Lispのリアルタイムガーベジコレクション

リアルタイムシステム

2006年12月14日
木曜2限, 10:15-11:55,
工学部2号館2F, 222講義室
特任講師　岡田　慧

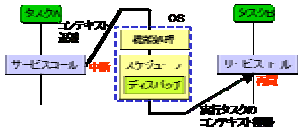
リアルタイムシステム講義予定

10/12	オリエンテーション, リアルタイム性とは	國吉
10/19	マルチタスク・マルチスレッド機構(1)	森
10/26	マルチタスク・マルチスレッド機構(2)	森
11/2	オペレーティングシステムの実行管理機構(1)	水内
11/9	オペレーティングシステムの実行管理機構(2)	水内
11/16	リアルタイムオペレーティングシステムの基礎	國吉
11/23	休日	
11/30	リアルタイムオペレーティングシステムの内部構造(1)	石綿 (産総研)
12/7	リアルタイムオペレーティングシステムの内部構造(2)	石綿 (産総研)
12/14	**Lispのリアルタイムガーベジコレクション**	**岡田**
12/21	RTミドルウェア　ーロボットソフトウェア標準ー	安藤 (産総研)
1/11	ヒューマノイドロボットのリアルタイム行動制御システム	加賀美 (産総研)
1/18	分散並列リアルタイムアーキテクチャ(1)	山崎 (慶応大)
1/25	分散並列リアルタイムアーキテクチャ(2)	山崎 (慶応大)

2

リアルタイムOS

- なんらかのイベントの発生に応答するまでの時間が許容時間内に収まること
- マルチタスクOS
  - 複数のタスクが並行に実行され処理されるOS
  - タスクスイッチ
    - タスクの実行切り替え
  - ディスパッチ
    - タスクを切り替え操作実行すること
  - プリエンプト
    - 実行中のタスクを中断すること
  - コンテキスト
    - タスクの実行再開に必要なデータ
- 実時間OS
  - タスク切り替え期間を短くしたもの



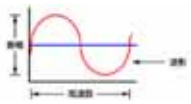
出典:http://www.assoc.tron.org/jpn/seminar/index.html

- 実時間OSが必要な例
  - 動画のリアルタイム・キャプチャー
  - 動画のフレーム信号を受け取ってから転送レート(30msec)以内に画像キャプチャ処理を終了すること.
  - CPUが他の処理(Webの画像ダウンロード)をしていて動画キャプチャ処理が待たされては困る.
  - 非リアルタイムOS(Windows, Linux)では動画のキャプチャー中は他の重い処理は出来ない

3

リアルタイム処理のサンプル

- 1音・着メロ
- 音の出し方
  - 周波数により音程が決まる
    - 440Hz = ラ音(A)
  - スピーカのON/OFFは 0x61 番地の　2bit め



```
while(1) {
    usleep(500);  /* この間隔を変えると音程が変化する */
    tmp = inb(0x61);
    if ( i++ & 1 ) { tmp = tmp | 0x02;
    }else{        tmp = tmp & ~0x02;}
    outb(tmp, 0x61);
  }
```

4

周期制御の実装法

- 実装1　usleep () を使う場合
  - usleepによりタスクの管理をカーネルに任せる
  - 非リアルタイムLinuxでのタスク管理の精度は10msec程度
- 実装2　getimeofday を使いbusywait する場合
  - while文の中で毎ループ時刻を取得する.
  - gettimeofday関数はusec単位の時刻の取得が可能
  - システムの負荷が高くなると音程がずれる
- 実装3　ARTLinuxを使う場合
  - ARTLinuxの実時間タスク用システムコールを使って周期を制御する
  - システムの負荷が高くとも音程はずれない

5

サンプルプログラム1の解説

- play_sound(int hz, int time) 関数
  - 周波数hzの音を, timeミリ秒出力する関数

```
int play_sound (int hz, int time) { // hz(Hz), time(msec)
  int tmp, i;

  for (i = 0; i < hz * 2 * time / 1000; ++i) {
    usleep(500*1000/hz);
    tmp = inb(0x61);
    tmp = (i & 1) ? tmp | 0x02 : tmp & ~0x02;
    outb(tmp, 0x61);
  }
  outb((inb(0x61) & ~0x02), 0x61);
}
```

6

サンプルプログラム2の解説

- usleep を gettimeofdayを使って置き換えた

```
int play_sound (int hz, int time) { // hz(Hz), time(msec)
  int tmp, i;
  long int d;
  struct timeval tv[2];
  struct timezone tz;

  for (i = 0; i < hz * 2 * time / 1000; ++i) {
    gettimeofday(&tv[0], &tz);
    do {
      gettimeofday(&tv[1], &tz);
      d = (tv[1].tv_sec - tv[0].tv_sec)*(1000000) +
        (tv[1].tv_usec - tv[0].tv_usec);
    } while (d < 500*1000/hz);
    tmp = inb(0x61);
    tmp = (i & 1) ? tmp | 0x02 : tmp & ~0x02;
    outb(tmp, 0x61);
  }
  outb((inb(0x61) & ~0x02), 0x61);
}
```


サンプルプログラム3の解説

- ART-Linuxを用いて実現

```
void play_sound (int hz, int time) { // hz(Hz), time(msec)
  int i, wait, ret, tmp;
  wait = 500*1000/hz;

  art_enter(ART_PRIO_MAX, ART_TASK_PERIODIC, wait);

  for (i = 0; i < hz * 2 * time / 1000; ++i) {
    art_wait();
    tmp = rtl_inb(0x61);
    tmp = (i & 1) ? tmp | 0x02 : tmp & ~0x02;
    rtl_outb(tmp, 0x61);
  }
  rtl_outb((inb(0x61) & ~0x02), 0x61);
}
```


Lispによるロボット開発環境

- インタプリタ言語
- ロボットのソフトウェアシステム
- ヒューマノイドの統合ソフトウェア環境

インタプリタ言語

- スクリプト言語／Lightweight Language (LL)／グルー言語
  - インタプリタ
    - ソースコードを逐次解釈しながら実行する⇔コンパイラ方式
  - 動的型付け言語
    - 型の限定を行わず実行時に合致するデータが渡されると期待して振舞う
  - 動的記憶管理
    - メモリの開放をプログラム処理系が自動的に行う
  - コードの作成，修正が容易なラピッドプロトタイプ(RP)言語
  - 「とにかく楽」
- 言語例
  - Perl(1987/Larry Wall), Python(1990/Guido van Rossum), Ruby(1995/まつもとゆきひろ)…
  - Lisp(1958/John McCarthy)
- ロボット用言語として最適
  - ロボットの反応を見ながら行動プログラムを記述できる
  - 外部ライブラリの導入とそれらの結びつけることが容易

実時間EusLisp

- 実時間スレッド
  - 周期実行可能なスレッド
  - 実時間OSが提供する実時間スレッド機能を利用
- 実時間GC
  - GCによる停止時間の短縮

EusLispでの実時間スレッドの実現

- 実時間OSが提供する実時間スレッド機能を利用する
- ART-Linux
  - art_enter(art_prio_t prio, art_flags_t flags, int usec)
    - 非実時間プロセスがart_enterシステムコールを呼び出すと実時間タスクに変換される
  - art_wait(void)
    - 実時間タスクがart_waitを呼び出すと次の周期実行開始時刻まで待機する
  - art_exit(void)
    - 実時間タスクがart_exitを呼び出すと，非実時間プロセスに変換される
- EusLisp他言語インターフェース
  - defforeign
    - EusLispから呼び出すためのC関数に対するエントリーを作る

- **実時間スレッド制御関数の定義**

```
(setq a (load-foreign "art-eus.so"))
(defforeign art-enter a "art_enter" () :integer)
(defforeign art-wait a "art_wait" () :integer)
(defforeign art-exit a "art_exit" () :integer)
```

- EusLisp**の実時間スレッドプログラム**

```
(print "play sounds with art functions")
(art-enter *art-prio-max* *art-task-periodic* (/ 500000 hz))
(dotimes (i (* hz 5))
  (art-wait)
  (sound-play))
(art-exit)
(exit)
```


ロボットのソフトウェアシステム

- **動作計画プランナ　1sec～**
- **画像処理･パターン処理　15～1000msec**
- **運動学･動力学計算 1～100msec**
- **力覚, 触覚, サーボ 1～10msec**
  - **センサ･サーボ処理は実時間（リアルタイム）性が要求される**
  - **制御用マイコンや実時間OSを用いたリアルタイム処理**
  - **上位のソフトウェアをLispにより記述してきている**

熟考系　反射系　記号処理　幾何計算　実時間処理

- **これまでは歩行や把持といった要素動作の研究**
- **環境･状況に応じて要素動作を選択･生成する行動研究へ**
  - **実世界知識の記述･行動計画プランナ等の上位層がターゲットになる**
- **大規模かつ複雑なソフトウェアの構造化･体系化**
- **→生産性の高い**Lisp**言語を中核にしたシステムの重要性**

ロボット研究用Lisp処理系EusLisp

- **インタプリタ言語**
- **オブジェクト指向**CommonLisp
- **三次元幾何モデラ**
- **ウィンドウグラフィックス**
- **マルチスレッド機能**
- **多言語インターフェース**

- 1986: **松井俊浩**
  - **産業技術総合研究所．当時は通産省電子技術総合研究所．**
  - **ロボットの研究ではセンサ処理，環境認識，動作作業計画がテーマ．そこではロボットと外界の三次元幾何モデルが重要になるため，記号処理システムから簡単に使えて拡張性の高いソリッドモデラが必要**
- http://www.dh.aist.go.jp/~t.matsui/eus/euslisp.html

- 1996:**マルチメディアデスプレイのシステム記述言語として開発** mc68020
- 1987:Sun**ワークステーションへの移植**
- 1988:**情報処理学会全国大会･記号処理研究会発表**
- 1994:Solaris**対応，マルチスレッド化，**xwindow toolkit
- 1995:**情報処理学会論文誌（マルチスレッド**Lisp）
- 1996:**慶応大学中西研** Realtime GC**移植**
- 1999:**東京大学金広** Windows, Cygwin**への移植**
- 2003:**京都大学湯浅研** Realtime GC**開発**

情報システム工学研究室（JSK）では･･･

- 1988:**ロボット用モデラー，モデルに基づく視覚処理で卒業･修士論文**
- 1990:**トラッキングビジョン･線分検出ハードウェアと接続**
- 1993:**リモートブレインライブラリ**
- 1993-1998:Sun**とトランスピュータビジョンでロボットシステムの記述言語**
- 2002:**モーションキャプチャデータ（**Bvh**）アニメーション**
- 2003:**動作生成プランナ．ソフトウェア視覚処理，音声認識ライブラリ**

EusLisp機能紹介：オブジェクト指向

- defclass *classname &key :super object :slots (var*)*
- defmethod *classname {(selector lambda-list . body)}**
- **オブジェクトの構造と動作は**defclass**マクロや**defmethod**特殊式により定義されている．**
- send *object selsector {arg}**
- **オブジェクトに**selector**と**arg**で構成されるメッセージを送信する**

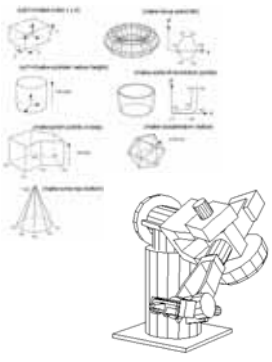
```
(defclass a
  :super object
  :slots (data))
(defmethod a
  (:init (args)
         (setq data args) self)
  (:data () data))
(setq daa (instance a :init 10))
(send daa :data)

(defclass b
  :super a
  :slots ())
(defmethod b
  (:init (args)
         (send-super :init args) self))
(setq daa (instance b :init 10))
(send daa :data)

(send dbb :methods)
(class-hierarchy a)
```


EusLisp機能紹介：幾何モデラ

- **座標系**(coordinates)
- :locate points
- :rotate theta axis
- **基本**body**の作成関数**
- (make-cube x y z)
- (make-prism points sweep)
- (make-cylinder radius height)
- (make-cone top bottom)
- (make-solid-of-revolution points)
- (make-torus points)
- (make-isosahedron radius)
- **基本**body**の合成関数**
- (body+ body1 body2)
- (body- body1 body2)



EusLisp機能紹介：ウィンドウ，スレッド

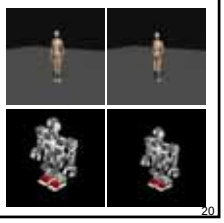
- ウインドウプログラミング機能
  - XToolKitは，ボタン，プルダウンメニュ，テキストwindowなどのGUI要素を使用してGUIを作成するのを容易にするための高レベルXwindowインターフェース
- マルチスレッド機能
  - (sys:make-thread num)
    - スレッドを生成しシステムのスレッドプールにおく．
  - (sys:free-threads)
    - スレッドプール内の空いたスレッドのリストを返す．
  - (sys:thread func &rest args)
    - スレッドプールから空いたスレッドを取り出し(func . args) の評価のために割り当てる．
  - (sys:thread-no-wait func &rest args)
    - スレッドはwait-threadされることなしに評価が終了したときにスレッドプールに戻される．
  - (sys:wait-thread thread)
    - sys:threadにより与えられたfuncallの評価が終了するのを待ち，その結果を受け取る．

```
$ (sys:make-thread 4)
$ (defun print-i (m)
  (unix:sleep 1)
  (dotimes (i m)
    (dotimes (j i) (format t " "))
    (format t "~A~%" i)
    (unix:usleep 100000)
  ))
$ (sys::thread-no-wait #'print-i 30)
$ (sys::thread-no-wait #'print-i 30)
0
 1
0
  2
 1
   3
```

19

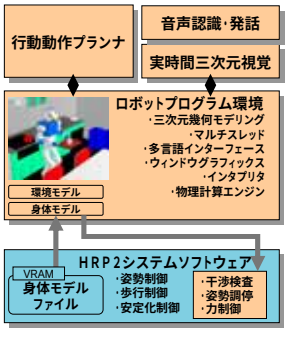
拡張版EusLisp (jskrbeusgl)

- 情報システム工学研究室(JSK)拡張版EusLisp
- Euslib
  - jsk
    - pickview.l ビューワー
    - galib.l GAライブラリ
    - neurolib.l ニューラルネットライブラリ
    - kalmanlib.l カルマンフィルタ
    - searchlib.l 探索ライブラリ(from AIMA)
    - viavoicelib.l 音声認識インターフェース
  - rbrain
    - basicmodel.l ロボットモデルクラス
    - robots.l ロボット定義
    - benet.l ロボット動作記述言語
    - statenet.l ロボット状態動作記述言語
    - eusbvh.l モーションキャプチャーデータ

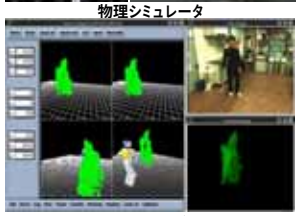


20

ヒューマノイドの統合ソフトウェア環境



物理シミュレータ



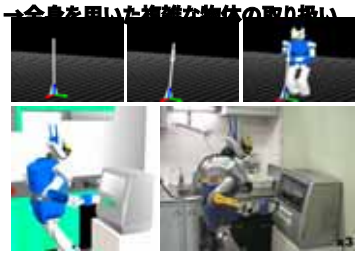
視覚処理と環境モデルの結合

21

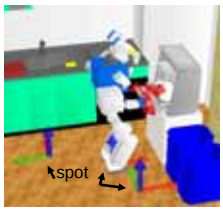
道具利用動作のための環境モデリング

- 環境知識
  - spot (移動位置) 情報
  - handle (把持点) 情報
  - attention (操作点) 情報

→全身を用いた複雑な物体の取り扱い

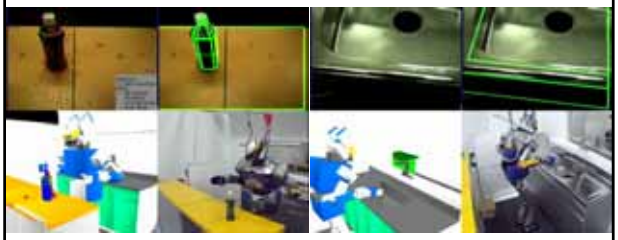


環境知識を用いた全身動作の生成

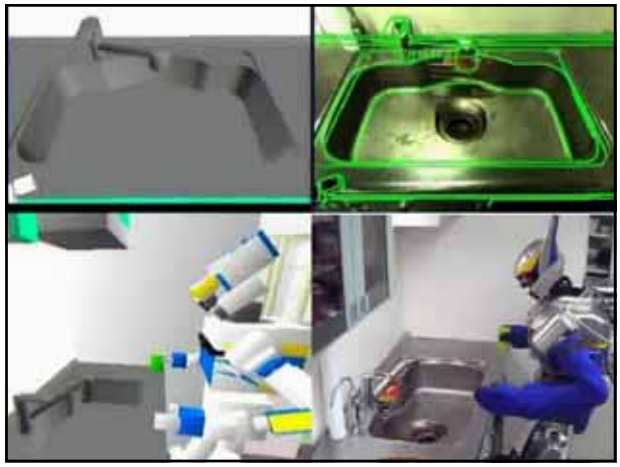


22

環境モデルを用いた視覚認識



23



ガベージコレクション

- ガベージコレクション（GC）とは
- GCの基本アルゴリズム
  - 参照カウント
  - 複写
  - マーク＆スイープ

ガベージコレクション (GC)

- 動的記憶管理
  - プログラム実行時の記憶領域の確保
  - Cf. 大域変数（コンパイル時割り当て），局所変数（スタックへの割り当て）
  - C: malloc
  - Pascal, C++, Java, Ruby, Python: new
  - Lisp: cons
- 使用されなくなった記憶領域の開放
  - ✕ プログラマの責任で開放(free, delete)
  - ○ システム（言語の実行系）が自動的に開放→**ガベージコレクション**
- メリット
  - 正確・信頼性の高いプログラム
    - 浮いたポインタをなくす．メモリーリークをなくす
  - プログラムの簡潔なインターフェースとデザイン，容易な開発
  - セキュリティ対策

26

サンプルプログラム（GC無し言語）

- スレッド1：メモリを確保する
- スレッド2：ソート（バブルソート）

```
bool do_delete = false;

class Mutator : public Thread {
  bool loop;
public:
  Mutator(){}
  void Setup (){loop = true;}
  void Execute(){
    while (loop) {
      vector<vector <int> *> *v = new vector<vector <int> *>();
      // mutator
      for (int k = 0; k < 100; k++)
        v->push_back(new vector<int>(100000));
      //                          メモリの確保処理
      if ( do_delete ) {
        for (vector<vector <int> *>::iterator i = v->begin(); i != v->end(); i++)
          delete (*i);
        delete v;
                                  メモリの開放処理
      }
    }
  }
}

int main(int argc, char *argv[]){
  Sort *sort = new Sort();
  Mutator *mutator = new Mutator();
  sort->Start();
  mutator->Start();
  sort->Join();
}
```

開放を行わないと大幅な性能低下を招く

```
class Sort : public Thread {
  void sort(vector<int> *a) {
    for (int i = a->size()-1; --i>=0; ) {
      bool swapped = false;
      for (int j = 0; j<i; j++) {
        if ((*a)[j] > (*a)[j+1]) {
          swap((*a)[j], (*a)[j+1]);
          swapped = true;
        }
      }
      if (!swapped) return;
    }
  }

public:
  vector<int> arr;
  void Setup (){
    arr.resize(20000);
  }
  void Execute(){
    scramble(&arr);
    sort(&arr);
  }
};
```

27

サンプルプログラム（GC有り言語）

- スレッド1：メモリを確保する
- スレッド2：ソート（バブルソート）
  - プログラマはメモリの開放処理の必要は無い

```
class Mutator extends Thread {
  boolean loop = true;

  public void run() {
    Vector v = new Vector();
    while ( loop ) {
      // mutator
      for (int k = 0; k < 100; k++) {
        Vector e = new Vector();
        for (int l = 0; l < 1000; l++){
          e.addElement(new Integer(l));
        }
        v.addElement(e);
      }
      //                    メモリの確保処理
      v = null;
                          メモリに別の値を代入
    }
  }
```

```
class Sort extends Thread {
 static void sort(int a[]) throws Exception {
      for (int i = a.length; --i>=0; ) {
        boolean swapped = false;
        for (int j = 0; j<i; j++) {
          if (a[j] > a[j+1]) {
            swap (a[j], a[j+1]);
            swapped = true;
          }
        }
        if (!swapped) return;
      }
  }

  public void run(){
    scramble();
    sort(arr);
  }
}

class GCTest {
  public static void main(String[] args) {
    Sort sort = new Sort();
    Mutator mutator = new Mutator();
    sort.start();
    mutator.start();
    sort.join();
  }
}
```

28

プログラミング言語の発展とGC

- 黎明期
  - 1957 FORTRAN（IBM/ジョン・バッカス）
  - 1960 COBOL
  - 1960 LISP（MIT/ジョン・マッカーシ）
- 構造化言語とオブジェクト指向
  - 1964 BASIC（ダートマス大学）
  - 1969 Pascal（チューリッヒ工科大学/ニクラス・ビィルト）構造化言語
  - 1970前半 Smalltalk（XEROX）オブジェクト指向
  - 1972 C（AT&T/デニス・リッチー）
  - 1983 C++
- インタプリタ言語，ネットワーク，GUIのためのオブジェクト指向
  - 1987 Perl（ラリー・ウォール）
  - 1990 Python（グイド・バンロッサム）
  - 1995 Ruby（まつもろゆきひろ）
  - 1995 Delphi（ボーランド）
  - 1995 Java（Sun）
  - 2002 C#（マイクロソフト）

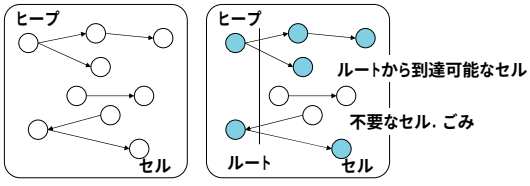
Heavyweight Language (HL)
大規模ソフトウェアの開発保守
↓
Lightweight Language (LL)
安全性，効率性
人間が書きやすい言語

・インタプリタ言語
ユーザがメモリ管理しなくてよい
↓
GCを持った言語

29

GCを理解するための用語

- ヒープ：記憶領域
- セル（オブジェクト）：記憶領域の単位，他のセルへのポインタやデータ
- ルート：ポインタ集合．大域変数，スタック．ルートから到達可能なセルが「生存」している，アクセス可能．到達不可能なセルは「ごみ」．
- GC：ヒープ中のセルを生存セルとごみに区別し，再利用のために束ねる



30

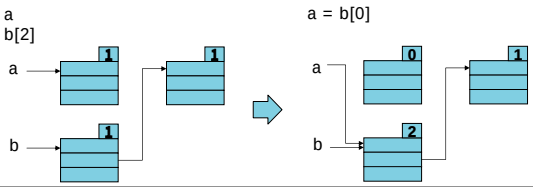
GCの基本アルゴリズム

- 参照カウント (Python)
- 複写
  - 世代別 (Java)
- マーク＆スイープ (Ruby)

31

参照カウント (1960 Collins)

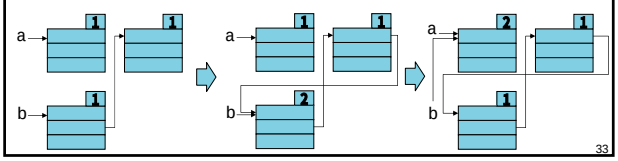
- 参照カウンタ
  - 「すべてのセルに，そのセルがいくつのポインタによって指されているか」を示す値（参照カウンタ）をもたせる
  - ポインタの付け替えが発生した段階で，
    - 新しいポインタの参照カウンタを増やす
    - 古いポインタの参照カウンタを減らす
  - 値が0になった段階でごみなったことがわかる



32

- 長所
  - ごみになったかどうかがそのセルの情報だけで判断可能
  - ごみになったセルは直ちに回収可能
- 短所
  - 環状構造を回収できない
  - セルへの参照の増減のたびにカウンタの更新が必要でオーバヘッド有り

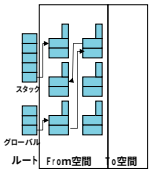
環状構造の例



33

複写 (1969 Cheney)

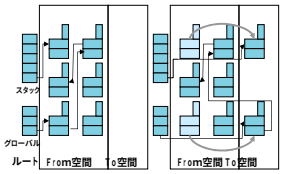
- ヒープをFrom空間とTo空間に分割する
- 生存するセルだけをTo空間にコピーする
  - ルートからポインタの参照関係をたどる
- コピーされたFrom空間のセルには，その番地にTo空間への引越し先ポインタを持つ
  - 複写しようとするセルに引越し先ポインタがあるばあいは，複写せずに，その複写先ポインタを，セルの新しい番地としてコピーする．



34

複写 (1969 Cheney)

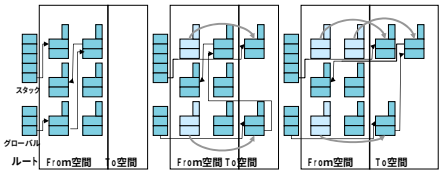
- ヒープをFrom空間とTo空間に分割する
- 生存するセルだけをTo空間にコピーする
  - ルートからポインタの参照関係をたどる
- コピーされたFrom空間のセルには，その番地にTo空間への引越し先ポインタを持つ
  - 複写しようとするセルに引越し先ポインタがあるばあいは，複写せずに，その複写先ポインタを，セルの新しい番地としてコピーする．



35

複写 (1969 Cheney)

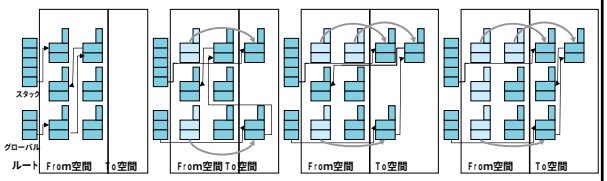
- ヒープをFrom空間とTo空間に分割する
- 生存するセルだけをTo空間にコピーする
  - ルートからポインタの参照関係をたどる
- コピーされたFrom空間のセルには，その番地にTo空間への引越し先ポインタを持つ
  - 複写しようとするセルに引越し先ポインタがあるばあいは，複写せずに，その複写先ポインタを，セルの新しい番地としてコピーする．



36

複写 (1969 Cheney)

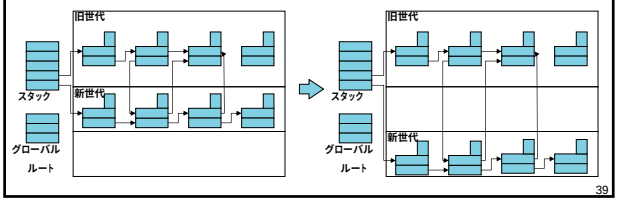
- ヒープをFrom空間とTo空間に分割する
- 生存するセルだけをTo空間にコピーする
  - ルートからポインタの参照関係をたどる
- コピーされたFrom空間のセルには，その番地にTo空間への引越し先ポインタを持つ
  - 複写しようとするセルに引越し先ポインタがあるばあいは，複写せずに，その複写先ポインタを，セルの新しい番地としてコピーする．



- 長所
  - 環状構造を回収できる
  - メモリ回収と同時にコンパクションが出来る
  - メモリの配置を効率的に出来る (関係あるセルが近い領域にある)
  - 計算時間は生存しているセルに比例する
- 短所
  - セルの領域が倍以上必要になる，位置が変わる
  - 大量の生存セルが存在する場合は複写のオーバヘッドが大きい

世代別ＧＣ (1983 Lieberman and Hewitt)

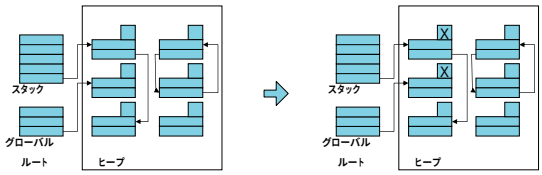
- 一般にセルは以下の傾向がある
  1. 年齢の若いセルほどごみになりやすい
  2. ポインタの方向は若いセルから古いセルへ向かうのが大多数である
- 若いセルを一領域に集め，その領域だけ頻繁にＧＣを行う
- 古いセルから新しいセルへのポインタの更新の必要有り
- ＧＣを一定回数潜り抜けた若いセルは古いセルの仲間となる



マーク＆スイープ (1970 McCarthy)

- マーク処理:
  - ルートから出発してポインタの参照関係をたどりながらすべての生存セルに印をつける
- スイープ処理:
  - ヒープ全体を操作して印のないセル (ごみ) を回収する

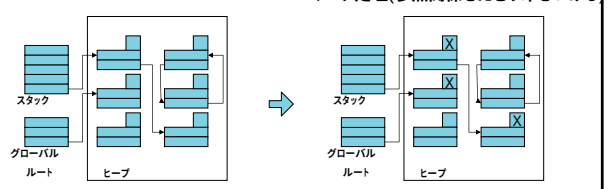
マーク処理(ルートに印をつける)



マーク＆スイープ

- マーク処理:
  - ルートから出発してポインタの参照関係をたどりながらすべての生存セルに印をつける
- スイープ処理:
  - ヒープ全体を操作して印のないセル (ごみ) を回収する

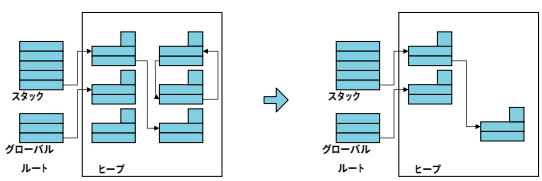
マーク処理(参照関係をたどり印をつける)



マーク＆スイープ

- マーク処理:
  - ルートから出発してポインタの参照関係をたどりながらすべての生存セルに印をつける
- スイープ処理:
  - ヒープ全体を操作して印のないセル (ごみ) を回収する

スイープ処理



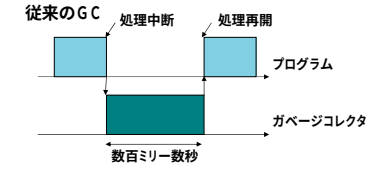
- 長所
  - 環状構造を回収できる
  - オーバヘッドが少ない．データー操作時のオーバヘッドは無い
  - ＧＣの実装以外の場所でＧＣのことを考えなくていい
- 短所
  - ごみあつめの処理時間はヒープのサイズに比例する
  - フラグメンテーションがおきる
  - ＧＣの負荷が一点に集中する

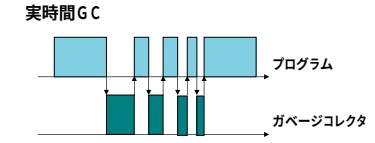
リアルタイムガベージコレクション

- リアルタイムＧＣ
  - インクリメンタルＧＣ
  - On-the-fly GC
  - スナップショット GC
- EusLispにおけるリアルタイムGC
  - リターンバリア
- ヒューマノイドによるリアルタイムGCの性能評価

実時間ＧＣ

- インクリメンタルＧＣ
- On-the-fly GC
- スナップショットＧＣ



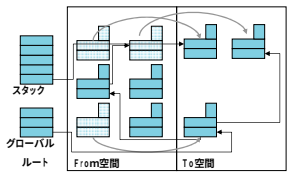


GCの動作時間を状況に応じて自動的に変更して実行

インクリメンタルＧＣ (1978 Baker)

- コピーＧＣを基本 (Bakerのアルゴリズム)
- 一回のＧＣ処理を，細かく分けてプログラムを実行する間に少しづつ実行する
- 一括ＧＣと異なる点
  - 新しくセルを割り当てるときに，コピー先の領域に割り当てる
    - 新しく割り当てられたセルが，すぐにごみになっても現在のＧＣでは回収されてない
- From空間のセルをプログラムが参照する場合・・・リードバリア処理

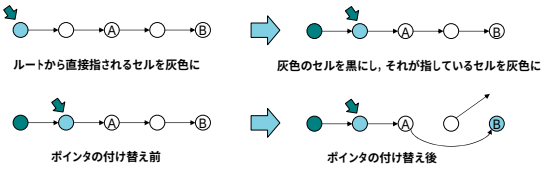
- リードバリア
  (To空間に複写されたセルはその古い番地に複写先のポインタを持つ)
  - そこにFrom空間へのポインタがあれば，To空間へのポインタに置き換え
  - そのポインタが指すセルがTo空間への引越し先ポインタを持っていれば，その引越し先ポインタで置き換え
  - そうでなければ，そのセルをTo空間へ複写し，引越し先ポインタを古いセル内に残して，元のポインタのかわりに，その引越し先ポインタを使用
- データを取り出すたびにオーバヘッドを生じる



On-the-fly GC (1978 Dijkstra)

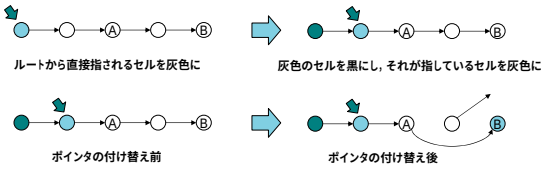
- マーク＆スイープ方式を基本
- プログラムを実行するスレッド(mutator)とＧＣ専用のスレッド(collector)の平行実行
- mutator によるポインタの付け替えが問題
  - Collector がセルAにマークをつけ，その先の処理を行っている
  - Mutatorによるポインタの付け替え．Ａからまだマークのついていないへ
  - Ｂは生存しているセルＡから指されているので生存しているが，マークがつかず，ごみとして回収されてしまう

- Dijkstraにより考案された方法（セルを，黒，灰色，白に分ける）
  - 最初，すべてのセルは白
  - Collectorはまず，ルートから直接指されるセルを灰色にする
  - 次に，灰色のセルを探して，そのセルを黒にし，それが指しているセルが白なら灰色にする
  - Mutatorはあるセルにポインタを格納する際に，そのポインタが指すセルが白なら灰色にする．
  - 灰色のセルがなくなった時点でマーク処理が終了，白いセルを回収する



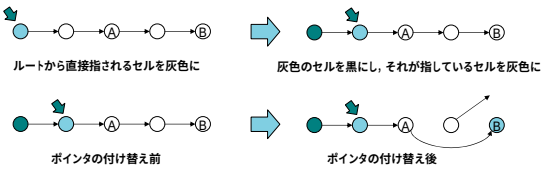
49

- セルを灰色にする操作
  - セルをＧＣ用のスタックにプッシュする
- セルを黒くする操作
  - マーク付け
- 白いセル：
  - マークがなくて，かつＧＣ用スタックから直接指されていないセル．ごみ．



50

- 黒いセルが白いセルを直接指すことはない．間接的に指す場合は必ず途中に灰色のセルを介する
- マーク処理終了時点で白いセルは全てごみだが，黒いセルは全て生存しているとは限らない．
- いったん黒くなってから不要になったセルは黒いまま．これらは次のＧＣで回収される



51

- 短所
  - セルの内容を変更する際，新しく格納されたポインタが指しているセルを灰色にするオーバーヘッドがある
    - 一般的なプログラムの実行中に使われる他の操作（たとえばリード）と比べると使用頻度が小さく，影響は少ない
  - ルートへの代入処理に対してもオーバヘッドが生じる
    - 実際のプログラムではレジスタやスタック（これらはルートになる）への操作が多く，このオーバーヘッドが無視できない

52

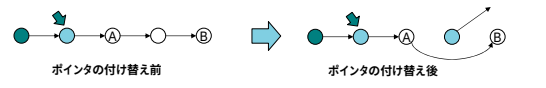
スナップショットＧＣ (1988 湯淺)

- ＧＣ開始時に使用中のセルはそのＧＣ中には回収しない
- ＧＣ中に新しく割り当てられたセルはそのＧＣに中に回収しない
  - ＧＣ開始時にヒープの写真（スナップショット）をとり，そのとおりにセルを回収する
- マーク＆スイープ方式が基本
- On-the-fly GC
  - セルの内容が変更されるとき，新しく格納されるポインタが白を指していれば，そのセルを灰色にする（ＧＣ用スタックにプッシュする）
  - ＧＣ中に割り振られたセルは灰色にする
- スナップショットＧＣ
  - セルの内容が変更されるとき，**いままで格納されていたポインタ**が白を指していれば，そのセルを灰色にする（ＧＣ用スタックにプッシュする）
  - ＧＣ中に割り振られたセルは黒にする
  - マーク中にルートポインタに変更があっても再スキャンの必要が無い

53

スナップショットＧＣ (1988 湯淺)

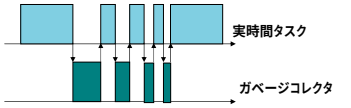
- ＧＣ開始時に使用中のセルはそのＧＣ中には回収しない
- ＧＣ中に新しく割り当てられたセルはそのＧＣに中に回収しない
  - ＧＣ開始時にヒープの写真（スナップショット）をとり，そのとおりにセルを回収する
- スナップショットＧＣ
  - セルの内容が変更されるとき，**いままで格納されていたポインタ**が白を指していれば，そのセルを灰色にする（ＧＣ用スタックにプッシュする）
  - ＧＣ中に割り振られたセルは黒にする



54

EusLispにおける実時間ＧＣ

- ストップＧＣ（マーク＆スイープ法）
  - ルートから到達可能なオブジェクトに印（マーク）をつけ，印の付かなかったオブジェクトを廃棄（スイープ）する
- Euslispにおける実時間ＧＣへの取り組み
  - １９９６　慶応大学中西研（RWC所属）　ＧＣ移植の試み
  - ２００３　京都大学湯浅研　実時間ＧＣ開発．ソフトウェア科学会発表
    スナップショットＧＣ＋リターンバリア
  - ２００４　ソフトウェア科学会:花井，湯浅，岡田，稲葉
    「ロボット行動ソフトウェア環境に適した実時間ごみ集め」



実時間タスクをしていないときにＧＣしたい
ＧＣは実時間タスクが実行可能になったら速やかにＣＰＵを明け渡す

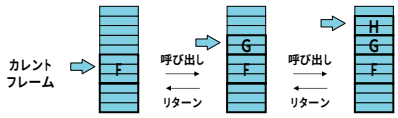
- 停止時間の短縮
  - 実時間スレッドが実行可能になったときにGCがプリエンプトできない状態であると実時間スレッドの再開を遅らせるしかない
  - 非プリエンプトなGC処理をなくす or なくせない部分は短く分割してEusスレッドが途中に割り込めるようにする
  - スナップショット　＋　リターンバリア方式を採用
    - 数100μsec程度の小さな停止時間を実現するのに適している
    - スナップショット
      - ライトバッファを用いて，ダーティな（書き換えられた）オブジェクトをマークする
    - リターンバリア
      - スタックを関数のフレーム単位に分割してルート挿入を行う

リターンバリア（湯淺2004）

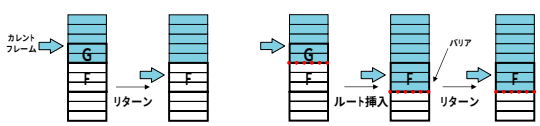
- ルート挿入
  - ルート（特にスタック）は頻繁に書き換わる
  - スナップショットＧＣではマーク操作を行う前にルートから直接されている全てセルに対して印をつける（スナップショットを取る・ルート挿入）
  - ルート挿入による停止時間は大きさに比例し，上限を見積もれない
- ルート挿入のインクリメンタル処理
  - セルAは印がつけられているが，セルBは挿入されていないルートから指されているため印がついてない．
  - このときポインタが書き換えられると，セルBはスタックから挿入済みのルートからしか指されていないので印がつかずゴミとされる

ポインタ書き換え前　　ポインタ書き換え後

- スタックをフレーム単位に分割してルート挿入を行う
  - プログラムは関数の呼び出しが入れ子になって実行される
  - 関数が呼び出されるとその関数に必要なデータ領域（引数，返り値，局所変数，リターンアドレス：関数フレーム）がスタックに積まれ，関数からリターンするとそのフレームは破棄される
  - 関数の実行中は，その関数フレーム内（カレントフレーム）にしか読み書きを行わない
  - 関数の実行が終了するとフレームはスタックからポップされる



- リターンバリア
  - 関数からのリターンが生じた際にスタック内のマークされた部分とそうで無い部分の境界（バリア）に注目し，カレントフレームがバリアを超えることを防ぐ
  - カレントフレームがバリアを越えそうになった場合，プログラムの実行を中断しルート挿入を進める



GC処理の制御

- 実時間ＧＣの実現には
  - 非プリエンプティブなＧＣ処理を短くする
  - 適度にヒープに空きがある状態でＧＣを開始し，プログラムのメモリ要求にこたえられない事態に陥る前に回収を行わなければならない（飢餓状態の回避）
- プログラムのメモリ要求に組み合せインクリメンタルに行う方法
  - ＧＣの処理量を調整しやすく飢餓状態の回避が容易
  - バースト的なメモリ要求があった場合ＧＣの負荷が大きくなりすぎる
- ＧＣ処理を専用スレッドを用いて行う方法
  - 飢餓状態を回避できるかがスケジューラに依存する

⇩

- ＧＣスレッドとインクリメンタル処理の併用

インクリメンタル処理の開始条件

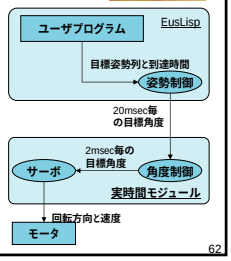
- CPUの空き時間にGC処理を進めるため，GCスレッドは低優先度
- GCの進み具合に応じてプログラムによるインクリメンタルGC処理を課す
- 飢餓状態に陥らないためのGCの開始条件
  - K:単位メモリ割当量に対するインクリメンタルに行うGC処理量[/byte]
  - W(t): 時刻tにおける残りのGC処理量
  - F(t): 時刻tにおける空きメモリ量
- F(t)>W(t)/K
- GCの評価
  - ルートスキャン処理における停止時間
  - 複数スレッドで別々にFFT処理を実行
  - ヒープの空きが少なくなったスレッドがGC開始要求を出してから実時間スレッドが処理を再開する時間

スレッド数	平均	最大	GC回数
1	31.0	32.0	40
2	40.2	78.0	40
3	47.0	97.0	43

usec

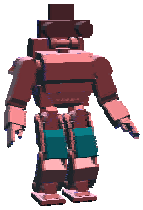
実時間GCの性能評価用ソフトウェア構造

- ヒューマノイドロボットHOAP-2（富士通オートメーション）
- 身長48cm，体重6kg
- 25自由度
  - 脚：6自由度x2，腕：4自由度x2，手：1自由度x2
  - 腰：1自由度x1，首：2自由度x1
- 体内のUSBネットワークで各モータを接続
- サーボループ
  - 2[msec]周期の実時間モジュール
  - カーネル空間内のメモリ空間の情報を読んでモータへの目標角度を送る
- 角度制御ループ
  - 2[msec]周期の実時間モジュール
  - ユーザ空間のEusLispから送られてきた目標角度を2msec毎に補間しサーボループへ渡す
- 姿勢制御ループ
  - 20[msec]周期のスレッド
  - 入力された目標姿勢角度と到達時間情報から20msec毎の目標角度を補間計算し実時間モジュールへ送る



実時間GCの性能評価

- 全身を用いた屈伸運動
- 20msec周期の全身の姿勢を計算
- 目標姿勢角度と到達時間を姿勢制御スレッドに渡す．



```
(setq av *initial-pose-vector*)
(setq m 200)
(dotimes (i (* m 3))
  (setq c2 (/ (- (cos (/ (* i pi) m)) 1.0) -2.0))
  (send *ns* :angle-vector
        (v+ av
            (float-vector 0.0 0.0 (* c2 -40) (* c2 80) (* c2 -40) 0.0
                          (* c2 -80) 0.0 0.0 0.0
                          0.0 0.0 (* c2 -40) (* c2 80) (* c2 -40) 0.0
                          (* c2 -80) 0 0.0 0.0
                          0.0 0 0)) 20))
```


実験風景

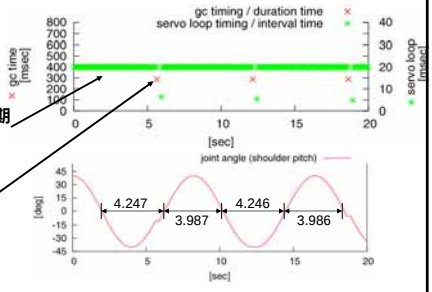
- 上段
  - ストップGCを用いた場合
- 下段
  - 実時間GCを用いた場合
- モータが出す音に集中してください

実時間GCの性能評価（ストップGC）

- 一回のGCで平均 287.3[msec]の停止時間
- その間ロボットの姿勢角度は更新されず，動きが止まる

姿勢制御ループの周期
（＊で示している）

GCによる停止時間
（x印で示している）

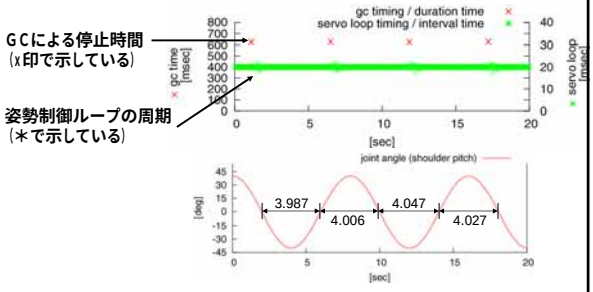


実時間GCの性能評価（実時間GC）

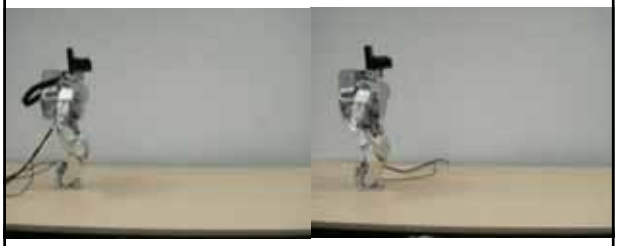
- 一回のGCの開始から終了で平均623.6[msec]の停止時間
- ロボットの動作は滑らか

GCによる停止時間
（x印で示している）

姿勢制御ループの周期
（＊で示している）



歩行動作における実時間GCの評価



ストップGCを用いた場合　　実時間GCを用いた場合

まとめ

- リアルタイムシステム
- Lispによるロボット開発環境
  - インタラクティブ言語
  - ロボットのソフトウェアシステム
  - ヒューマノイドの統合ソフトウェア環境
- ガベージコレクション
  - ガベージコレクション(GC)とは
  - GCの基本アルゴリズム
    - 参照カウント
    - 複写
    - マーク&スイープ
- リアルタイムガベージコレクション
  - リアルタイムGCアルゴリズム
    - インクリメンタルGC
    - On-the-fly GC
    - スナップショット GC
  - EusLispにおけるリアルタイムGC
    - リターンバリア
  - ヒューマノイドによるリアルタイムGCの性能評価
- http://www.jsk.t.u-tokyo.ac.jp/~k-okada/realtime2006/