共有メモリ（virtual shared memory）と呼ぶこともある．
- **実現方式の分類**
  - **コンパイラ方式**
    - コンパイル時に，リモートメモリへのアクセスを通信処理に書き換えて，実行可能プログラムを生成する．
    - 新しいコンパイラと，既存のアプリケーションプログラム（AP）の再コンパイルが必要．
  - **実行時方式**
    - プログラムの実行時に分散共有メモリへのアクセスがおきると，例外（exception）を発生させて制御をOSに移す．
    - OSが，リモートメモリへのアクセスを代行する．
    - 既存のAPはそのまま使えるが，OSの改造が必要．

7

前回のおさらい（7）

- リモートメモリの操作方式
  - 複写方式
    - 遠隔の演算ノードのメモリへのアクセスが発生すると，操作対象のメモリ域を手元に複写してくる．
    - 複写されたメモリ域を操作する．
    - 更新があれば，更新内容を他の演算ノードに反映させる．
    - 複写方式においては，一貫性制御が必要
      - SC, RC, LRC, HLRC
  - 非複写方式
    - 遠隔の演算ノードのメモリへのアクセスが発生すると，メモリ操作の内容をそのメモリのある演算ノードに転送する．
    - 転送された操作内容に基づいてメモリ操作を代行してもらう．

8

キャッシュコヒーレンス問題

この項目は，教科書には含まれていないが，分散共有メモリの一貫性制御によく似た問題であるので，ここで解説しておく．

9

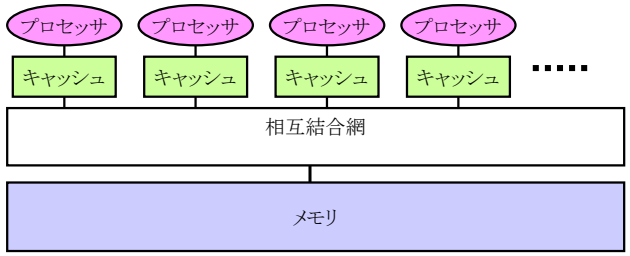
共有メモリ型なら一貫性の問題は起きないのか？

- キャッシュ（cache）
  - メモリよりも高速の素子で作られた記憶域
    - ただし，この記憶域はメモリに比べて高価なので，サイズは小さい．
    - プロセッサの動作速度に比べてメモリの動作速度は遅いので，頻繁にアクセスするデータはメモリからここに複写して使用する．
      - キャッシュに保持するデータの単位を**キャッシュライン**と呼ぶ．
  - 各プロセッサに，キャッシュが存在する．
    - シングルプロセッサでは，メモリとキャッシュの内容が一致しないことがある．
    - 共有メモリ型並列計算機では，実メモリとキャッシュ，あるいは，キャッシュ相互の内容が一致しない可能性が出てくる．

10

共有実メモリとキャッシュ

- キャッシュコヒーレンス（cache coherence）
  - 分散共有メモリの一貫性制御と同様の問題
  - ただし，分散共有メモリの場合よりも，**ハードウェアを積極的に活用することが多い**．



11

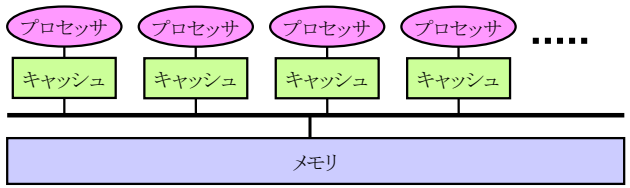
単一プロセッサのキャッシュコヒーレンス

- 大きく分けて2つ（このような機能をハードウェアで実現する）
  - **ライトスルー**（write through，ストアスルー store through ともいう）
    - キャッシュに書き込むときは，メモリにも反映させる．
      - 書き込みのたびに待たされる．
      - バッファ（buffer）にためておけば，待たずに先に進むことも可能．
  - **ライトバック**（ write through，ストアイン store in ともいう）
    - キャッシュが更新されメモリとの不一致がおきても，該当するキャッシュラインに「dirty」とだけ記録して，そのままにしておく．
    - キャッシュはメモリよりもはるかに容量が小さいので，入り切れなくなると，捨てなければならない．
    - 捨てるときに，dirtyなキャッシュラインをブロックにまとめてメモリに反映させる．

12

キャッシュコヒーレンスを保つ機構（1）

- スヌープキャッシュ（snoop cache）
  - バス型の共有メモリで有効
    - 各プロセッサは，ライトスルーかライトバックで自分のキャッシュとメモリのコヒーレンスを保つ．
      - メモリへの書き込みは，必ずバスを通る．
    - 他のプロセッサは，バスを監視して，自分のキャッシュにあるデータへの書き込みがあれば，それを受けて**自分のキャッシュの内容を変更**する．



13

スヌープキャッシュによるコヒーレンス保持（1）

- ライトスルー
  - 無効化型
    - メモリ上のデータが更新されると，該当するキャッシュラインを無効にする（捨てる）．
  - 更新型
    - 更新されると，それに合わせて，該当するキャッシュラインを更新する．

14

スヌープキャッシュによるコヒーレンス保持（2）

- 無効化型ライトバック
  - キャッシュ上のデータを更新すると，そのプロセッサでは，そのキャッシュラインをdirtyにする．そのことを伝える信号がバスに流れると，他のプロセッサは，そのラインを捨てる．
  - dirty なキャッシュラインを持つプロセッサは，そのキャッシュラインがライトバックされるまではキャッシュを自由に変更できる．
  - dirtyなキャッシュラインへの書き込み要求が他のプロセッサから出ると，そのラインがメモリにライトバックされてから，そのプロセッサのキャッシュに複写される．
- 更新型ライトバック
  - キャッシュラインへ書き込みがバス上で検知されると，他のプロセッサもキャッシュラインを変更する．

15

キャッシュコヒーレンスを保つ機構（2）

- バス型以外では…
  - どのプロセッサに最新のキャッシュラインがあるのか，全プロセッサに問い合わせなければわからない．
  - キャッシュラインがdirtyになったことを他の全プロセッサに伝えるのが困難．
- ハードウェアで実現する方法：**ディレクトリ方式**
  - 同じデータが他のどのプロセッサにキャッシュされているかを示す情報をキャッシュ内に保持しておき，これによって無効化や更新を行う．
- ソフトウェアで実現する方法
  - キャッシュコヒーレンスを保証しなければならないタイミングをプログラムに書いておき，その時点でライトスルーする．
  - ライトスルーする際には，同じキャッシュラインを持つ他のプロセッサに無効化の命令を送る．
    - こうしておけば，新しいキャッシュラインを作るときには，常にメモリから読めばよいので，他のプロセッサに問い合わせる必要はない．

16

キャッシュコヒーレンスのまとめ

- 共有メモリ型並列計算機でも，分散共有メモリと同様の一貫性の問題がある．
  - メモリとキャッシュの間
  - キャッシュ相互
- 単一プロセッサのキャッシュとメモリ
  - ライトスルーとライトバック
- 並列計算機では
  - スヌープキャッシュ
  - ディレクトリ方式
  - ソフトウェアによる方式

17

メモリやキャッシュの一貫性についてのコメント

- メモリコンシステンシやキャッシュコヒーレンスを議論するときの注意点
  - 単一の読み出しや書き込み操作が正しく行われるかどうか（生き残れるかどうか）だけを考えていることが多い．
    - 書き込んだ結果が反映されない（よそで書かれた結果を読めない）というようなことが起きないように制御する．
  - 複数の読み出しや書き込みからなる一連の処理手順全体が矛盾なく実行されるかどうかまでは考えていない．
    - このようなレベルでの「正しさ」を保証するためには，**同期機構**や**並行処理制御**を必要とする．

18

プロセスのスケジューリング

19

スケジューラ（scheduler）

- プロセスの実行順序を決定する機構
  - 単一プロセッサのシステムでは
    - プロセスに割り当てられたタイムスライスが終了したとき，または，プロセスが待ち状態に入ったときにプロセス切り替えが発生するので，そのときに介入する．
    - 目的
      - プロセッサの遊ぶ時間を少なくし，処理効率を向上させる．
    - 次に実行するプロセスは，最も優先度の高いもの，または，最も早く待ち状態に入ったもの，など
  - 並列処理や分散処理におけるスケジューリングの目的
    上記の目的に加えて…
    - 負荷分散
    - プロセッサ間通信の削減

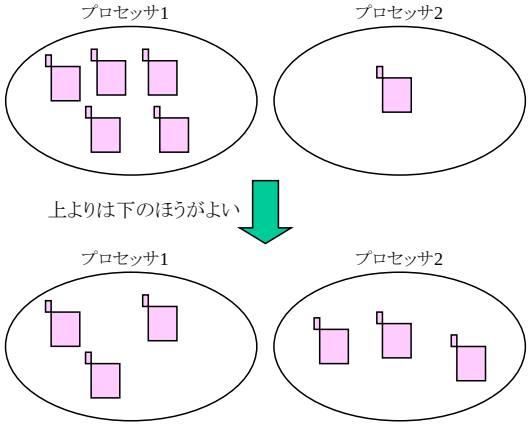
20

負荷分散

- プロセッサ負荷の分散
  - 各プロセッサで動作するプロセスの数ができるだけ均等になるようにする.
    - 特定のプロセッサに多数のプロセスが集中していると
      - 特定のプロセッサだけ仕事が多い.
      - プロセス切り替え(メモリ空間の切り替え)が多数発生して効率が低下する.
- メモリ負荷の分散
  - 各プロセッサで必要となるメモリができるだけ均等になるようにする.
    - 大きなメモリを使用するプロセスが特定のプロセッサに偏っていると, 仮想記憶のためのページング(=ディスク入出力)が多数発生して効率が低下する.

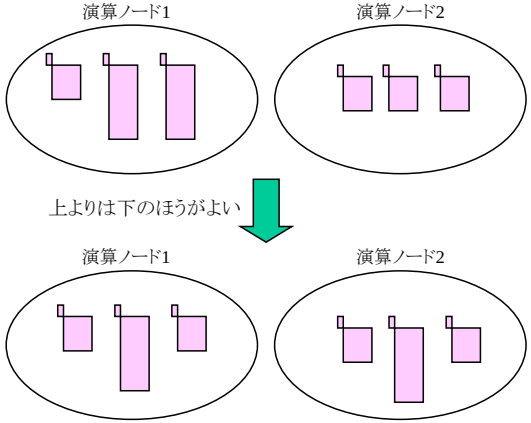
21

プロセッサ負荷の分散の模式図



22

メモリ負荷の分散の模式図



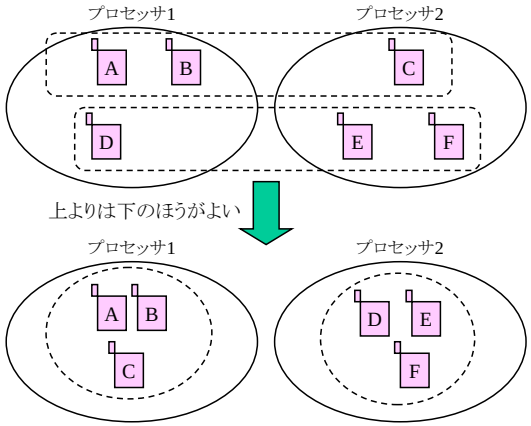
23

プロセッサ間通信の削減

- 連携する(=相互に通信を必要とする)プロセスが, 異なるプロセッサにまたがって配置されると…
  - 一連の処理を完了させるまでに, 多数の**プロセッサ間通信**が必要になる.
  - **プロセス間通信**を避けることはできないが, 同一のプロセッサ内であれば, プロセッサ間よりは高速に処理できる.
- 連携するプロセスができるだけ同一プロセッサ上に配置されるようにプロセスの配置を決定する.

24

プロセッサ間通信の削減の模式図



25

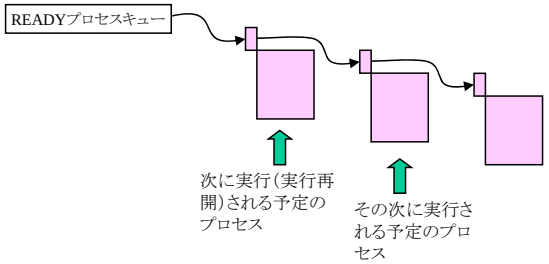
負荷分散・通信量削減はいつ可能か？

- 実は，いつでもできるわけではない．
  - プロセスを起動するときに，どのプロセッサで起動させるかを選ぶことはできる．
    - 不都合なプロセス配置にならないように，あらかじめ防止策をとる．
  - しかし，あるプロセッサ上で起動した後のプロセスを，別のプロセッサに移動させるのは，必ずしも容易ではない．
    - 後で述べるような「**プロセス移動機能**」が別途必要になる．
    - これは結構大変．

26

スケジューリングの機構

- READYプロセスキュー
  - OS内で，実行可能なプロセスを登録したキュー（queue，待ち行列）



27

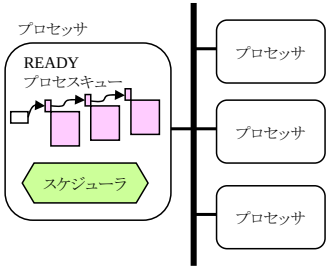
スケジューラとREADYプロセスキューの配置

- 並列処理・分散処理を行うシステムでは，スケジューラやREADYプロセスキューの配置に，いくつかの選択肢がある．
  - スケジューラの配置
    - 全体で1つにするか，各プロセッサに配置するか
  - READYプロセスキューの配置
    - 全体で1つにするか，各プロセッサに配置するか

28

配置方法(1)

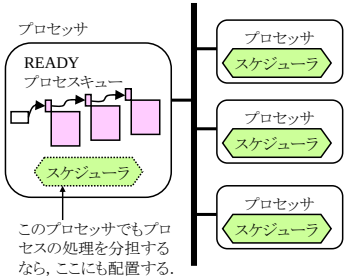
- スケジューラ
  - 全体で1つ
- READYプロセスキュー
  - 全体で1つ
- プロセススケジューリングに必要な情報が1カ所に集中するため，機構が簡単
  - スケジューラは全体の負荷情報が把握できるので，負荷分散や通信量削減がしやすい．
- スケジューリングを行うプロセッサに障害が起きると，システム全体がダウンする．
  - 障害がないときでも，プロセスの配布が大変．



29

配置方法(2)

- スケジューラ
  - 各プロセッサに1つ
- READYプロセスキュー
  - 全体で1つ
- READYプロセスキューが1カ所にあるため，負荷分散や通信量削減は比較的しやすい．
  - ただし，大半のキュー操作にはプロセッサ間通信が発生する．
- ほとんどのプロセッサの不具合に強いが，READYプロセスキューのあるプロセッサに障害が起きると，システム全体がダウンする．
- プロセスの配布も(1)と同様に大変．

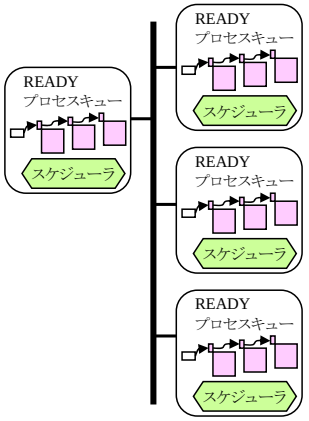


これとは逆に，スケジューラを1つに，キューを各プロセッサに配置する方法は，理論的には可能だが，他の方法に比べて長所がないため，現実的でない．

30

配置方法(3)

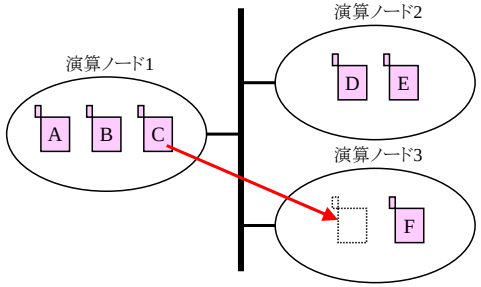
- スケジューラ
  - 各プロセッサに1つ
- READYプロセスキュー
  - 各プロセッサに1つ
- 各プロセッサが対等な機構を有するので，プロセッサの不具合に強い．
- 全体の負荷情報の把握が難しいため，負荷分散や通信量削減は難しい．



31

プロセス移動(process migration)機能

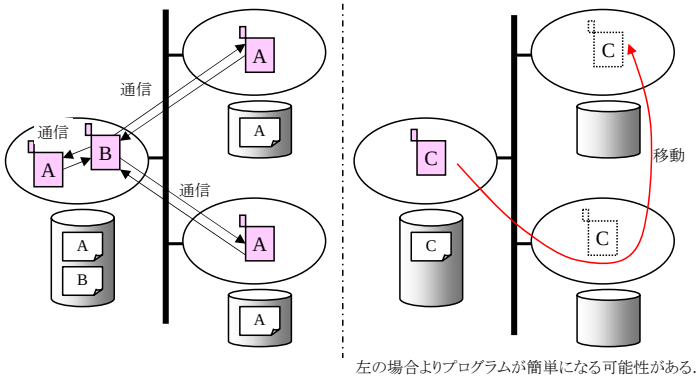
- 走行中のプロセスを，別の演算ノードへ移す機能
  - 目的
    - 負荷分散，プロセッサ間通信の削減
    - 特に，プロセスの起動時にうまく配置できなかったものを実行中に別の演算ノードに移動させる．



32

プロセス移動機能の利用例

- 各演算ノードの状態を監視するプログラムAと，それらを通じて全体の状況を管理するプログラムBがある場合



左の場合よりプログラムが簡単になる可能性がある．
プログラムの格納が1カ所でよい．

プロセス移動が実現できる条件（1）

- 移動先でプロセスの実行が再開できる条件
  - プロセスの移動直前の状況が移動先で再現できること．具体的には，以下の情報を移送し，新しいプロセスに継承させられること．
    - ①**プログラムコード**
    - ②**データ**
    - ③**スタック**（ユーザスタック，カーネルスタック）
    - ④**レジスタ等の値**
    - ⑤**プロセスの属性**
      - プロセス識別子やプロセス優先度など
    - ⑥**利用している資源に関する状態**
      - オープンしているファイルなど

34

プロセス移動が実現できる条件（2）

- 以下の情報は，必ず移送しなければならない．
  - ①**プログラムコード**
    - ただし，移動プロセスと同じプログラムが移動先にあれば，移動は不要
  - ②**データ**
  - ③**スタック**（ユーザスタック，カーネルスタック）
    - ただし，プロセスの移動がシステムコール呼び出し直後に起きる（＝ユーザプロセスの状態はすべてユーザスタックに保存される）ようにすれば，カーネルスタックの移動は不要
  - ④**レジスタ等の値**

35

プロセス移動が実現できる条件（3）

- 以下の情報は，そのまま移送できなくてもよいが，移動先で問題なく動作できるように変換されなければならない．
  - ⑤**プロセスの属性**
    - **プロセス識別子**は，移動先の他のプロセスと衝突してはならない．
      - 移動先でも衝突しないようにあらかじめすべてのプロセスにシステム全域で一意なプロセス識別子を付けることは非常に困難．
      - 一意なプロセス識別子を付けることが不可能なら，移送時に変換が必要．
    - **プロセス優先度**は複数のプロセスが同じ値を持っていてもよいので衝突しても問題はないが，移動先でも適正な値になっている必要がある．
      - 移動先のOSが異なる場合には移送時に変換が必要．

36

プロセス移動が実現できる条件（4）

- ただし，以下の情報は，移動先にまったく同じ環境と資源がなければ無意味である．
  - **⑥利用している資源に関する状態**（オープンしているファイルなど）
- このため，以下のいずれかで対処することになる．
  - 移動先で，資源に対して初期化処理をやり直す．
  - ①～⑤まで以外には当該プロセスが必要とする資源がない，あるいは，あったとしても継続性がなくてもよいという場合には，⑥の情報の移送を行わない．

37

プロセスのスケジューリングについてのまとめ

- プロセスをいつどこで動かすか
  - 演算ノードの負荷の分散
  - プロセス間通信の削減
- スケジューラとREADYプロセスキュー
  - どのように配置するか
- プロセス移動機能
  - プロセスが移動できるための条件

38

次回予告

- 次回（11/28）は演習を行います．
  - 教科書・プリント・ノート等の**持ち込み不可**．
  - **解き終わったら，退出してよい**．
- 解答例の提示，コメント等はその次の回に行います．

39