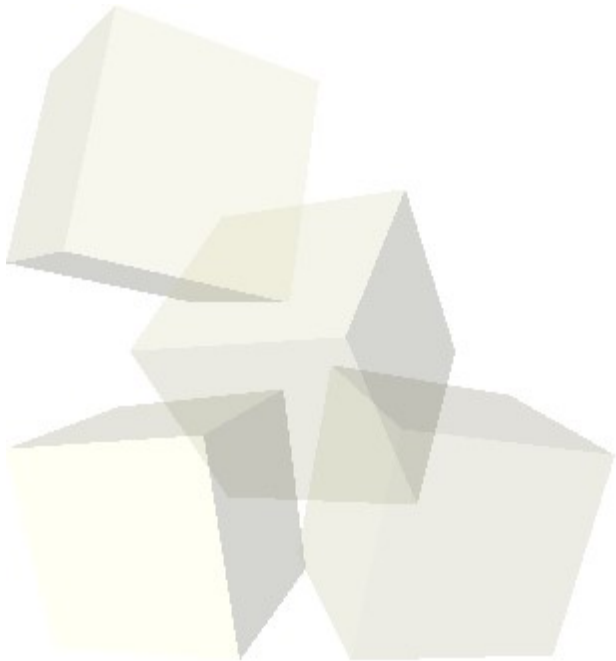




オペレーティングシステム

#1 序論





OSの授業がなぜ難しく感じるか？

- 皆さんは今まで正解を覚える形でしか勉強をしていない
- オペレーティング“システム”は、いろいろな正解の組み合わせがあるので、なにが正解ということはない
- 全てのシステム形態には、良いところと、悪いところが存在する
 - 例えば性能はいいけど、高価だとか
- 従って、物事の本質を正しく理解する必要がある
- さらに、始めて聞く言葉が頻出する
 - 従って、予習、復習は必須。言葉を覚えるのは理解の最初の一步
 - 決して言葉を覚えるのが目的ではない



- 計算機の基本周期(クロック)は？
 - 1単位の処理時間 10^{-9} 秒(1ns)、クロックだと 10^9 回(1GHz)
- 主記憶へのアクセス遅延は？
 - 10^{-7} 秒(100ns)
- 2次記憶(ハードディスク)へのアクセス遅延は？
 - 10^{-2} 秒(5ms～10ms)
- 人間のキーボード入力速度は？
 - 早くても 10^{-1} 秒

この桁の違いを、頭に入れること。もしピンとこないのなら、
円を付けて考えればいい

10^9 円=10億円、 10^1 円=10円



■ G(ギガ)

- 10の9乗 = 10億

■ M(メガ)

- 10の6乗 = 100万

■ K(キロ)

- 10の3乗 = 1000

■ n(ナノ)

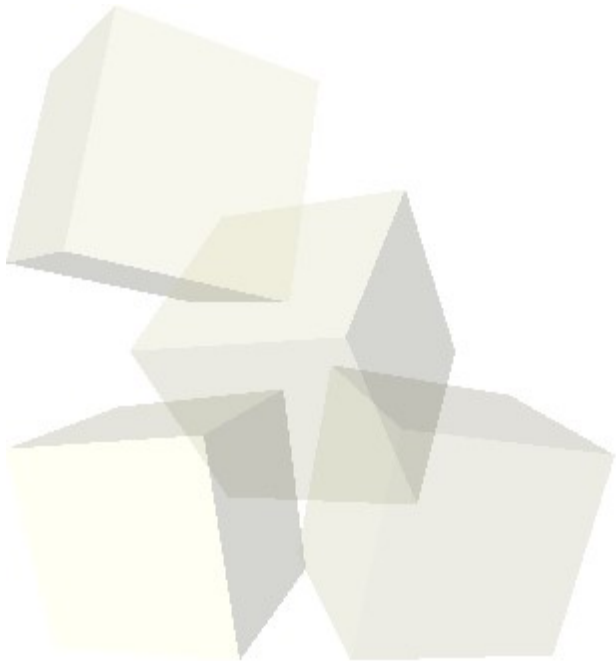
■ μ (マイクロ)

■ m(ミリ)

それぞれ1000倍違う。実社会で1000倍違う状況を想像すること



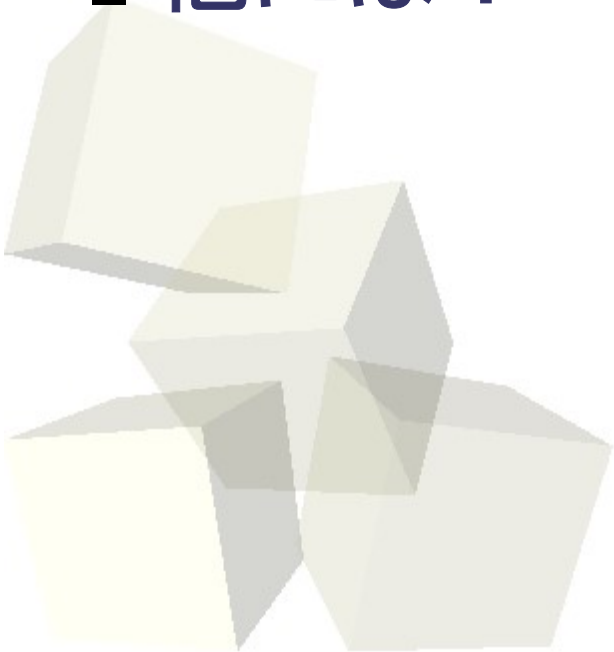
1.1 OSの役割



- Windows系

- Mac系

- 他には？



■ Windows系

- Windows10 ...



■ MacOS系

- MacOS X...



■ UNIX系

- Solaris, AIX ... 昔はたくさんの派生あり



■ 似非UNIX系

- Linux, FreeBSD, OpenBSD, NetBSD, ...



■ 他には？

知っているOSを挙げてみよう

■ パソコン・ワークステーションだけじゃない

■ IOS, Android

- スマートフォン



■ TRON(The Realtime Operating system Nucleus)系

- 自動車, ナビ, 炊飯器, ...

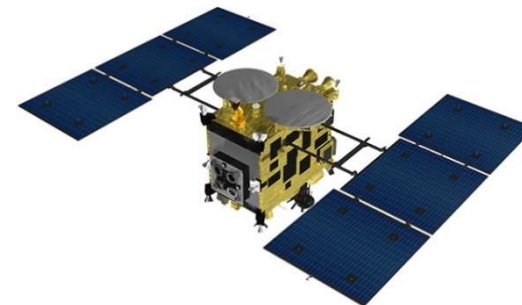


■ ほかにも

- はやぶさ1, はやぶさ2

→ データ処理用 μ ITRON

→ 姿勢軌道制御 VxWorks





■ OSを必要とするのは

- 電子機器(大小を問わず)

■ これら電子機器の特徴

- 入力デバイスがある
 - キーボード, タッチパネル, ボタン, テンキー
- 出力デバイス(表示系)がある
 - ディスプレイ, コンソール
- 記憶領域がある
 - ハードディスク, シリコンメモリ

■ つまり「様々なハードウェアで構成されている」



OSの機能とそれが提供するもの

- 機能

- ハードウェアの抽象化
- ハードウェアの資源管理
- 仕事のスケジューリング

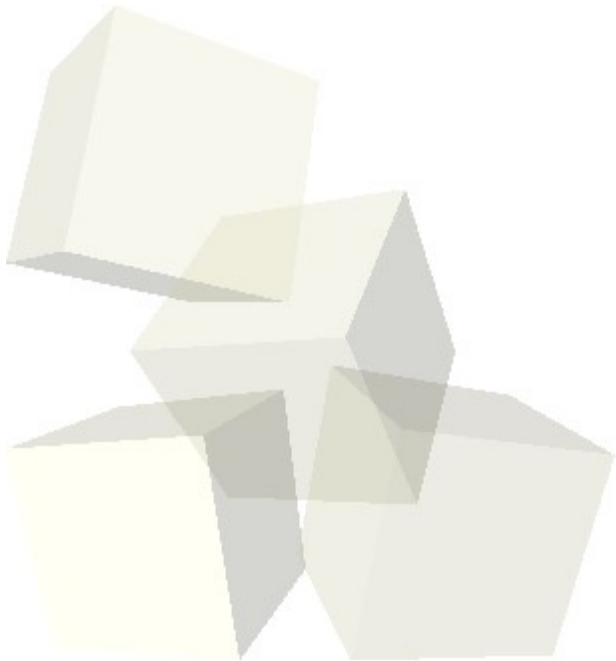
- 提供するもの

- ハードウェアへのアクセスの容易性
- ハードウェア資源の確保に対する確認の容易性
- ハードウェアの利用効率の向上

「ハードウェアに密接に関係」

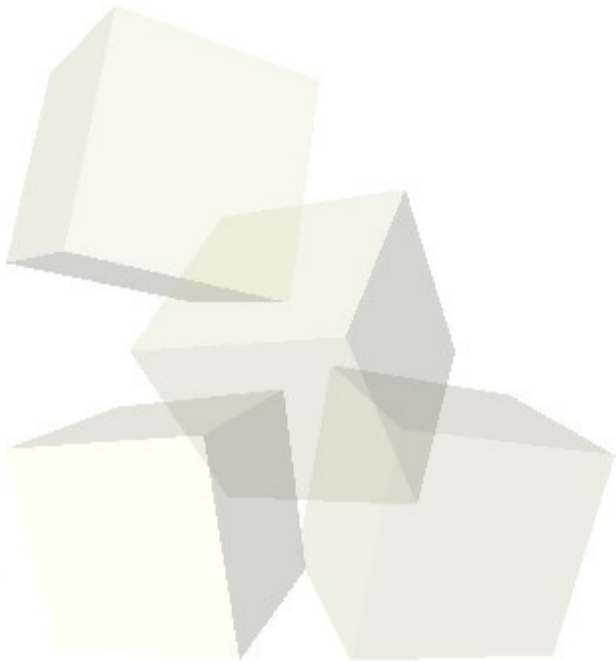


- 1. 抽象化によるアクセス容易性
- 2. 資源管理による確認容易性
- 3. スケジューリングによる効率





- 1. 抽象化によるアクセス容易性
- 2. 資源管理による確認容易性
- 3. スケジューリングによる効率



1. 抽象化によるアクセス容易性

- 例) ディスプレイに文字列を表示
 - ・ 入力された2つの数字の加算結果を表示
- C言語では...

```
#include <stdio.h>
void main(){
    int i, j;
    scanf("%d %d", &i, &j);
    printf("%d¥n", i+j);
}
```

- ・ 画面の大きさ、入力デバイスの違いなどを考慮せずにプログラムを書ける



1. 抽象化によるアクセス容易性

■ もしOSがなかったら...

- マウスのフォーカスが入力したいプログラムに当たっているか？
- キーボードの種類別に、1文字を読み込むように指示
- 入力の終了を検知(改行キー、決定キーなど)
- 画面制御装置に表示指令
 - ディスプレイの大きさは？
 - 表示するべき位置(座標)は？

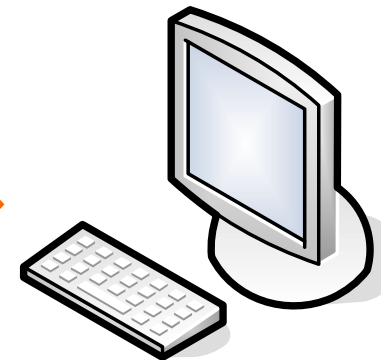
OSがこれらを自動的に処理してくれている



1. 抽象化によるアクセス容易性

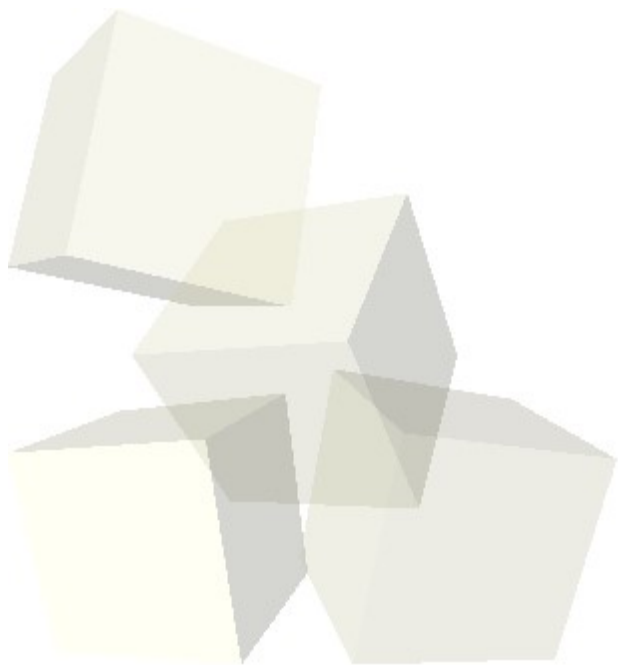
■ つまり、OSが

- ・ 携帯電話の画面や、パソコンのディスプレイなどは「表示装置 (UNIXではstdout)」というものに
- ・ 携帯電話のテンキーや、パソコンのキーボードは「入力装置 (UNIXではstdin)」というものに
- ・ 「**抽象化**」してくれることで、
私たち (主にプログラマ) は個々の機器に対する深い専門知識がなくてもプログラムを書くことができる (**アクセスが容易**になっている)





- 1. 抽象化によるアクセス容易性
- 2. 資源管理による確認容易性
- 3. スケジューリングによる効率





2. 資源管理による確認容易性

- 一般に、機器上では複数プログラムが同時に実行
(マルチタスク)
- 例) 携帯電話で予定表を見ながらメール書き
 - 予定管理プログラム(カレンダー)
 - メール
 - 画面の片隅には時刻表示(時計)
 - 電波状況の表示
 - 着信待ち
 - メール着信チェック
 - etc, etc, etc....
 - 我々が考える以上のプログラムが動いている!





2. 資源管理による確認容易性

■ 携帯電話のメールプログラムを作ることを想像してみよう

現在のスマホには複数CPU(コア)が搭載されているが、ここでは話を単純にするために1CPUとする。

- ・ 携帯電話にはCPUが1つしかない
- ・ 記憶領域(メモリ)にも限りがある
- ・ つまり「**ハードウェア資源は有限**」



■ 何かの処理のたびに空きをチェックしては大変！

- ・ CPUを使いたい、CPUは空いているか？
(他の同時実行プログラムが使っていないか)
- ・ メモリを使いたい、メモリに空きはあるか？
(他の同時実行プログラムが使い切っていないか)

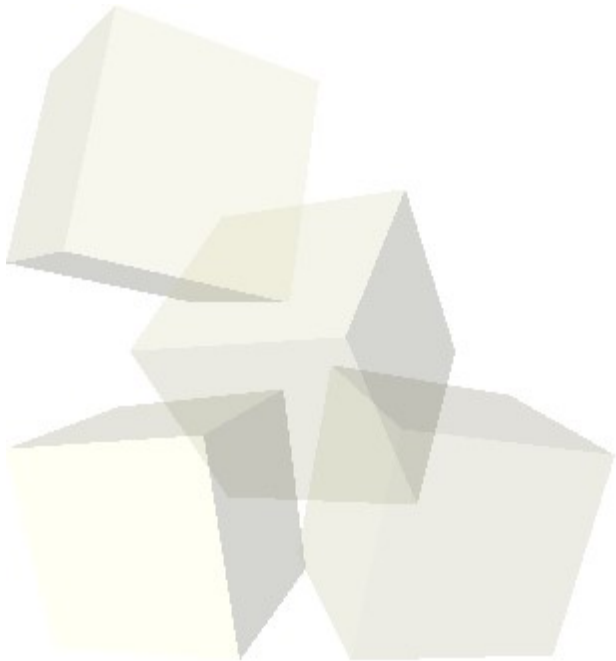


2. 資源管理による確認容易性

■ そこでOSは、

- CPUが無限個あるように
 - メモリの大きさが無限大であるように
 - **見せかけて**くれる。このことで、
-
- プログラム側はハードウェア資源を利用できるかどうかのチェックをしなくてよい(**確認容易性**)
 - 常に「利用できる」と仮定して動作すればよい
 - 実際、複数プログラムが同時に資源を使おうとした場合には、OSがそれらを調停してくれる(**資源管理**)

- 1. 抽象化によるアクセス容易性
- 2. 資源管理による確認容易性
- 3. スケジューリングによる効率





3. スケジューリングによる効率

■ 2. 資源管理によって

- ・ ハードウェアリソースを無限に見せる
- ・ でも実際は有限
- ・ 複数プログラムが同時に使おうとするとき、調停が必要

■ スケジューリング

- ・ プログラムに対する、リソースの割り当ての仕方

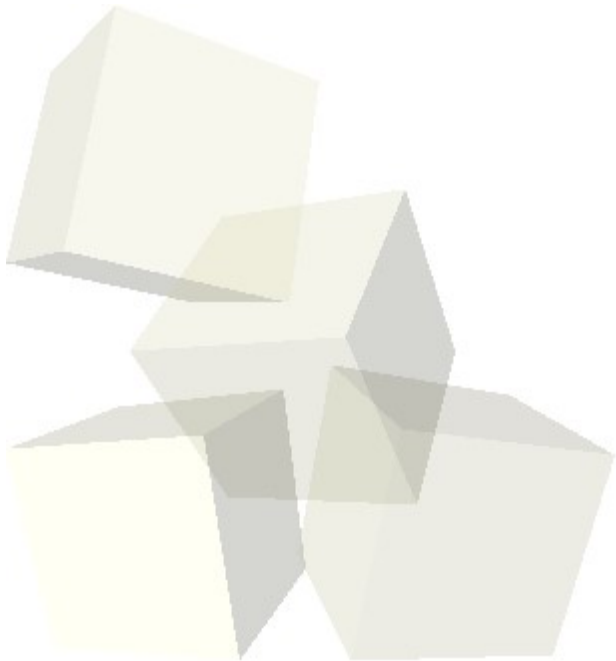
3. スケジューリングによる効率

- スケジューリングが実行効率に影響
- 例) 3つの窓口で30人のお客を処理
 - 窓口それぞれに列を作って並ぶと...
 - たまたま遅い客の後ろに並んだ人は損をする
 - 全体としても、30人を処理し終わる時間が遅くなる
 - スケジューリングが悪い
 - 1列に並んで順に空いた窓口を使うほうがよい
- 機器が処理する仕事のスケジューリングは、もう少し複雑



■ ここからは

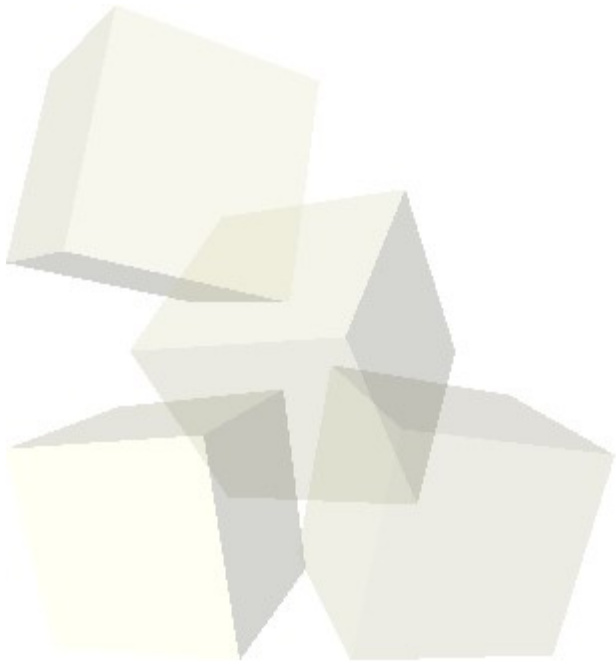
- ・ 2. ハードウェアリソースの仮想化
- ・ 3. スケジューリング
- ・ について少し詳しく





1.2

ハードウェアリソースの仮想化 と スケジューリング





■ 多重化

- 有限のリソース(資源)を、
仮想的に実際より多く見せること
- たくさんのプログラムで使えるようにする

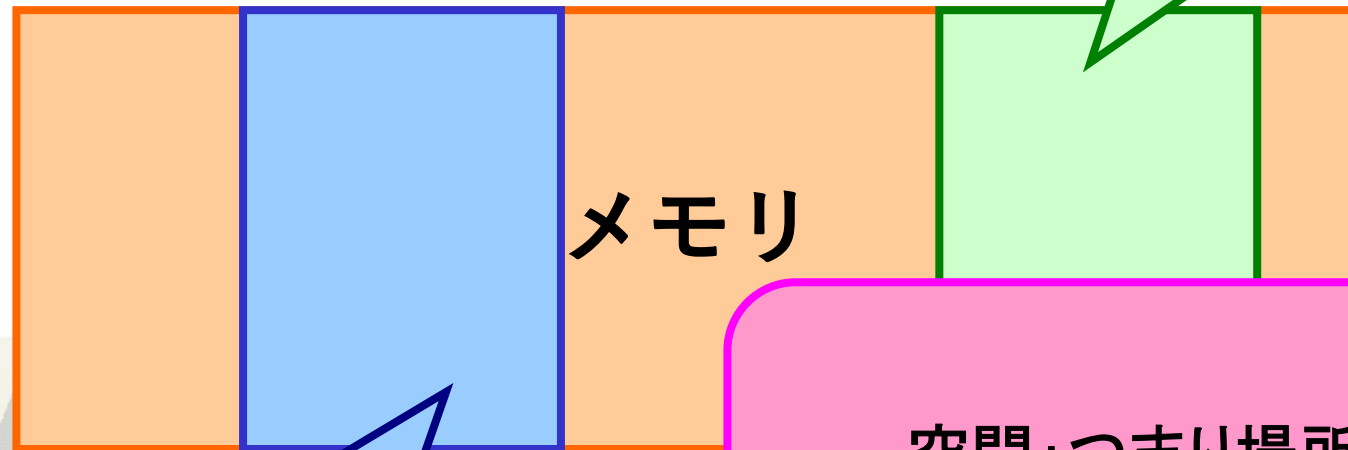
■ 空間分割による多重化

■ 時分割による多重化



■ ハードウェアリソースを複数領域に区切る

- あるプログラムにメモリの特定番地から特定番地までしか使わせない
- これによって競合を防げる



プログラムAが
使用

プログラムBが
使用

空間：つまり場所ごとに
誰が使用するかを決める



■ 複数プログラムが交互に使う

- CPUなど空間分割は不可能



■ 例) 歩行者天国

- 特定の曜日は歩行者専用道路として歩行者が使う
- それ以外の曜日は自動車道として自動車が使う

■ 切り替えオーバヘッド

- 使用者が切り替わる時に**無駄(オーバヘッド)発生**
 - 道の中にいる自動車・歩行者を一旦追い出したり
 - 通行止めの標識を立てたり・撤去したり

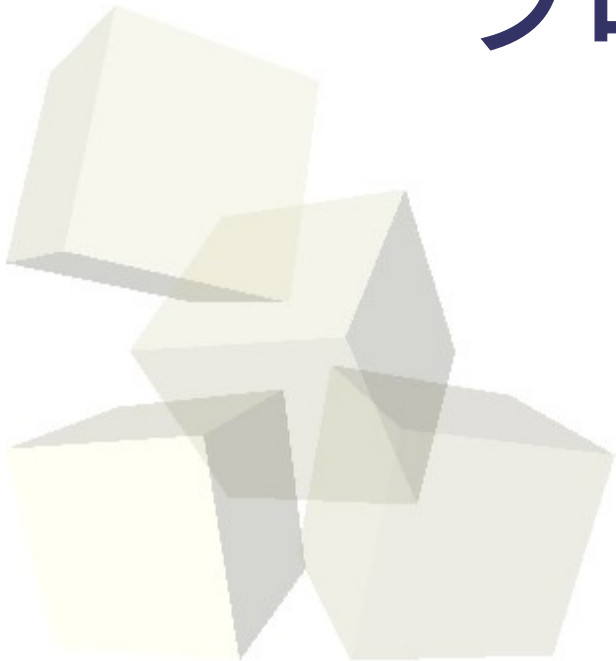


- 電子機器では、もっと短い時間で切り替え
 - 数十ms
- 短い時間で切り替えるということは...
 - 頻繁に切り替え(切り替え回数増加)
 - つまりオーバヘッドも増加
 - いかに**効率よい順番**でリソースを割り振るかが重要



1.3

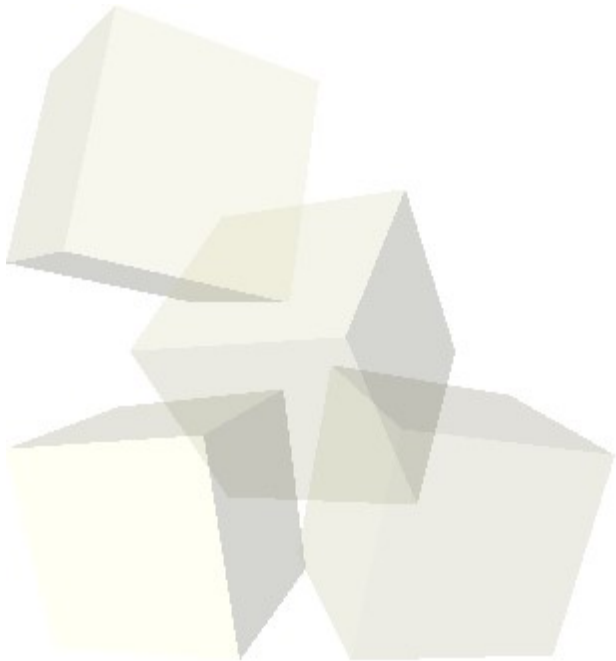
プログラムの処理形態





■ 処理形態

- すなわちスケジューリングについて...





■ ...の前に、単位の説明

■ ジョブ

- ユーザがシステムに対して依頼する仕事の単位

■ プロセス

- システムが処理する仕事の単位
- システムはジョブをプロセスに分割して処理
- リソースはプロセス単位で割り当てられる

■ バッチ処理

- 必要な情報を前もって決定
- 実行してほしいジョブを一括依頼

■ 対話（インタラクティブ）処理

- そのつど、プログラムに対して入力を行う



■ ジョブを一括して依頼

- ユーザは、必要なリソース、実行したいプログラム、処理に使うデータの全てをあらかじめ決定
- それをJCL (Job Control Language) によって記述し、コンピュータに投入

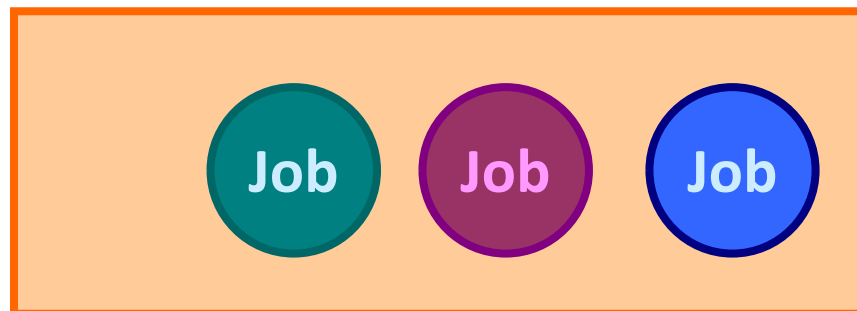
```
//JOB1      JOB (12345), CLASS=X

//STEP1     EXEC PGM=TEST
//DDIN      DD DISP=SHR, DSN=INPUT1
//DDOUT     DD DISP=(NEW, CATLG), DNS=OUTPUT(+1),
//          UNIT=SYSDA, SPACE=(CYL,(15,15),RLSE),DCB+*.DDIN
//          :
```

- 各ジョブは、実行中は全てのリソースを占有する



待ち行列



リソース
(CPU, メモリなど)

- 一定期間ごとに大量のデータをまとめて処理するような場合に便利
 - 例) 売り上げデータ・受注データの集計



■ 利点

- スケジューリングが単純

- 前もって、プログラムが必要とするリソースがわかる
- 複数のジョブのスケジューリングが比較的楽
例) 必要リソースの少ないジョブを優先的に実行すると
全ジョブの平均待ち時間(ターンアラウンドタイム)
が短くなる
- ジョブの切り替えも少なくてすむため、無駄が少ない

■ 欠点

- 前もって全てを決めないとジョブが投入できない



- 必要に応じて人間が入力し、それを処理する
 - 人間の反応速度は 10^{-1} 秒 程度
 - 計算機は 10^{-9} 秒 オーダで命令を処理
 - 人間の入力を待っている時間はすごく無駄
特に昔、計算機が非常に高価だった時代はなおさら
 - 一定時間に処理できる仕事量(スループット)が低下＝もったいない
- タイムシェアリングシステム(TSS)
 - 使っていないCPU時間を他に割り当てよう!



■ 非常に短いCPU時間を複数プロセスに順に割当

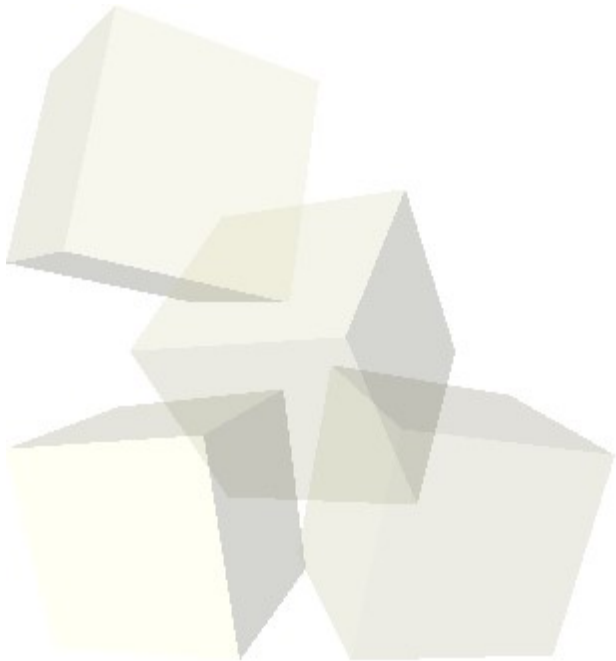
- 割り当てられる単位時間(**クオンタム**)は数十ms
- 割当が一周したらまた最初から

■ プロセスから見ると...

- 細切れのCPU使用权(時間)が、短い間隔を置きながら与えられる
- プロセスは入力待ちなどの遊びも多いので、この「短い間隔」はユーザからは見えにくく、さも**CPUを占有して使っているように見える**

■ 利点

- 自分のほかに大きなプロセスがあっても、そのプロセスが終わるまで長い間待たされたりしない（↑バッチ処理ではありうる）
- ユーザから見ても、対話的に入力してからその反応が返ってくるまでの時間（レスポンスタイム）が短くなる

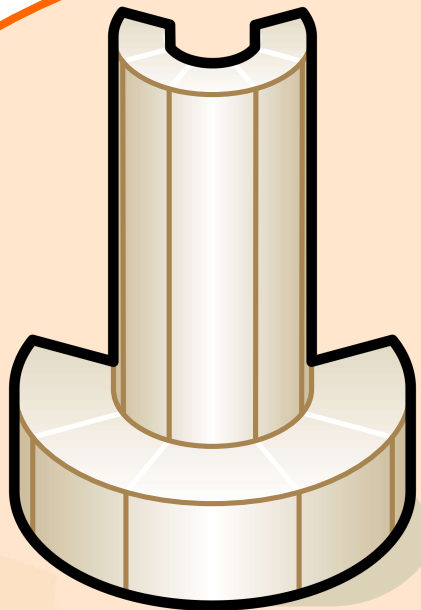




バッチ処理と対話処理

バッチ処理

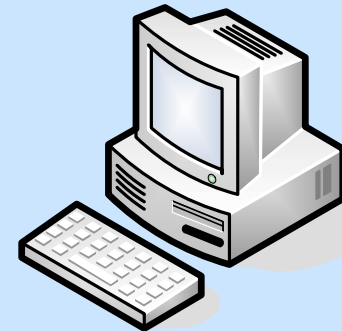
対話処理



スパコン



メインフレーム



パソコン

- ・ほとんどが対話処理
- ・レスポンスタイム最優先

■ ある決められた時間に必ず処理を終わらせる

- 例) 自動車の安全装置
- 「他のプロセスにリソースが使われていたので間に合いませんでした」
では済まない！
- クオントムの短いTSSでも間に合わない可能性



■ どうやったら実現できるか = 非常に複雑

- 同時実行プロセス数に制限
- 記憶領域の使用制限
- etc, etc...
- しかも処理速度を落としてはいけない



■ 複数のコンピュータで処理

- ネットワークの高速化により実現
- みんなでやれば速くできる!(ように思える)
 - 実際はそんなに単純ではない(あとで説明)

■ 例) 広い意味での分散処理

- Webページ閲覧
 - ページ送信側(サーバ)と受信側(クライアント)で協調処理(**クライアントサーバモデル**)
- ファイル共有
 - NFS (Network File System), SMB(Server Message Block)

■ 分散OS

- ネットワークでつながった複数のコンピュータ上で1つのOSを動かす
- 自動的に空いているCPUでプロセス実行
- 問題点：
 - 互換性がない(いままでのプログラムがそのまま動かない)
 - OS自体を分散させることによる性能低下
 - etc, etc...



■ 並列・分散コンピューティング

- OSではなくアプリケーションを分散

■ 並列コンピューティング

- 1つの計算機に複数のCPU
- 1つのCPUに複数のコア（演算ユニット）



■ クラスタコンピューティング

- 高速LANで複数の計算機をつないで処理



■ グリッドコンピューティング

- 広域インターネットにつながった計算機を多数使って処理





- 並列・分散コンピューティングの各形態の
利点・欠点は何か？
 - ・ どのような仕事に向いて、どのような仕事に向かないか
 - ・ 例) 人間の場合、二人以上でレポートを作成する
時はどうか？

- 並列コンピューティング
- クラスタコンピューティング
- グリッドコンピューティング

ポイントは

- ・ 距離
- ・ 人数
- ・ 連絡頻度



■ 並列コンピューティング

- 数値計算
- 力学シミュレーション

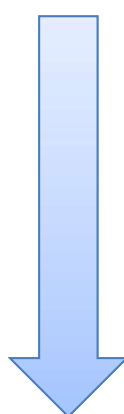
■ クラスタコンピューティング

- 画像処理
- 数値計算

■ グリッドコンピューティング

- SETI@home
 - 地球外電波信号の解析
- ゲノム解析

分散処理で知っておく値 by Jeff Dean



L1キャッシュ参照	0.5 ns
分岐予測ミス	5 ns
L2キャッシュ参照	7 ns
メモリ参照	100 ns
1KBをZIP圧縮	3,000 ns
1Gbpsで2KBを送る	20,000 ns
メモリから1MB連続で読む	250,000 ns
同一データセンター内のマシンと通信位置往復	500,000 ns
HDDシーク	10,000,000 ns
HDDから1MB読み出し	20,000,000 ns
カリフォルニアとオランダ間で通信1往復	150,000,000 ns

71429倍遅い！！

分散システム化すると基本的に遅くなる。その原因のほとんどはネットワーク遅延。
分散化して速くなるのはむしろ例外的と思った方がいい。

頼むから、安易な考えで分散ミドルウェアを使用することは止めて欲しい。



■ OSの目的

- 1. リソース抽象化によるアクセス容易性
- 2. 資源(リソース)管理による確認容易性
- 3. スケジューリングによる実行効率向上

■ 単位

- ジョブ
 - ユーザから見た仕事単位
- プロセス
 - リソースが配分される仕事単位



■ プログラムの処理形態

- バッチ処理
- 対話(インタラクティブ)処理
- ほかにも...
 - リアルタイム処理
 - 並列・分散処理

■ 効率の指標

- 実行を依頼してから結果が返るまでの時間
 - レスポンスタイム(対話処理)
 - ターンアラウンドタイム(バッチ処理)
- 一定時間に処理される仕事量:スループット