第7章 負荷平均化と終了検知

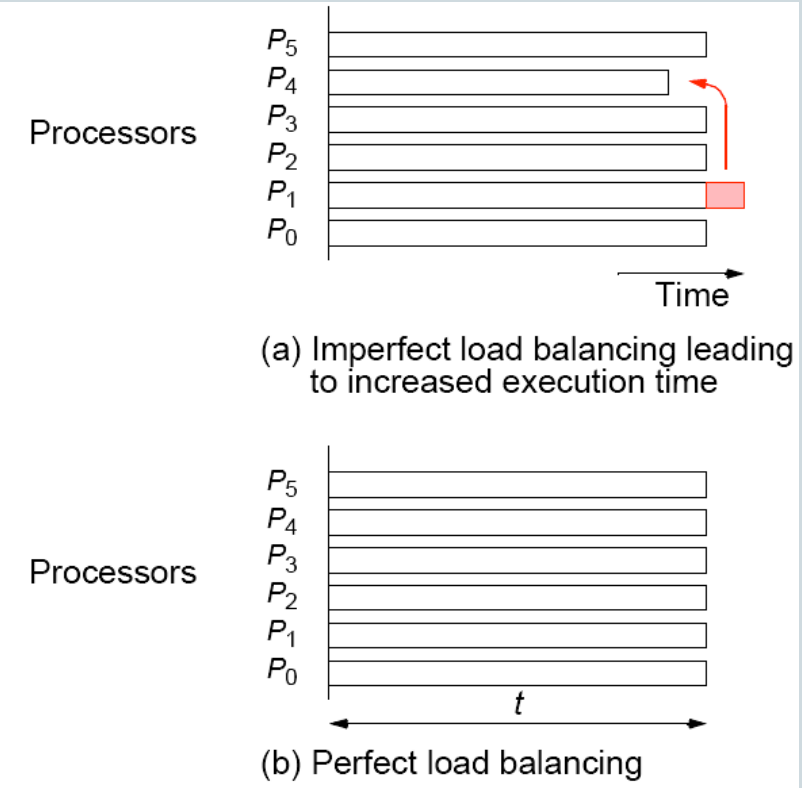
7.1 負荷平均化(1)

プロセッサに均一に仕事を分散することによって最小の実行時間を実現する.

- マンデルブローの例を思いだそう!
- 図は負荷平均化を説明している.

静的負荷平均化

- 予め各部の実行時間の見積りが必要.
- マルチプロセッサスケジューリング
  またはマッピング問題と呼ばれる.
- ある特別なケースを除いてNP困難な
  問題である.



7.2 動的負荷平均化

プログラム実行中にプロセッサにタスクが割り付けられる.

- 次の種類に分類できる.
  - 中央管理型
  - 非中央管理型

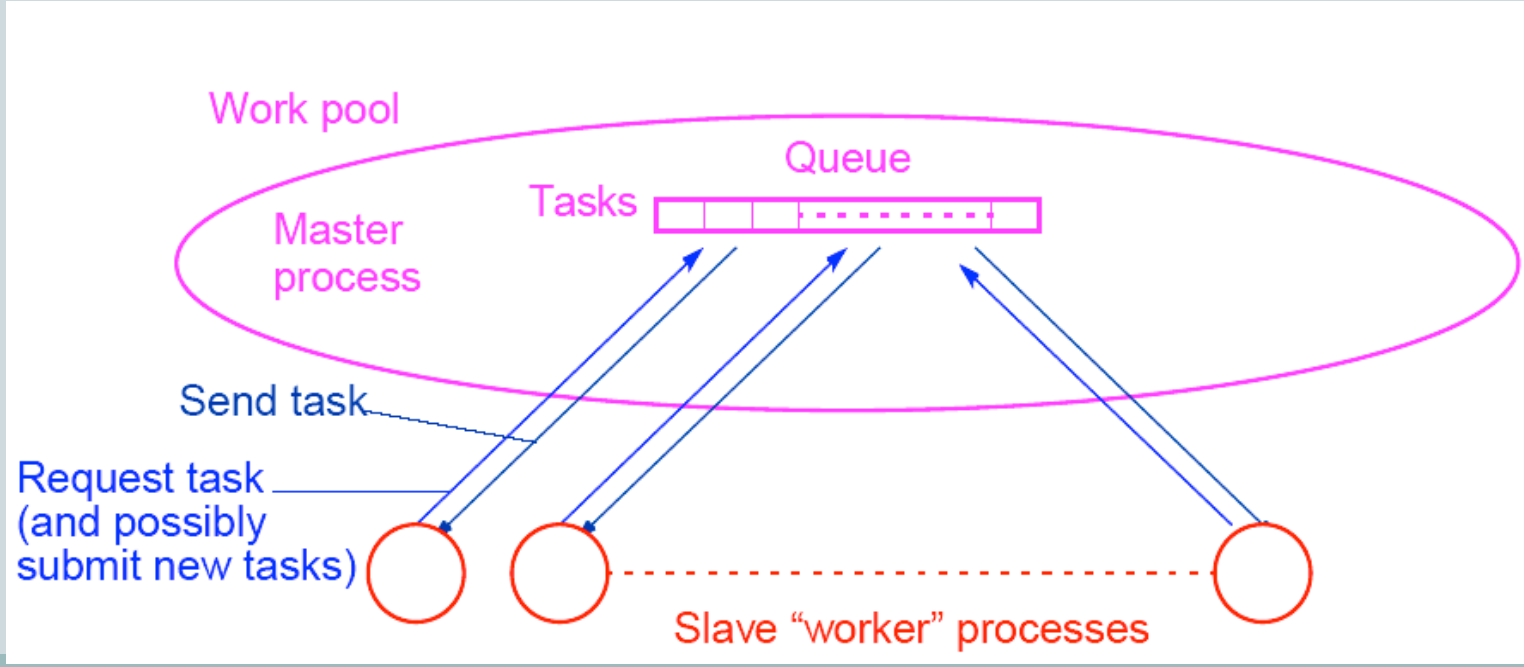
中央管理型動的負荷平均化

- マスタプロセスが処理すべきタスクの集まりを保持している.
- タスクはスレーブプロセスへ送られる.
- スレーブプロセスは一つのタスクの処理を終了すると, マスタプロセスに他のタスクを要求する.

7.2.1 中央管理型動的負荷平均化(1)

ワークプール法と呼ばれる.

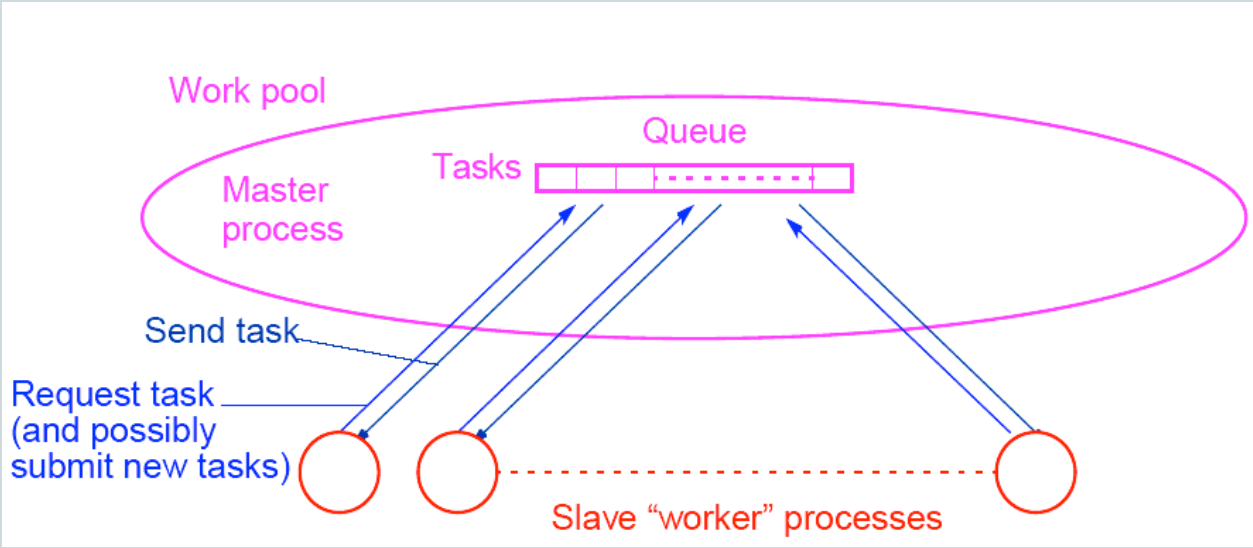
- マンデルブローの例を思い出そう.
- 実行の過程でタスク数が変化するような場合にも適用できる.
- ワークプールのタスク数がゼロになったら終了する.



7.2.1 中央管理型動的負荷平均化(2)

タスクをどの順番で渡したらよいか?

- 一般的に，より大きいかまたは最も複雑なタスクを最初に扱うと良い．なぜ？
- 全てのタスクが同じ大きさで同じ重要度であれば先入れ先出し行列(FIFO)で十分．
- そうでない場合には，優先順序付キュー(ヒープ)を用いるとよい．



7.2.1 中央管理型動的負荷平均化(3)

終了

- タスク待ち行列が空である.
- 各タスクが新しいタスクを生成することなしに次に処理すべきタスクを要求している.

欠点

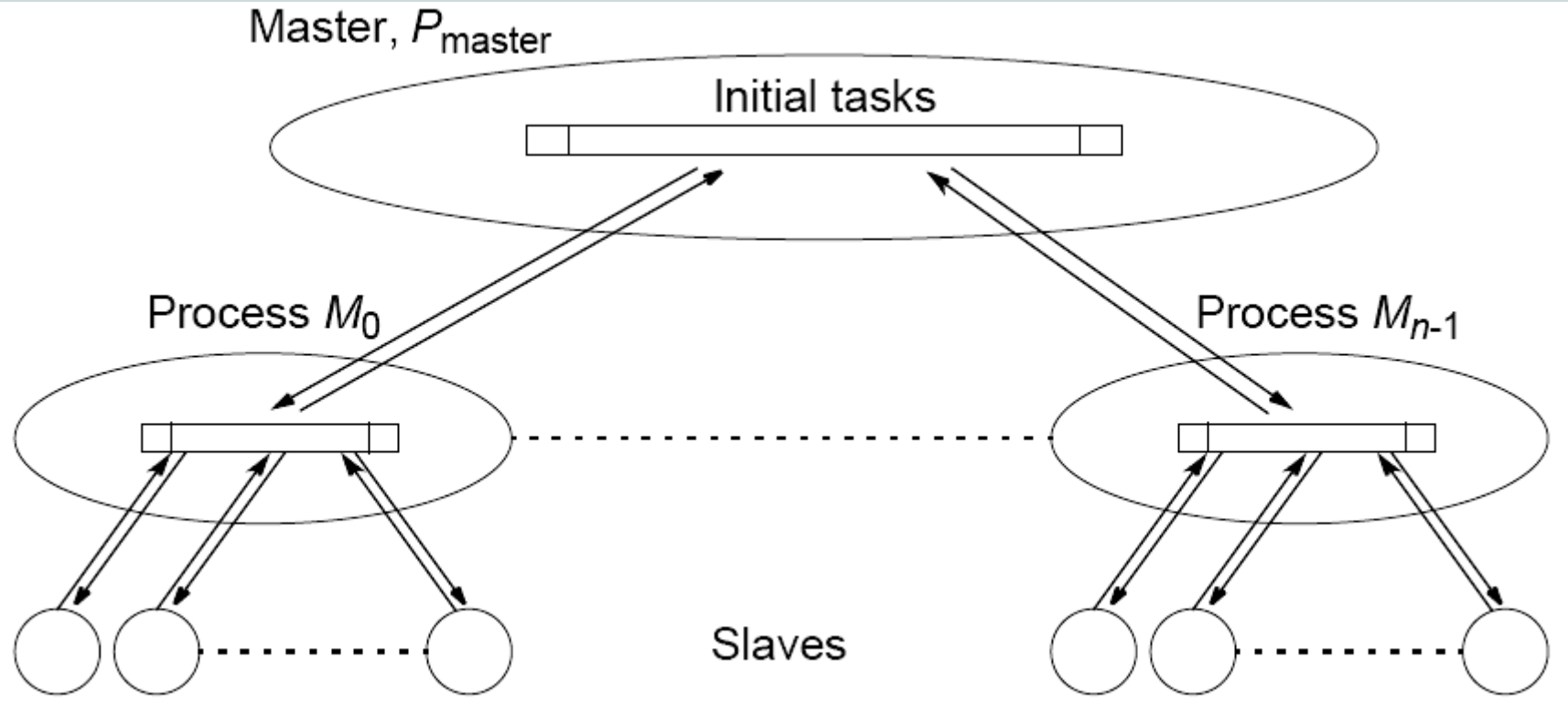
- マスタプロセスは一度には一つのタスクしか起動できない.
- 多くのプロセスが同時に要求したときにボトルネックが生じる可能性が高い.
- スケーラビリティは一般的に良くない.

少ないスレーブで一つのタスクの計算量が大きいときには中央管理型

7.2.2 非中央管理型動的負荷平均化(4)

ワークプールを分散させる.

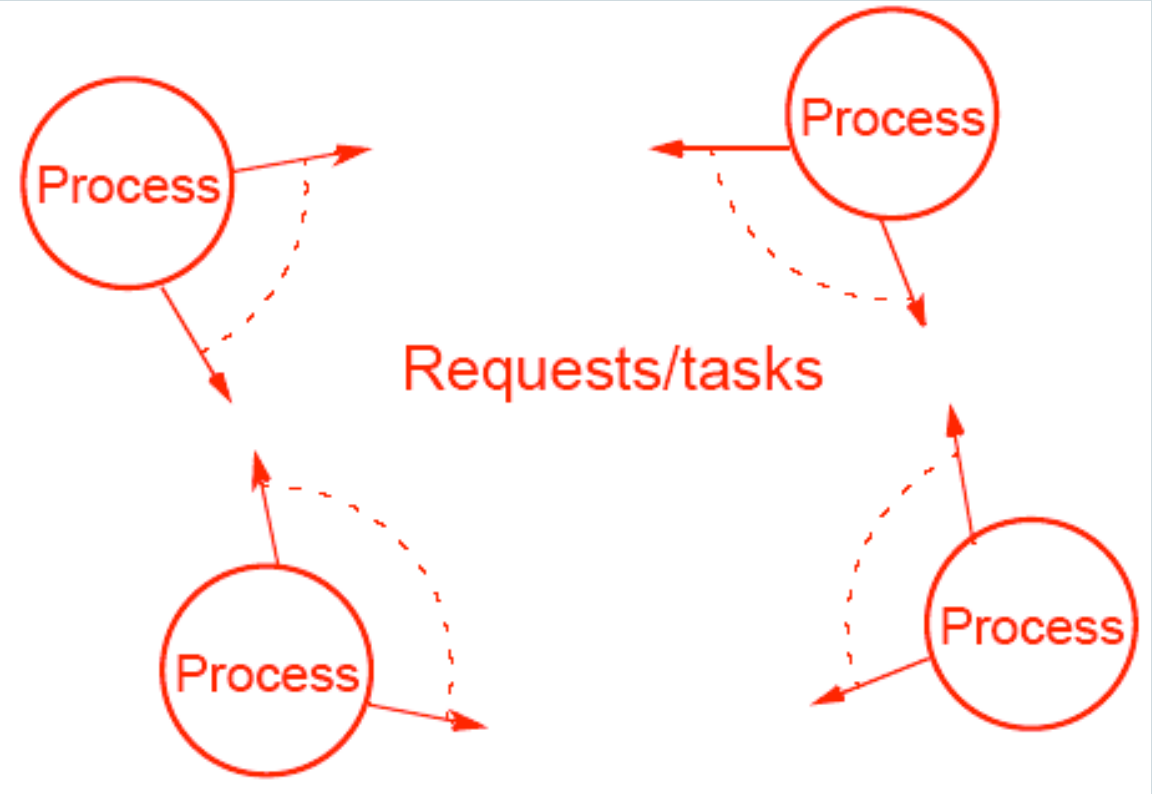
- マスターがミニマスターに分類する.



7.2.2 非中央管理型動的負荷平均化(5)

完全分散型ワークプール

- 平面的に配置され，お互いの話合いによってタスクをやりとりする．



7.2.2 非中央管理型動的負荷平均化(6)

受取り側主導

- 自分が選んだ他のプロセス(プロセッサ)にタスクを要求する.

送り手主導

- 自分の選んだプロセス(プロセッサ)にタスクを送る.
- 高い負荷のプロセス(プロセッサ)が, 自分のタスクの幾つかをそれを受け取ることができるプロセス(プロセッサ)に送る.

7.2.2 非中央管理型動的負荷平均化(7)

プロセスの選択(受取り側主導)

- ラウンドロビンアルゴリズム
  - プロセスP_iがプロセスP_xにタスクを要求する.
  - xは，要求の度にインクリメントされるカウンタの値をnで割ったときの剰余(nはプロセス数)
  - i==xの際には再度インクリメントする.
- ランダムボーリングアルゴリズム
  - xが0からn−1までの間のランダムな値とする.
  - プロセスP_iがプロセスP_xにタスクを要求する.

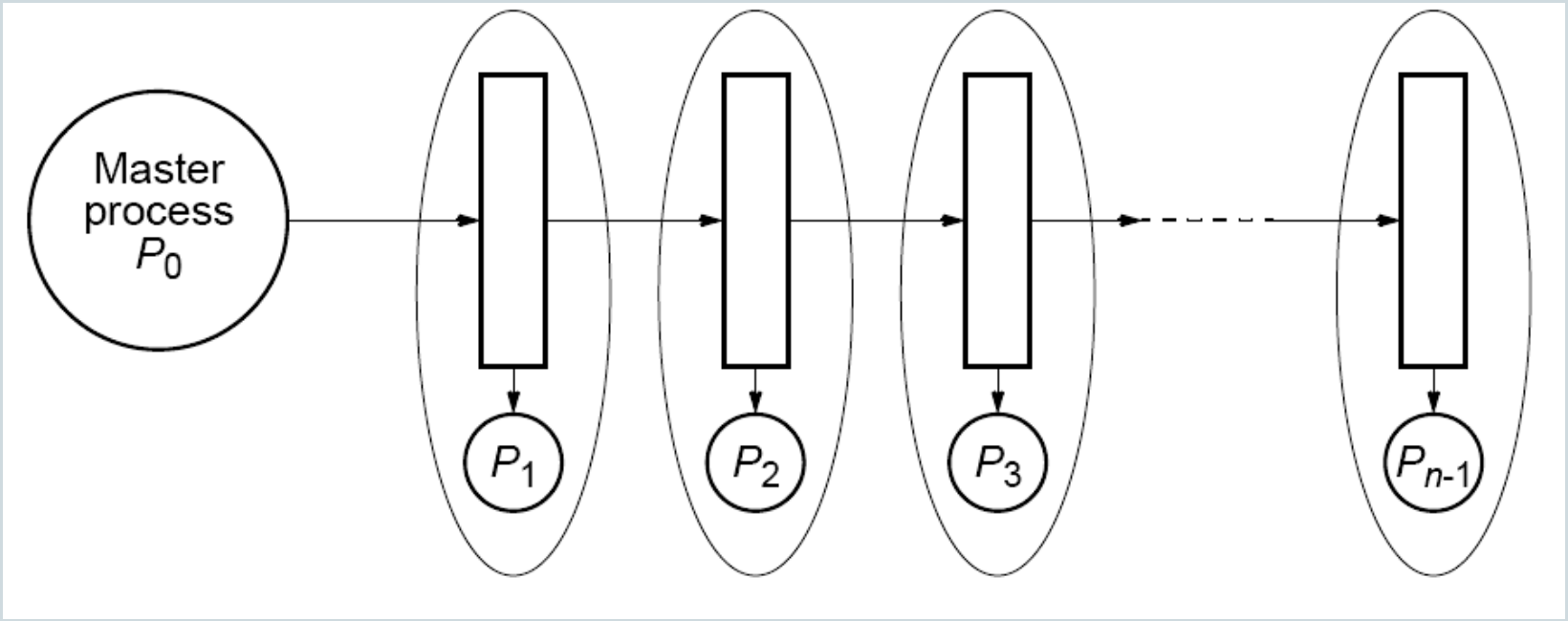
タスクの割当て(受取り側主導)

- まだ着手していないタスクの一部を要求してきたプロセスに送る.
- どのタスクを先に割当てるかには，いろんな戦略がある.

7.2.3 直線構造を利用した負荷平均化(1)

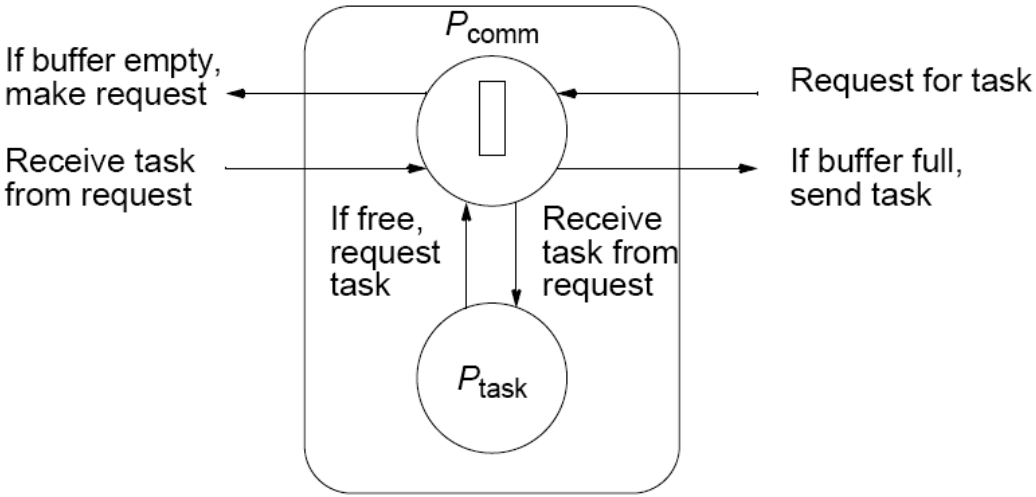
図のように待ち行列を構築する.

- 待ち行列の各要素は一つのプロセスに対応しており, パイプライン構造になっている. 待ち行列の出口が複数あることに注意.



7.2.3 直線構造を利用した負荷平均化(2)

- マスタプロセスは待ち行列の入り口でタスクを挿入する.
  - プロセスがアイドルなら, タスクを待ち行列から取り出す.
  - 左にあるタスクは, 取り除かれたタスクがあった場所を埋めるように詰められる.
  - 新しいタスクが待ち行列の入り口から挿入される.
- 各プロセスは図のような構成になる.
  - 左右の通信を受け持つサブプロセス(スレッド)
  - 実際のタスク処理を受け持つサブプロセス(スレッド)



7.3 分散終了検知アルゴリズム

分散されたタスクをどのように終了させるか?

終了条件

- 時刻tに分散的に終了するには,
  - 時間tでプロセス全体に対して, そのアプリケーション独自の局所終了条件が存在する.
  - 時刻tで, プロセス間で送信中のメッセージがない.

2番目の条件はなぜ必要か?

- 1番目の条件は比較的認識しやすい. 各プロセスが局所条件を満足したかどうか判断して, そうであれば, マスターにその旨を伝える.
- 2番目の条件はどうか?
  - メッセージがプロセス間を移動する時間を知ることは困難である.
  - 送信中のメッセージが移動し終えるのに十分な時間だけ待つ?
  - システム依存なのであまり好まれない.

7.3.2 応答メッセージの利用(1)

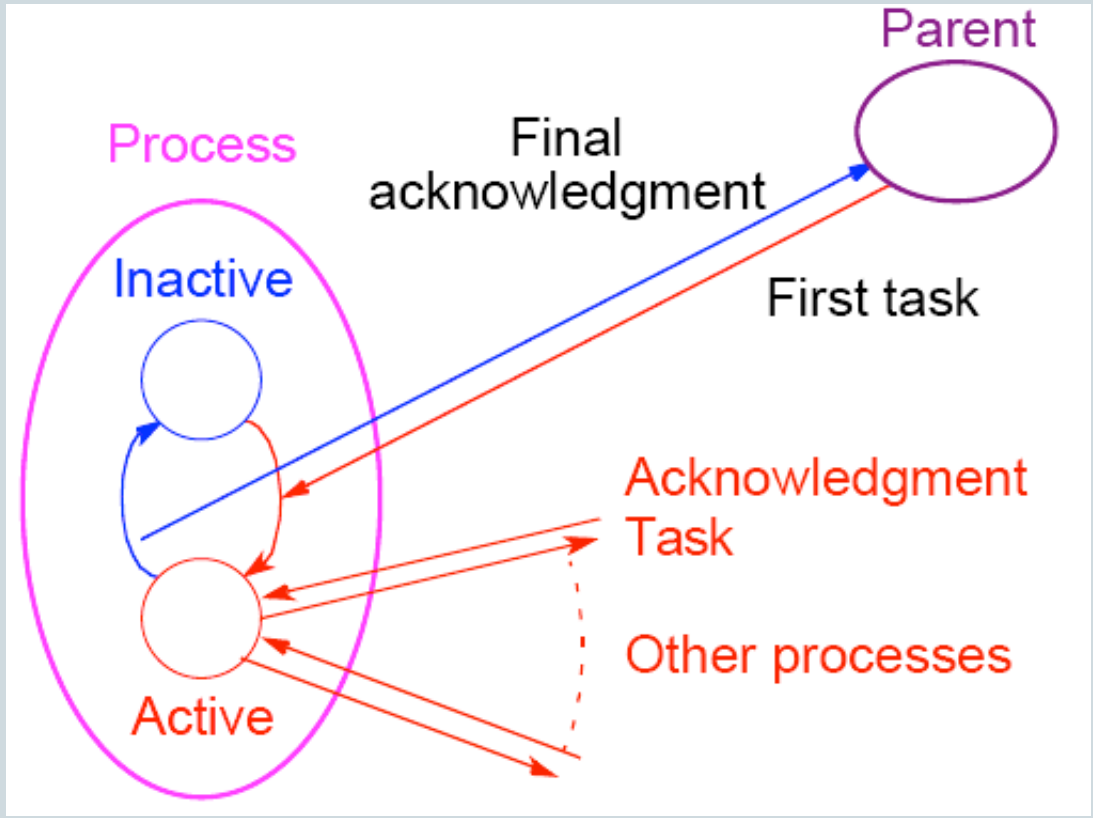
活性，不活性と親子関係

- 各プロセスは，「不活性」あるいは「活性」である．
- 最初は，プロセスは「不活性」である．
- 別のプロセスからタスクを受け取ったとたんに活性状態になる．
- あるプロセスを不活性から活性にしたプロセスをそのプロセスの親という．
- 活性プロセスは，さらに別のプロセスからタスクをもらうことができる．ただし，その場合は親子関係は作られない．

確認メッセージ

- タスクをもらったプロセスは受取り確認メッセージを送信する．
- 親プロセスからタスクをもらった時以外は，タスクをもらった時点でただちに確認メッセージを送信する．

7.3.2 応答メッセージの利用(2)



7.3.2 応答メッセージの利用(3)

不活性状態への変換

- 不活性状態に戻る条件が成立したときに親プロセスに「確認メッセージ」を送る.

不活性状態に戻る条件

- プロセスの局所終了条件が存在する(全てのタスクは終了している).
- 自分の受け取ったタスクに対する確認メッセージ(親以外)を全て送り終わった.
- 自分が送ったタスクに対する確認メッセージを全て受け取った.

最初のプロセスが終了したら, 計算は終了したことになる.

- 次の二つの条件(前述)は満足しているか, 確認せよ.
  - 時間tでプロセス全体に対して, そのアプリケーション独自の局所終了条件が存在する.
  - 時刻tで, プロセス間で送信中のメッセージがない.