2012年度
計算機システム演習　第6回
2012.06.01

福田　圭祐

本日の内容 (Outline)

1. **動的メモリ割り当て**: ヒープ領域にデータを確保
   - 動的配列： 動的に配列の領域を確保
   - c.f.) 前回は$spを用いてスタック領域上にデータを格納

サービス	番号 ($v0)	引数	返り値	意味
print_int	1	$a0（整数）		整数値を表示
print_string	4	$a0（文字列のアドレス）		文字列を表示
read_int	5		$v0（整数）	整数値を読込む
read_string	8	$a0（バッファ） $a1（長さ）		文字列を読込む
sbrk	9	$a0（メモリサイズ）	$v0（アドレス）	メモリを割り当て
exit	10			プログラム終了

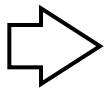
2. **例外処理**
   - トラップ例外

動的配列

- sbrk システムコールで実行時(動的)に確保される配列
  - ⇔ 静的配列：データセグメントで実行前から(静的に)サイズが定義された配列
  - sbrk ($v0=9)：メモリ領域割り当てを要求
  - 引数（$a0）：確保するメモリ量（Byte）
  - 返り値（$v0）：確保されたメモリの先頭アドレス

```
int* b = (int *)malloc(4*4)
```

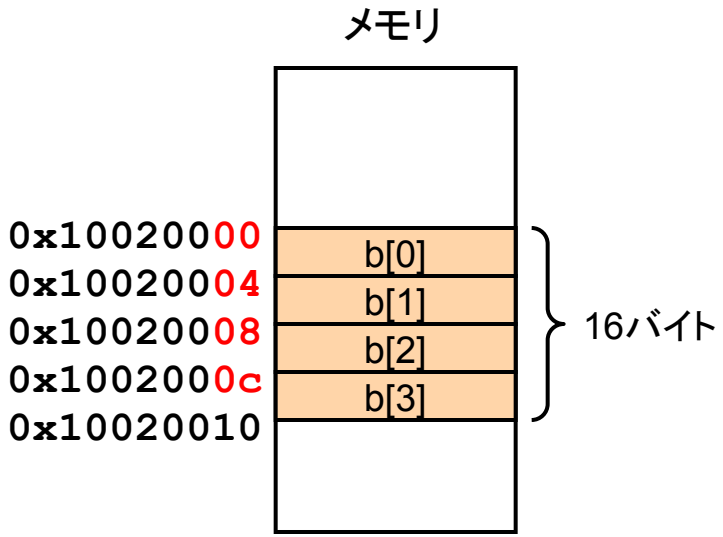
int[4]の割り当て



```
li $v0, 9
li $a0, 16                    # 4*4byte
syscall                # sbrk
move $t0, $v0
```

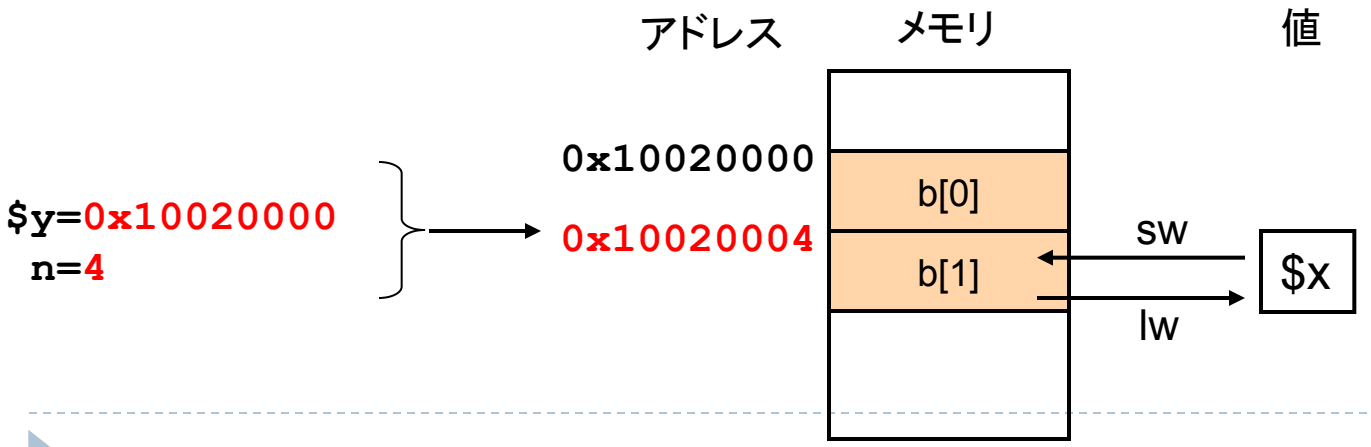
動的配列のアドレス

- sbrk した結果…
  - `$v0 = 0x10020000`
- bの先頭アドレス：**0x10020000**
  - b[0] のアドレス
    0x10020000
  - b[1] のアドレス
    0x10020004
  - ：



動的配列の操作

- メモリアクセスは`sw`, `lw`
  - データセグメント、スタック領域（スタックセグメント）、ヒープ領域
- メモリアクセス命令
  - `sw $x, n($y)`
    - $x の値をメモリのアドレス n + $y に代入
  - `lw $x, n($y)`
    - メモリのアドレス n + $y にある値を $x に代入



動的配列の操作の例

- **b[1]**
  - 先頭のアドレス：$t0 = b
  - インデクス：I

```
# 4+b
lw $v0, 4($t0)
```

- **b[i]**
  - 先頭のアドレス：$t0 = b
  - インデックス：$a0 = i

```
add $t1, $a0, $a0
add $t1, $t1, $t1
add $t2, $t0, $t1
lw $v0, 0($t2)
```

$t1 = 4*$a0

$t2 = $t0 + 4*$a0

インデクス*i* にアクセスする場合、
*4*倍*(i*4)*すればよい

動的配列 (サンプル)

- データ数を入力させ、
  そのサイズnの配列bを
  b[i]=iと初期化するプログラム

m8.s

```
        .text
main:
        li $v0, 5
        syscall

        add $a0, $v0, $v0
        add $a0, $a0, $a0
        li $v0, 9
        syscall
        move $t0 ,$v0

        move $t1, $v0
while:
        beq $t1, $zero, end
        addi $t1, $t1, -1

        add $t2, $t1, $t1
        add $t2, $t2, $t2
        add $t2, $t0, $t2
        sw $t1, 0($t2)
        j while
end:
        jr $ra
```

$v0*4 Byte

$t1: 配列のサイズ

$t0 + $t1*4 Byte

```
while(i == 0) {
    i--;
    b[i] = i
}
```

動的配列 (サンプル)

- データ数を入力させ、
  そのサイズnの配列bを
  b[i]=iと初期化するプログラム

```
while(i == 0) {
    i--;
    b[i] = i
}
```

m8.s

```
        .text
main:
        li $v0, 5
        syscall

        move $t1, $v0

        add $a0, $v0, $v0
        add $a0, $a0, $a0
        li $v0, 9
        syscall
        move $t0 ,$v0

while:
        beq $t1, $zero, end
        addi $t1, $t1, -1

        add $t2, $t1, $t1
        add $t2, $t2, $t2
        add $t2, $t0, $t2
        sw $t1, 0($t2)
        j while
end:
        jr $ra
```

$t1: 配列のサイズ

$v0*4 Byte

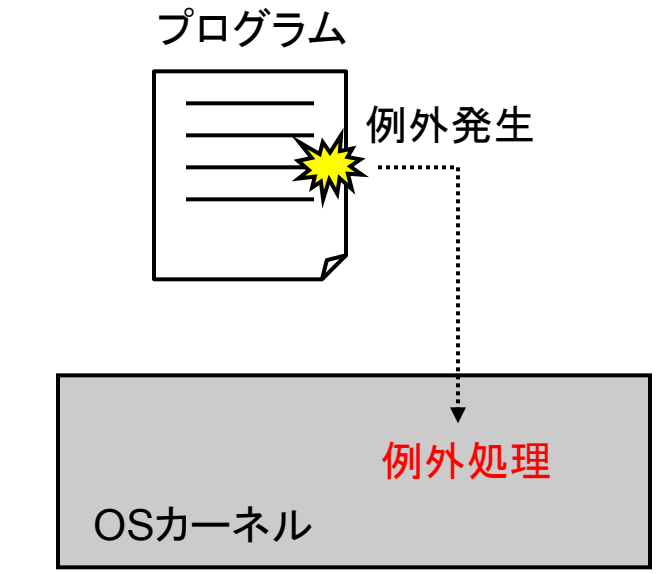
$t0 + $t1*4 Byte

例外処理

- Java の例外処理
  - 自分で書いた catch 文の中の処理が行われる

```
try {
        :
        :
} catch (Exception e) {
        例外処理
}
```

- CPU の例外処理
  - OSカーネル内で用意された処理が行われる



※ OSカーネル
- OSの基本機能を実装したプログラム群
- e.g,) 入出力、メモリ管理、プロセス間通信

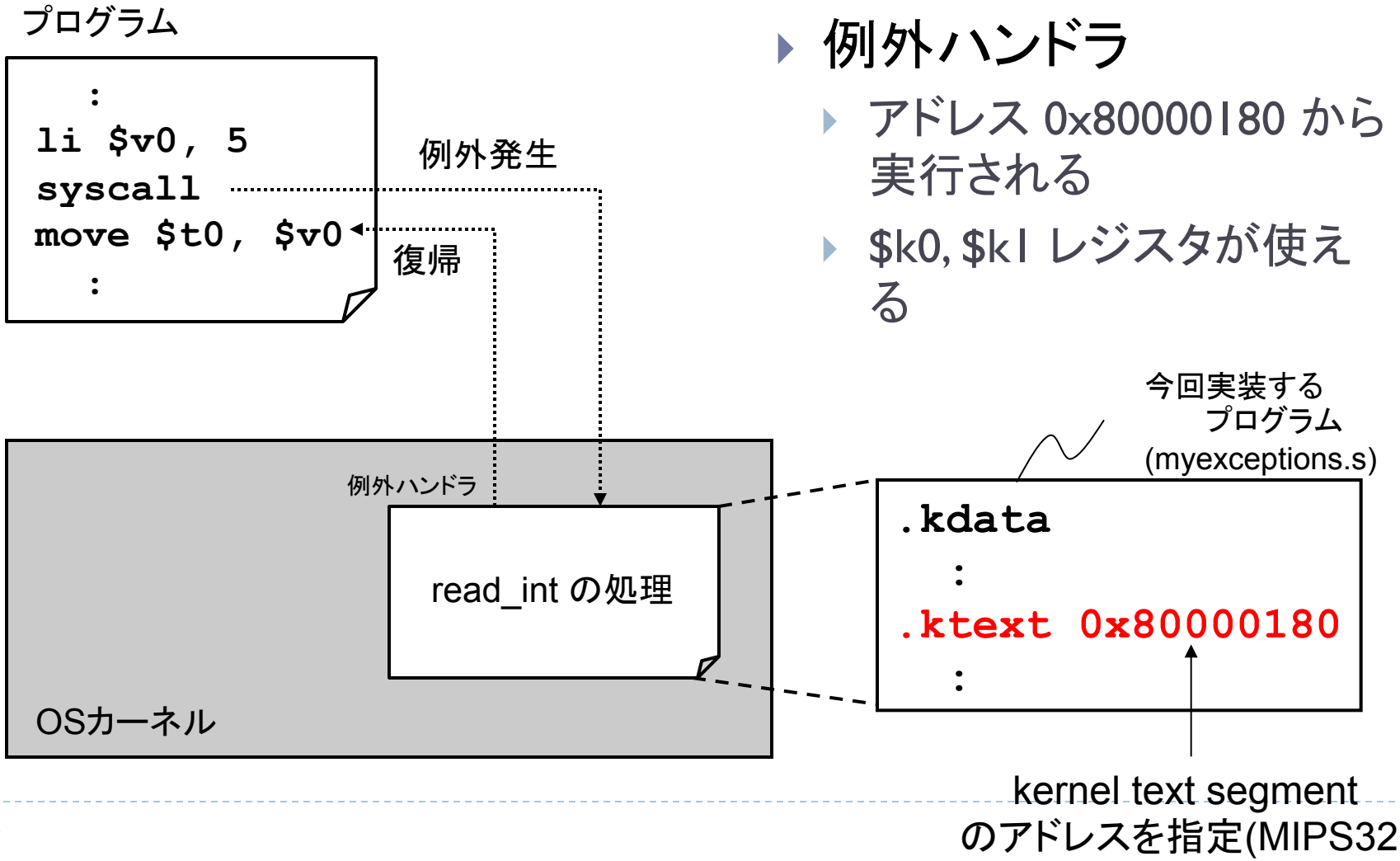
例外の種類

- プログラムの不正実行で発生するもの
  - 不正アドレスのアクセス
    - 例：`lw $t0, 0x00000001`
    - Exception 4 [Address error in inst/data fetch]
- プログラムから明示的に発生させるもの
  - OSカーネルでしか使えない機能を利用するため
    - ファイル入出力
    - メモリ割り当てなど

プログラムから明示的に発生させる例外

例外番号	内容
8	syscall 例外
13	トラップ例外

例外処理の流れ



- 例外ハンドラ
  - アドレス 0x80000180 から実行される
  - $k0, $k1 レジスタが使える

例外ハンドラの実装の流れ

1. レジスタ退避
   - $at
   - 例外ハンドラ内で使用するレジスタ
2. 例外番号の取得
3. （実行したい処理）
4. PC（プログラムカウンタ）の調整
5. レジスタ復帰
   - 例外ハンドラ内で使用したレジスタ
   - $at
6. 例外ハンドラ終了命令(eret)

1. レジスタの退避

- 例外ハンドラ内で使うレジスタを保存
  - callee-saveで行う (=> $sを使用)
    - caller-saveだと例外が発生する命令ごとにレジスタを退避
  - 例外処理がプログラムの実行に悪影響を与えないようにする

```
        .kdata
save_s0:
        .word 0                         # $s0を保存するメモリ領域

        .ktext 0x80000180
        .set noat                       # $atを扱うことを許可する
        move   $k1, $at                 # 疑似命令が使うレジスタを$k1に保存
        .set at                         # $atを扱うことを不許可にする
        sw     $s0, save_s0             # $s0を使うなら元の値を退避保存
```

擬似命令（再）

- MPIS には無い命令
  - `blt $s1, $s2, label`


```
slt     $at, $s1, $s2
bne     $at, $zero, label
```

- 命令自体はあるが、オペランドが違う命令
  - `sw $t1, A($t0)`


```
lui     $at, 4097       #A
addu    $at, $at, $t0
sw      $t1, 0($at)
```

★

- 擬似命令内では$atレジスタが使用される
  - 例外ハンドラ内で $at レジスタが上書きされると困る（ ★）

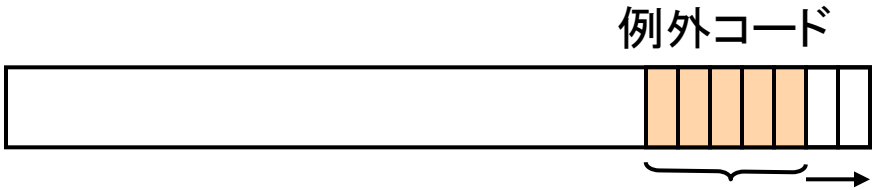
※lui: load upper immediate
※addu: add unsigned

コプロセッサ

- メインプロセッサとは別のプロセッサが例外を管理する
  - 例外番号、例外発生アドレス…など
- コプロセッサのレジスタには mfc0, mtc0 命令でアクセスできる
  - **`mfc0`** *CPUレジスタ，コプロセッサレジスタ*
    - コプロセッサのレジスタの内容をCPUレジスタにコピー
  - **`mtc0`** *CPUレジスタ，コプロセッサレジスタ*
    - CPUレジスタの内容をコプロセッサのレジスタにコピー

2. 例外番号の取得

- コプロセッサの Cause レジスタに発生した例外番号が入っている
  - $13 レジスタ




```
mfc0   $k0, $13
srl    $k0, $k0, 2   # 2ビット右シフト
andi   $k0, $k0, 0x1f      # 下位5ビットを取り出す
```

※srl: shift right logical

3. 例外ハンドラで行う処理

- 例外に応じた処理をする
  - 不正メモリアクセス
    - エラーを表示
    - プログラムを終了
  - syscall
    - $v0 で指定されたサービスを実行
      - $a0～$a3を引数とする
    - $v0 に結果を代入

4. PC の調整

- 例外処理終了後、次の命令から実行再開
  - （サブルーチン呼び出しの場合は jal 命令が $ra レジスタに次の命令のアドレスを入れてくれた）
  - EPC（Exception Program Counter）レジスタに例外が発生した命令のアドレスが入っている
    - コプロセッサの $14 レジスタ
  - EPC の値を 4 増やせば次の命令から再開できる

```
mfc0   $k0, $14
addiu  $k0, $k0, 4
mtc0   $k0, $14
```

5. レジスタの復帰と6. 例外ハンドラ終了

- 例外発生時と全てのレジスタの内容を同じにしておく必要がある
  - syscall の場合は $v0 だけ変わる
- eret 命令で EPC レジスタが指すアドレスから実行を再開

```
lw     $s0, save_s0          # $s0を復帰
.set noat                    # $atを扱うことを許可
move   $at, $k1              # $atを復帰
.set at

eret
```

スタートアップルーチン

- myexceptions.s には main ルーチンを呼び出す初期ルーチンも書かなければならない

```
        .text

        .globl __start          # 他のモジュールから
                                呼び出せるようにする

__start:
        jal   main              # mainを呼び出す

        li    $v0, 10
        syscall                 # exitシステムコール
```

⇦ 注：globalではない

まとめ：例外ハンドラ

```
        .kdata
        #1. 退避用の配列の宣言
        .ktext 0x80000180
        # 1. 例外ハンドラで使用するレジスタを退避($atと$s0など）
        # 2. Causeレジスタから例外番号を取得
        # 3. 実行したい処理
        # 4. EPCレジスタを4進める
        # 5. 退避したレジスタを復帰
        eret
          :
        .text
        .globl __start
__start:
        # mainルーチンを呼び出す初期ルーチン
```

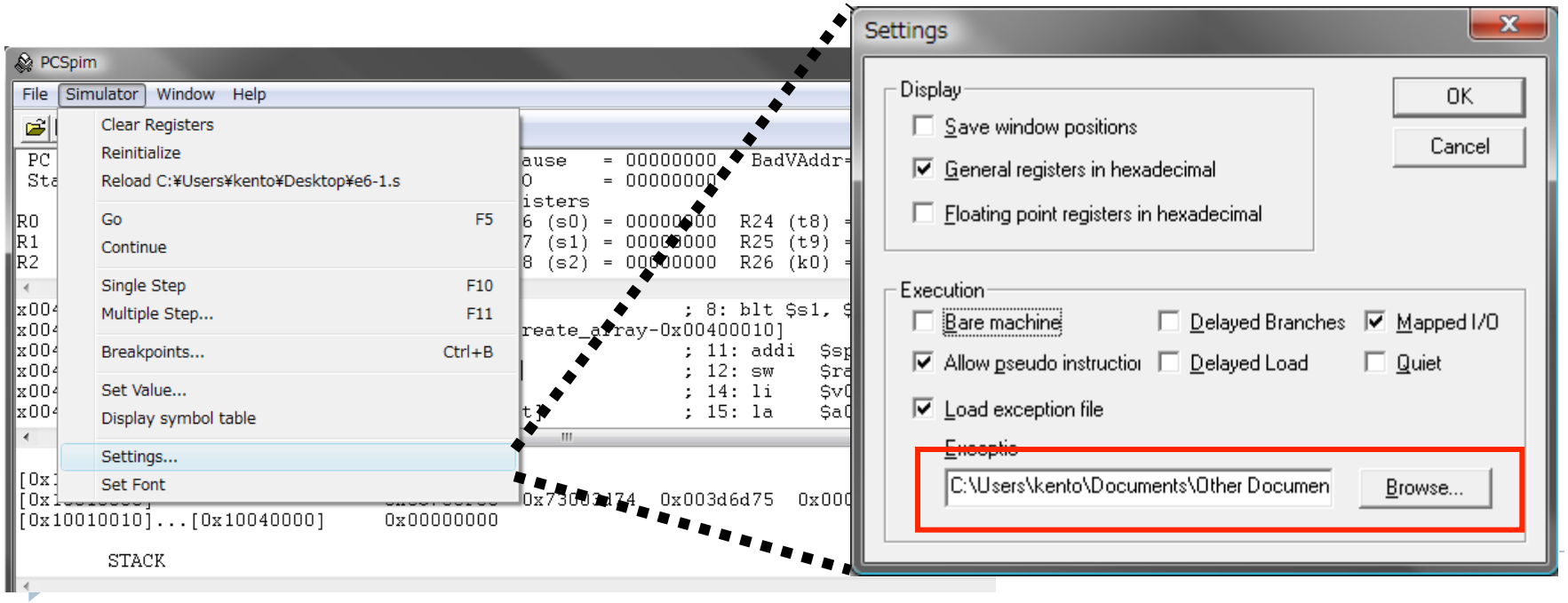
SPIM での例外ハンドラ

- デフォルトで <install directory>/lib/exceptions.s が使われている
  - 変更するには -exception_file オプションをつけて xspim を起動すればよい
    - xspim –exception_file [path]
    - Loaded: myexceptions.s と表示される

```
$ xspim –exception_file ./Kadai/ex06/myexceptions.s
```

SPIMでの例外ハンドラの設定

- デフォルトではC:\ProgramFiles\PCSpim\exceptions.sが使用されている
  - 変更するには[Simulator] > [Settings…] > Exceptioの[Browse…]でmyexceptions.sを指定



注意

- 次回SPIMを実行したとき以前使用した例外ファイルを使用
- デフォルトファイルへ戻すときは明示的に設定する必要がある
  - windows: C:\Program Files\PCSpim\exceptions.s を指定
  - mac OSX: ~/xspim/lib/exception.s を指定

補足：全32本のレジスタ

使用できるレジスタ

Name	Register number	Usage
$zero	0	the constant value 0
$v0-$v1	2-3	values for results and expression evaluation
$a0-$a3	4-7	arguments
$t0-$t7	8-15	temporaries
$s0-$s7	16-23	saved
$t8-$t9	24-25	more temporaries
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address

使用できないレジスタ

1 → $at
(assembler temporary)
reserved by the assembler

26-27 → $k0-$k1
reserved for use by the interrupt/trap handler

課題

課題1

- データ数、データを入力させ、全てのデータの和を計算し、出力するプログラムを書け
- 次の2つのサブルーチンを実装すること
  - create_array
    - 引数：配列の要素数($a0)
    - 戻り値：配列の先頭アドレス($v0)
    - 要素数分の動的配列を作成し、受け取ったデータで初期化する
      - 内部で read_int syscallを **データ数** 回呼び出してよい
  - calc_sum
    - 引数：配列の先頭アドレス($a0)、配列のサイズ($a1)
    - 戻り値：和($v0)
  - caller-save、あるいはcallee-saveで適切にレジスタを退避・復元すること

実行例

```
count=5
1
3
2
4
5
sum=15
```

課題1：ヒント

```
 # 必要に応じてレジスタを退避させる
 # 必要に応じてプロンプト(count=など)
 # を表示させること
  .text
main:

  # 要素数を読込む
  li    $v0, 5
  syscall
  move  $t0, $v0
  # create_arrayを呼び出す
  move  $a0, $t0
  jal   create_array
  # calc_sumを呼び出す
  move  $a0, $v0
  move  $a1, $t0
  jal   calc_sum
  # ナビゲーション（sum=など）を表示
  # 結果を表示

create_array
  # $a0: 作成する配列のサイズ
  # $v0: 配列の先頭アドレス
  # read_int syscallを $a0 回呼び出す
  # li    $v0, 5
  # syscall
  # move  $t0, $v0

calc_sum
  # $a0: 配列の先頭アドレス
  # $a1: 配列のサイズ
  # $v0: 和の結果
```

課題2

- myexceptions.s を作成し、引数の値を10倍するシステムコールを作れ
  - 引数: $a0 (10したい値)、サービス番号 : $v0=100
  - 返り値 : $v0 ($a0 * 10)
  - ここではトラップ例外をわざと発生させる
    - **teq $zero, $zero** でトラップ例外を発生させられる

※ teq:Trap if Equal

```
li      $v0, 100                # システムコール番号100
li      $a0, 55                 # 10倍したい値
teq     $zero, $zero            # トラップ例外を発生させる

move    $a0, $v0
li      $v0, 1
syscall                         # 10倍した値を表示
```

新しいシステムコールを使うプログラム例

課題2：補足

myexceptions.s

```
            .kdata
            #1.  退避用の配列の宣言
            .ktext 0x80000180
            # 1. 例外ハンドラで使用するレジスタを退避($atと$t0など)
            # 2. Causeレジスタから例外番号を取得
            # 3. 例外番号が13以外ならdoneにジャンプ
            # 3. システムコール番号($v0)が100以外ならdoneにジャンプ
            # 3. 引数($a0)の値を10倍(mul命令)して$v0に代入
done:
            # 4. EPCレジスタを4進める
            # 5. 退避したレジスタを復帰
            eret
              :
            .text
            .globl __start
__start:
            # mainルーチンを呼び出す初期ルーチン
```

課題提出

- 〆切： 6/15 (金) 23:59
- 提出物：以下のファイルを圧縮したもの
  - ドキュメント(pdf,plain txt,wordなんでも可)
    - 課題1,2の実行結果
    - 感想、質問等
  - プログラムソース
    - 課題1,2
- 提出方法： Webから提出
  - 授業のページからリンク
  - パスワードを忘れてしまった人は佐藤まで