システムプログラム概論

プロセス

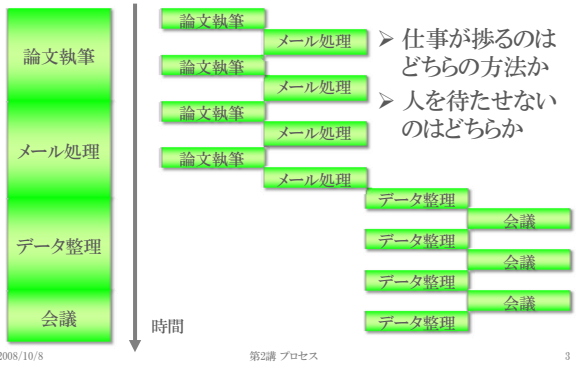
第2講: 平成20年10月8日 (水) 2限 S1教室

中村 嘉隆(なかむら よしたか)
奈良先端科学技術大学院大学 助教
y-nakamr@is.naist.jp
http://narayama.naist.jp/~y-nakamr/

今日の講義概要

- マルチプログラミング
- プロセスの管理
- スケジューリング方式

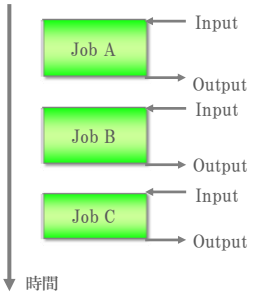
複数の仕事を処理する二つの方法



- 仕事が捗るのはどちらの方法か
- 人を待たせないのはどちらか

スケジューリング(1):ジョブスケジューラ

- 今日ではほとんど使われないが, スーパーコンピュータ等で残っている
- バッチ処理(不在処理)
  - Ex. 何時何分から○○の処理をする
- NQS, LSF 等

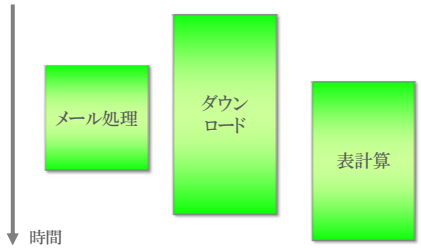


マルチプログラミング

- 現在のコンピュータは, 同時に複数の仕事を行う
  - **マルチプログラミング**
    - CPU が実行するプログラムを切り替える(数十～数百ミリ秒)
    - 擬似並行処理(pseudoparallelism)
      - ある時点で CPU は一つのプログラムを実行
      - ユーザは並行処理をしていると錯覚
  - **マルチプロセッサ**
    - ハードウェアにより並行処理
    - 複数の CPU が物理メモリを共有

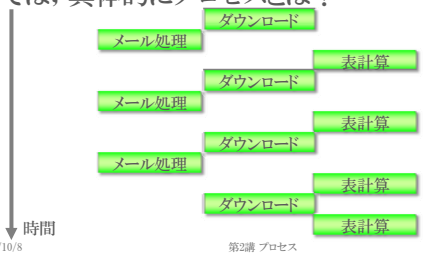
マルチプログラミング

- **プロセス**とは, 並行処理を扱いやすくするための概念モデル



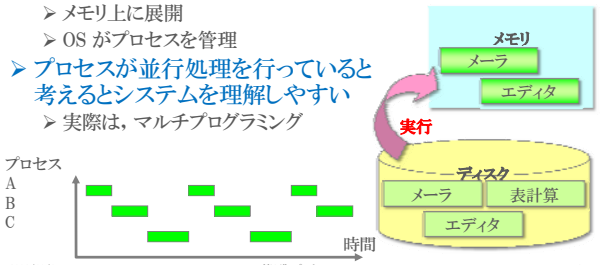
スケジューリング(2):プロセススケジューラ

- マルチプログラミングを実現する基本的仕組み
- 対話型処理では必須
- では,具体的にプロセスとは?



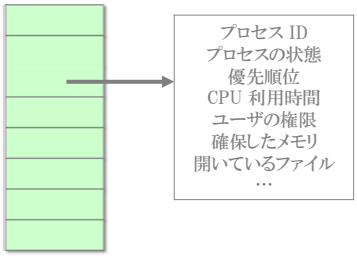
プロセス

- プロセスとは **実行中のプログラム**
  - プログラムカウンタ,レジスタ,変数の現在値を保持
  - 概念的には,個々のプロセスが仮想 CPU を持つ
  - メモリ上に展開
  - OS がプロセスを管理
- プロセスが並行処理を行っていると考えるとシステムを理解しやすい
  - 実際は,マルチプログラミング



プロセスの管理

- OS はプロセス一覧を保持
  - プロセステーブル



プロセスの生成

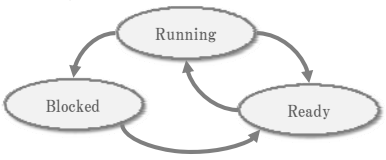
- 基本イベント
  - メモリ領域の確保
  - ディスクからプログラムを読み込みメモリ上へ展開
  - プロセステーブルへ新規プロセスの登録
- プロセスの生成:システムコールを使用
  - UNIX: `fork` システムコール
    - 呼び出し側プロセスのクローンを作成
    - 子プロセスは `execve` システムコール(指定した実行可能プログラムをロードして実行)を呼び出し,環境構築
  - Windows: `CreateProcess` システムコール
    - プロセスの作成とプログラムのロード
  - 親と子プロセスは個別のメモリアドレス空間を持つ

プロセスの終了

- 基本イベント
  - 当該プロセスが使っていたメモリ領域の開放
  - プロセステーブルからの削除
- プロセスが終了する状況
  - 通常終了(自主的)
    - UNIX: `exit`, Windows: `ExitProcess` システムコール
  - エラー終了(自主的)
  - 致命的エラー(非自主的)
  - 他のプロセスにより `kill`

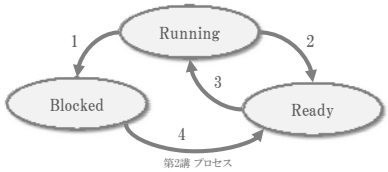
プロセスの状態

- 待ち状態 (blocked)
  - 原理的にプロセスを実行できない(入力待ちなど)
- 実行可能状態 (ready)
  - 実行可能だが,一時的に CPU が割り当てられていない
- 実行状態 (running)
  - 現に CPU を使用している



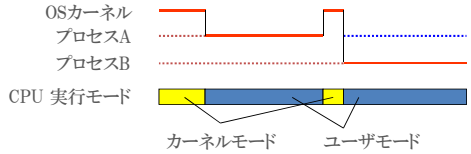
プロセスの状態の遷移

1. プロセスが実行を継続できない
2. 実行中のプロセスが十分な時間実行された
   - スケジューラが判断
3. 他のプロセスが十分な時間実行された
4. 外部イベントが発生した（割り込み）



カーネルモードとユーザモード

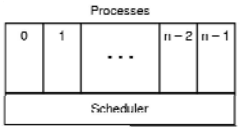
- I/O 処理によるブロック，割込時にプロセスの切替を行いたい
  - ✓ I/O 処理は OS を経由させる
  - ✓ プロセスが直接 I/O 処理をできないようにする
- ✓ **カーネルモード**（特権モード）：全ての機械語命令が実行可能
- ✓ **ユーザモード**（一般モード）：I/O 命令などの実行を禁止



OS の処理を行うとき→カーネルモード
ユーザプロセスを実行するとき→ユーザモード

プロセスの OS 内での実現（1）

- プロセスの実行→割込み/ブロック→スケジューリング→プロセスの実行→…
- 割り込み/ブロックが発生（ユーザモード→カーネルモード）
  - プロセスの状態：running → ready/blocked



1. Hardware stacks program counter, etc.
2. Hardware loads new program counter from interrupt vector.
3. Assembly language procedure saves registers.
4. Assembly language procedure sets up new stack.
5. C interrupt service runs (typically reads and buffers input).
6. Scheduler decides which process is to run next.
7. C procedure returns to the assembly code.
8. Assembly language procedure starts up new current process.

割込の種類毎のプログラム番地

カーネルモードで実行

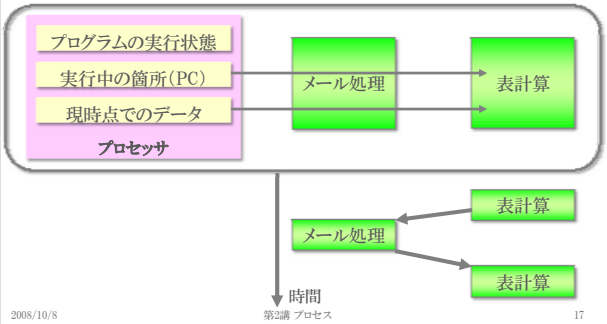
プロセスの OS 内での実現（2）

- OS はプロセスを**プロセステーブル**により管理
  - **プロセス制御ブロック**：プロセステーブル内に保持される各プロセスに関する情報

Process management	Memory management	File management
Registers	Pointer to text segment	Root directory
Program counter	Pointer to data segment	Working directory
Program status word	Pointer to stack segment	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

プロセスの切り替え

- コンテキストスイッチ



プロセス間をどう切り替える？

スケジューリングアルゴリズムに求められる要件

- **公平さ**：各プロセスに CPU 時間を公平に分け与える
- **ポリシーの実行**：決められたポリシーを実行する
- **バランス**：システムの全パートを動作させる
- **スループット**：一定時間内に処理できるジョブ数を最大化する
- **ターンアラウンド時間**：ジョブ投入から実行終了までの平均待ち時間を最小化する
- **CPU 利用効率**：常に CPU を駆動させる
- **応答時間**：処理要求にすばやく応答する
- **デッドライン**：決められた時間までに処理を完了する

上記はトレードオフの関係にあるものもある→目的毎に最適化
例）スループット vs 応答時間

プロセススケジューリング方式(1)

バッチシステム対象のスケジューリング方式

- **FCFS**: First-Come First-Served
  - ジョブが投入された順番に実行する方式
- **SJF**: Shortest Job First
  - 短い処理時間のジョブから先に実行する方式
  - 処理終了までの平均待時間が FCFS より短くなる
- Shortest Remaining Job Next
  - 残りの処理時間が最小のものから先に実行
  - 新規投入されたジョブ A の処理時間が，実行中のジョブ B の残り処理時間より小さいなら，A を実行

FCFS と SJF の例

ターンアラウンド時間
(実行終了までの平均待時間)
＝(8+12+16+20)/4=14分

ターンアラウンド時間
＝(4+8+12+20)/4=11分

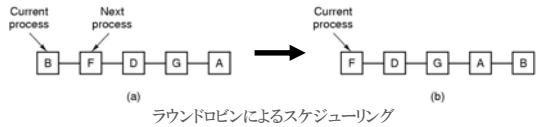
※全てのジョブが同時投入された場合

プロセススケジューリング方式(2)

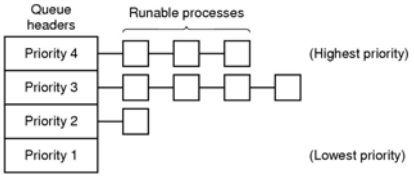
対話型システムを対象にしたスケジューリング方式
CPU を利用できる時間を一定の時間間隔(**quantum**)に分け，複数プロセスに順に割り当てることで疑似並行処理を実現

- **ラウンドロビン**(RR: Round-Robin)
  - 全てのプロセスに公平に CPU を割当て
  - 各プロセスは quantum の分だけ処理を行う→時間経過後(または処理終了/ブロックで)他のプロセスに切り替え
  - 実現方法は簡単→実行可能プロセスのリストを保持
- **優先度順スケジューリング**
  - プロセスに優先度(priority)をつけ，最も高い優先度を持つプロセスを実行
  - 同じ優先度のプロセス同士ではラウンドロビン
  - 問題点：低い優先度のプロセスが全く実行されない状態(飢餓状態(starvation))に陥る可能性がある

RR と優先度順スケジューリングの例



ラウンドロビンによるスケジューリング



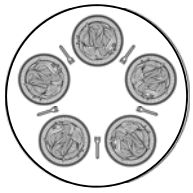
優先度順スケジューリング

Quantum の適切な長さ

- **コンテキストスイッチのオーバヘッド**
  - 各種レジスタ，メモリマップ，各種テーブルの保存・復元などの処理が必要
  - コンテキストスイッチの時間が 1 ミリ秒，quantum が 4 ミリ秒だとしたら？→ CPU 時間の 20% が無駄に使用
  - quantum=100 ミリ秒としたら，オーバヘッドは 1% だが，10 個のプロセスがある時に，応答時間が最大 1 秒に...
- **quantum の適切な長さ**
  - コンテキストスイッチに 1 ミリ秒かかる場合，quantum=20～50 ミリ秒が妥当

資源飢餓(Resource Starvation)

- マルチタスクに関連した問題であり，プロセスが必要な資源を永久に獲得できない状況
- **デッドロック**によっても発生
  - 例) ダイクストラの**食事する哲学者の問題**
    - 哲学者の動作：食べる/瞑想する
    - 食べるためには，2 本のフォークが必要
    - 一度には一つのフォークしかとれない
    - 5 人の哲学者が一斉にフォークをとる→1 本のフォークしか取れない→デッドロックが発生



- スケジューリングアルゴリズムに問題がある場合に発生
  - 優先度順スケジューリングで，優先度の低いプロセスに CPU が割り当てられない
  - 優先度を定期的に割り当てなおす，などの方策が必要

優先度の逆転(Priority Inversion)

- スケジューリングにおいて，優先順位の低いプロセスが優先順位の高いプロセスが必要としているリソースを占有しているときに発生する状態
- 原理
  - プロセス A, B, C（優先順位 A, B, C）
  - A は C の結果を必要とする
  - B が C の直前に動作し，C を飢餓状態に陥れる
  - A は最も優先度が高いのにいつまでも実行されない
- 解決例
  - 優先度継承：依存関係のあるプロセスの中で最も高い優先度を継承

第 1 回ミニレポート

- 提出期限：10/20 授業開始時
- 形式：A4 用紙に印刷した物（手書きでもよい）
- 以下のプロセス A～E を，FCFS，SJF，RR でスケジューリングする
  (1) A-E のターンアラウンド時間を求め，比較せよ
  (2) C-E のみのターンアラウンド時間を求め，比較せよ

  ※同じ投入時刻の物は A，B，C，D，E の投入順と考えよ
  ※RR の切り替えオーバヘッドは無視してよい．またquantum は 0.5 とせよ

プロセスA: 投入時間 0, 処理時間 4
プロセスB: 投入時間 0, 処理時間 2
プロセスC: 投入時間 3, 処理時間 1
プロセスD: 投入時間 3, 処理時間 1
プロセスE: 投入時間 3, 処理時間 1

まとめ

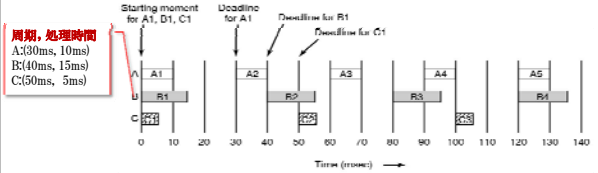
- マルチプログラミング
  - 複数のプログラムを（見かけ上）同時に実行する仕組み
- プロセスの管理
  - プロセスの生成，終了はシステムコールを使用
  - 複数のプロセスの情報をプロセステーブルに保持
- スケジューリング
  - 時分割処理なし：FCFS，SJF
  - 時分割処理あり：RR，優先度スケジューリング

付録：プロセススケジューリング方式(3)

リアルタイムスケジューリング

- 異なる**デッドライン**・処理内容を持つ複数プロセスが CPU を奪い合う
- 各プロセスがデッドラインに間に合うように調整が必要

デッドラインを持つ周期プロセスの例



■全てのプロセスがスケジューリング可能（デッドライン内に処理を完了）な条件

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1 \qquad P_i: \text{周期}, \ C_i: \text{処理時間}$$

Rate Monotonic Scheduling (RMS)

周期的に動作するプロセスのみに使用できる

- アルゴリズム
  - 各プロセスに優先度（=1/周期）を与える
  - 常に最高優先度を持つプロセスを実行
  - あるプロセスの処理中に，もっと高い優先度を持つプロセスが実行可能になると，そのプロセスが CPU を横取りする
- 以下の条件を満たすプロセスに対し使用可能
  1. 各周期プロセスは周期内に処理を終える必要がある
  2. 各周期プロセスは他のプロセスに依存していない
  3. 各周期の処理に必要な CPU 時間は同じ
  4. 周期的でないプロセスはデッドラインを持たない
  5. CPU の横取り（preemption）はオーバヘッドなしで可能

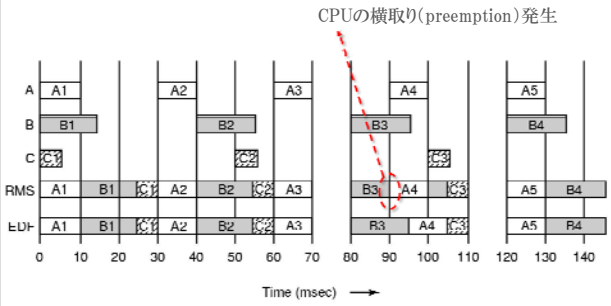
Earliest Deadline First (EDF)

任意のデッドラインを持つプロセスに有効

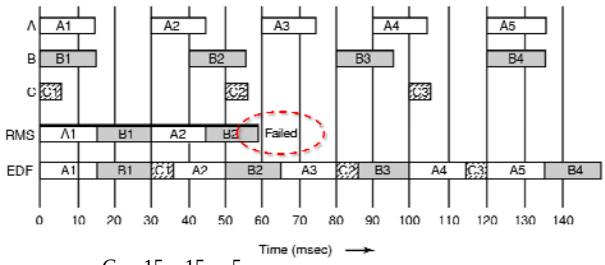
周期的でなくても良い

- アルゴリズム
  - 各プロセスは処理を要求する時に**デッドライン**を申告
  - スケジューラは処理待ちのプロセスをデッドラインが早い順に整列
  - 最も早いデッドラインを持つプロセスに処理が移る
  - 新しく処理要求を行ったプロセスが最も早いデッドラインを持つ時は，現在実行中のプロセスから CPU を横取りする

RMS と EDF によるスケジューリング例(1)



RMS と EDF によるスケジューリング例(2)



$\sum_{i=1}^{3} \frac{C_i}{P_i} = \frac{15}{30} + \frac{15}{40} + \frac{5}{50} = 0.975$ なので，スケジュール可能なハズ...

プロセス数 =3 の場合，RMS では CPU 利用率 78% しか達成できない(EDF は 100%)