# システムプログラミング 2
# 期末レポート

氏名: 山田　敬汰 (Yamada，Keita)
学生番号: 09430559

出題日: 2019 年 12 月 9 日
提出日: 2019 年??月??日
締切日: 2020 年 1 月 27 日

## 1　概要

本演習では，前回の演習で学習したアセンブリ言語について，システムコールを用いた C 言語との連携，C 言語のプログラムをコンパイルした際の手続き呼び出し規約に基づいたスタックの取り扱い，auto 変数と static 変数という宣言方法によって異なる二種類の変数のライフサイクル，そしてアセンブリ言語での可変引数関数の実現についての考察および実装等，これまでよりもより深い部分について理解を深めた．以下に，今回の授業内で実践した 5 つの演習課題についての詳しい内容を記述する．

## 2　課題 2-1

### 2.1　課題内容

SPIM が提供するシステムコールを C 言語から実行できるようにしたい． 教科書 A.6 節 「手続き呼出し規約」に従って，各種手続きをアセンブラで記述せよ． ファイル名は，syscalls.s とすること．

また，記述した syscalls.s の関数を C 言語から呼び出すことで， ハノイの塔 (hanoi.c とする) を完成させよ．

### 2.2　C 言語で記述したプログラム例

```
1 #include <stdio.h>
2
3 void hanoi(int n, int start, int finish, int extra)
4 {
5     if (n != 0)
6     {
7         hanoi(n - 1, start, extra, finish);
8         print_string("Move disk ");
9         print_int(n);
```

```
10          print_string(" from peg ");
11          print_int(start);
12          print_string(" to peg ");
13          print_int(finish);
14          print_string(".\n");
15          hanoi(n - 1, extra, finish, start);
16      }
17  }
18  main()
19  {
20      int n;
21      print_string("Enter number of disks> ");
22      n = read_int();
23      hanoi(n, 1, 2, 3);
24  }
```

### 2.3 作成したプログラム

```
 1  .text
 2  .align 2
 3
 4  _print_int:
 5   subu    $sp,$sp,24
 6   sw      $ra,20($sp)
 7
 8   li      $v0, 1
 9   syscall
10
11   lw      $ra,20($sp)
12   addu    $sp,$sp,24
13   j       $ra
14
15  _print_string:
16   subu    $sp,$sp,24
17   sw      $ra,20($sp)
18
19   li      $v0, 4
20   syscall
21
22   lw      $ra,20($sp)
23   addu    $sp,$sp,24
24   j       $ra
25
26  _read_int:
```

```
27  subu    $sp,$sp,24
28  sw      $ra,20($sp)
29
30  li      $v0, 5
31  syscall
32
33  lw      $ra,20($sp)
34  addu    $sp,$sp,24
35  j       $ra
36
37 _read_string:
38  subu    $sp,$sp,24
39  sw      $ra,20($sp)
40
41  li      $v0, 8
42  syscall
43
44  lw      $ra,20($sp)
45  addu    $sp,$sp,24
46  j       $ra
47
48 _exit:
49  li      $v0, 10
50  syscall
51
```

### 2.4 実行結果

```
Enter number of disks>  4
Move disk 1 from peg 1 to peg 3.
Move disk 2 from peg 1 to peg 2.
Move disk 1 from peg 3 to peg 2.
Move disk 3 from peg 1 to peg 3.
Move disk 1 from peg 2 to peg 1.
Move disk 2 from peg 2 to peg 3.
Move disk 1 from peg 1 to peg 3.
Move disk 4 from peg 1 to peg 2.
Move disk 1 from peg 3 to peg 2.
Move disk 2 from peg 3 to peg 1.
Move disk 1 from peg 2 to peg 1.
Move disk 3 from peg 3 to peg 2.
Move disk 1 from peg 1 to peg 3.
Move disk 2 from peg 1 to peg 2.
Move disk 1 from peg 3 to peg 2.
```

### 2.5 プログラムの解説

ここでは_print_int 部分での手続きを例に解説する．

1. スタックの領域を 24 バイト確保し，戻りアドレスを格納しておく．戻りアドレスを退避させている理由としては，syscall 内の手続きが OS に一任され，アセンブリのプログラムから分からないようになっているからである．(つまり，OS が勝手に $\$ra$ レジスタの値を壊している可能性を考慮している．)

2. $\$v0$ レジスタに適切な番号（_print_int の場合は 1）を格納し，syscall 命令を発行する．

3. スタックに格納しておいた戻りアドレスを $\$ra$ レジスタに再び格納し，呼び出し元に帰る．

その他の手続きも $\$v0$ レジスタの値を変更することで，同様の手順で実行することができる．(OS によって抽象化されている．)

また，exit 手続きのみ，スタックに戻りアドレスを格納していない．その理由としては syscall の発行によってプロセスが終了するので，値を退避させたところで復帰させる方法がないからである．ただ，手続き呼び出しを厳密に守る場合はスタックへの退避の手続きを記述する場合もある．その場合でも実行結果は変わらない．

### 2.6 考察

今回のプログラムでは syscall を呼び出す部分のみをアセンブリ言語で記述している．これは，C 言語の中から直接 syscall 命令を発行する方法が存在しないからである．そして，実行する時に C 言語の部分をアセンブリ言語にコンパイルし，syscalls.s と共に正しい順序でメモリ上に読み込むことで，プログラムを実行することができる．ここで，ファイル読み込みを間違えた場合は関数の参照先が未定義となり，プログラムが例外を発生する．

また，手続き呼び出し規約によって「引数はどのレジスタに入っていて，戻り値はこのレジスタに入っている」というのが決められているので，この規約を守っている限りは C 言語とアセンブリ言語との連携を円滑に行うことができる．言い換えれば，プログラマはコンパイラがどのように C 言語のプログラムを変換するのか（レジスタを決定するルール）を知っていなければアセンブリ言語を書くことができない，ということである．

## 3 課題 2-2

### 3.1 課題内容

hanoi.s を例に spim-gcc の引数保存に関するスタックの利用方法について，説明せよ．そのことは，規約上許されるスタックフレームの最小値 24 とどう関係しているか．このスタックフレームの最小値規約を守らないとどのような問題が生じるかについて解説せよ．

### 3.2 与えられたプログラム

```
 1        .file 1 "hanoi.c"
 2
        （中略）
13
```

```
14
15      .rdata
16      .align 0
17      .align 2
18 $LC0:
19      .ascii "Move disk \000"
20      .align 2
21 $LC1:
22      .ascii " from peg \000"
23      .align 2
24 $LC2:
25      .ascii " to peg \000"
26      .align 2
27 $LC3:
28      .ascii ".\n\000"
29      .text
30      .align 2
31      .set nomips16
32 _hanoi:
33      subu $sp,$sp,24
34      sw $ra,20($sp)
35      sw $fp,16($sp)
36      move $fp,$sp
37      sw $a0,24($fp)
38      sw $a1,28($fp)
39      sw $a2,32($fp)
40      sw $a3,36($fp)
41      lw $v0,24($fp)
42      beq $v0,$zero,$L3
43      lw $v0,24($fp)
44      addu $v0,$v0,-1
45      move $a0,$v0
46      lw $a1,28($fp)
47      lw $a2,36($fp)
48      lw $a3,32($fp)
49      jal _hanoi
50      la $a0,$LC0
51      jal _print_string
52      lw $a0,24($fp)
53      jal _print_int
54      la $a0,$LC1
55      jal _print_string
56      lw $a0,28($fp)
57      jal _print_int
```

```
 58      la $a0,$LC2
 59      jal _print_string
 60      lw $a0,32($fp)
 61      jal _print_int
 62      la $a0,$LC3
 63      jal _print_string
 64      lw $v0,24($fp)
 65      addu $v0,$v0,-1
 66      move $a0,$v0
 67      lw $a1,36($fp)
 68      lw $a2,32($fp)
 69      lw $a3,28($fp)
 70      jal _hanoi
 71 $L3:
 72      move $sp,$fp
 73      lw $ra,20($sp)
 74      lw $fp,16($sp)
 75      addu $sp,$sp,24
 76      j $ra
 77      .rdata
 78      .align 0
 79      .align 2
 80 $LC4:
 81      .ascii "Enter number of disks> \000"
 82      .text
 83      .align 2
 84      .set nomips16
 85 main:
 86      subu $sp,$sp,32
 87      sw $ra,28($sp)
 88      sw $fp,24($sp)
 89      move $fp,$sp
 90      la $a0,$LC4
 91      jal _print_string
 92      jal _read_int
 93      sw $v0,16($fp)
 94      lw $a0,16($fp)
 95      li $a1,1 # 0x1
 96      li $a2,2 # 0x2
 97      li $a3,3 # 0x3
 98      jal _hanoi
 99      move $sp,$fp
100      lw $ra,28($sp)
101      lw $fp,24($sp)
```

表 1: スタックを用いた引数保存

| $sp | offset | 内容 | 備考 | 行番号（格納処理の場所） |
|---|---|---|---|---|
| : | : | : | : | : |
| 31 行目での$sp | -56 | $a0 | 第 1 引数 | 35 行目（2 回目） |
| | -52 | $a1 | 第 2 引数 | 36 行目（2 回目） |
| | -48 | $a2 | 第 3 引数 | 37 行目（2 回目） |
| | -44 | $a3 | 第 4 引数 | 38 行目（2 回目） |
| | -40 | $fp | フレームポインタ | 33 行目 |
| | -36 | $ra | 戻りアドレス | 32 行目 |
| 82 行目での$sp | -32 | $a0 | 第 1 引数 | 35 行目 |
| | -28 | $a1 | 第 2 引数 | 36 行目 |
| | -24 | $a2 | 第 3 引数 | 37 行目 |
| | -20 | $a3 | 第 4 引数 | 38 行目 |
| | -16 | $v0 | 戻り値 | 89 行目 |
| | -12 | | | |
| | -8 | $fp | フレームポインタ | 84 行目 |
| | -4 | $ra | 戻りアドレス | 83 行目 |

```
102     addu $sp,$sp,32
103     j $ra
```

### 3.3 解説

spim-gcc の引数保存について，表 1 に示す．具体的な保存手順について説明すると，以下のようになる．

1. スタックを 32 バイト確保し，$fp,$ra レジスタの値を退避させる．(82-85 行目)
2. hanoi ラベルにジャンプした後，スタックをさらに 24 バイト確保し，$fp,$ra レジスタの値を退避させる．(31-33 行目)
3. 1 で確保したスタックの空き領域に，($a0 – $a3) を退避させる．(呼び出された側で退避の判断を行うことで，余計な書き込み処理を減らすことが可能となる) (35-38 行目)
4. hanoi ラベルに再帰的にジャンプするので，必要な回数だけ 2-3 間の処理を繰り返す．

また，規約上許されてるスタックフレームの最小値が 24 となっているのは，被呼び出し関数が退避させる可能性のある 4 つの引数レジスタ ($a0 – $a3)，呼び出し関数の戻りアドレス ($ra)，そして呼び出し関数のフレームポインタ ($fp) のための領域を確保する必要があるからである (レジスタ 6 個分)．

このスタックフレームの最小値について，守らない場合どうなるかについて以下に解説する．

- 呼び出された側が守らない場合：自身が確保した領域内に引数レジスタ ($a0 – $a3) を退避するので，不具合は発生しない．(呼び出し側が無駄な空き領域を確保しただけで済む)
- 呼び出した側が守らない場合：呼び出された側は，呼び出し側が引数レジスタ ($a0 – $a3) を保存するためのスタック領域を確保してくれている前提で引数レジスタ ($a0 – $a3) を

退避させようとするので，呼び出し側がスタックに格納している値を破壊してしまい，不具合が発生する．

# 4 課題 2-3

## 4.1 課題内容

以下のプログラム report2-1.c をコンパイルした結果をもとに， auto 変数と static 変数の違い，ポインタと配列の違いについてレポートせよ．

## 4.2 与えられたプログラム

```
 1 int primes_stat[10];
 2
 3 char *string_ptr = "ABCDEFG";
 4 char string_ary[] = "ABCDEFG";
 5
 6 void print_var(char *name, int val)
 7 {
 8     print_string(name);
 9     print_string(" = ");
10     print_int(val);
11     print_string("\n");
12 }
13
14 main()
15 {
16     int primes_auto[10];
17
18     primes_stat[0] = 2;
19     primes_auto[0] = 3;
20
21     print_var("primes_stat[0]", primes_stat[0]);
22     print_var("primes_auto[0]", primes_auto[0]);
23 }
```

## 4.3 与えられたプログラム（コンパイル後）

```
 1     .file 1 "2-3.c"
 2
       （中略）
13
14
15     .rdata
```

```
16      .align 0
17      .align 2
18  $LC0:
19      .ascii "ABCDEFG\000"
20      .data
21      .align 0
22      .align 2
23  _string_ptr:
24      .word $LC0
25      .align 2
26  _string_ary:
27      .ascii "ABCDEFG\000"
28      .rdata
29      .align 0
30      .align 2
31  $LC1:
32      .ascii " = \000"
33      .align 2
34  $LC2:
35      .ascii "\n\000"
36      .text
37      .align 2
38      .set nomips16
39  _print_var:
40      subu $sp,$sp,24
41      sw $ra,20($sp)
42      sw $fp,16($sp)
43      move $fp,$sp
44      sw $a0,24($fp)
45      sw $a1,28($fp)
46      lw $a0,24($fp)
47      jal _print_string
48      la $a0,$LC1
49      jal _print_string
50      lw $a0,28($fp)
51      jal _print_int
52      la $a0,$LC2
53      jal _print_string
54      move $sp,$fp
55      lw $ra,20($sp)
56      lw $fp,16($sp)
57      addu $sp,$sp,24
58      j $ra
59      .rdata
```

```
60      .align 0
61      .align 2
62 $LC3:
63      .ascii "primes_stat[0]\000"
64      .align 2
65 $LC4:
66      .ascii "primes_auto[0]\000"
67      .text
68      .align 2
69      .set nomips16
70 main:
71      subu $sp,$sp,64
72      sw $ra,60($sp)
73      sw $fp,56($sp)
74      move $fp,$sp
75      li $v0,2 # 0x2
76      sw $v0,_primes_stat
77      li $v0,3 # 0x3
78      sw $v0,16($fp)
79      la $a0,$LC3
80      lw $a1,_primes_stat
81      jal _print_var
82      la $a0,$LC4
83      lw $a1,16($fp)
84      jal _print_var
85      move $sp,$fp
86      lw $ra,60($sp)
87      lw $fp,56($sp)
88      addu $sp,$sp,64
89      j $ra
90
91      .comm _primes_stat,40
```

### 4.4 考察

まず，auto 変数と static 変数の違いについて考察する．auto 変数とは，関数内で動的に確保され，その関数が終了するまで値を保持し続ける変数である．今回与えられたプログラム内では primes_auto が該当する．アセンブリコード内を見ると main ラベルの直後で 64 バイトのスタックが確保されており，primes_auto の値を保存するためにスタック領域が利用されているということが分かる．ここで，確保するスタックのサイズが 64 バイトとなっているのは，手続き呼び出し規約で定められている最低限のスタックサイズ 24 バイトと primes_auto の値を格納するために必要な領域（$4 \times 10 = 40$ バイト）の和が 64 バイトだからである．これを確かめるために，簡易的な C 言語のプログラムを作成した（auto 変数に関する処理のみで構成されたプログラム）．

```
1 main()
```

```
 2 {
 3     int primes_auto[2];
 4
 5     primes_auto[0] = 0;
 6     primes_auto[1] = 1;
 7
 8     print_int(primes_auto[0]);
 9     print_int(primes_auto[1]);
10 }
```

これをコンパイルすると以下のようなコードが生成される.

```
 1         .file 1 "2-3-2.c"
 2
                （中略）
13
14
15         .text
16         .align 2
17         main:
18         subu $sp,$sp,32        # 24 + 4 * 2
19         sw $ra,28($sp)
20         sw $fp,24($sp)
21         move $fp,$sp
22         sw $zero,16($fp)
23         li $v0,1 # 0x1
24         sw $v0,20($fp)
25         lw $a0,16($fp)
26         jal _print_int
27         lw $a0,20($fp)
28         jal _print_int
29         move $sp,$fp
30         lw $ra,28($sp)
31         lw $fp,24($sp)
32         addu $sp,$sp,32
33         j $ra
```

このコンパイル結果からも分かるように，関数内で変数を宣言した場合は，関数の開始時に auto 変数の値を確保できるようにスタックを多めに確保している．そして，変数の値はスタックに格納され，処理が終わるとともに値は失われてしまう.（スタックが解放される）

これに対して，static 変数とは，プログラムの開始時に静的に確保され，プログラムの終了時まで値を保持し続ける変数である．今回与えられたプログラムでは primes_stat がこれに該当する．アセンブリコード内を見ると，91 行目のアセンブリ指令.comm の部分でメモリ内に 40 バイト分の領域を確保しており，primes_stat の値を保存するためにこの領域が使用されていることが分かる．この領域はアセンブリ指令で確保された場所であり，プログラム終了まで解放されることはない（つまり，プログラムの終了時まで値を保持することができる）.

表 2: 二種類の変数の比較

| | auto 変数 | static 変数 |
|---|---|---|
| 保存場所 | メモリ（スタック部） | メモリ（データ部） |
| メリット | 関数の中でのみ値を保持しているので<br>メモリ空間の節約ができる．<br>（使うときだけ確保する）<br>スタックに積み上げながら値を保持しているので，再帰呼び出しで書き込みが<br>起きた際でも，別々の場所に値を格納できる． | メモリ内の固定した位置に値を<br>格納しているので，再利用が容易．<br>スタックを使う場合と比較して，<br>読み書きを素早く行うことができる．<br>（確保，解放が不要） |
| デメリット | 同じ値を何度も呼び出す場合だと，<br>スタックの確保および解放処理を<br>無駄に実行する事になる． | 値を格納する位置は一箇所に固定<br>されているため，再帰呼び出しを<br>してしまうと意図せず値を<br>上書きしてしまう危険性がある． |

この二種類の変数のメリットとデメリットについて比較した結果を表 2 にまとめる．

次に，配列とポインタの違いについて考察する．C 言語でこれらを用いる際には，変数に格納されているのはどちらも先頭アドレスでなので，ほぼ同様に扱えるが，コンパイル後のアセンブリコードを見ると，明確な相違点が存在する．それは，配列として宣言している場合は該当ラベルの位置に値がそのまま書かれている（4.3 節のプログラムの 27 行目）のに対し，ポインタとして宣言している場合は該当ラベルの位置に直接値が書かれているのではなく，その値が書かれているラベルが書かれており（4.3 節のプログラムの 24 行目），間接的に値にアクセスしているということである．

この二つを比較するために作成した簡易的な C 言語のプログラムを以下に示す．

```
1  char *string_ptr = "ABCDEFG";
2  char string_ary[] = "ABCDEFG";
3
4  main()
5  {
6      string_ptr = "AB";
7      //string_ary = "AB"; コンパイルエラー発生
8  }
```

これをコンパイルすると以下のようなコードが生成される．

```
1          .file   1 "2-3-3.c"
2
             （中略）
13
14
15         .rdata
16         .align  2
17 $LC0:
18         .asciiz "ABCDEFG"
19         .data
```

```
20          .align  2
21  _string_ptr:
22          .word   $LC0
23          .align  2
24  _string_ary:
25          .asciiz "ABCDEFG"
26          .rdata
27          .align  2
28  $LC1:
29          .asciiz "AB"
30          .text
31          .align  2
32  main:
33          subu    $sp,$sp,8
34          sw      $fp,0($sp)
35          move    $fp,$sp
36          la      $v0,$LC1
37          sw      $v0,_string_ptr
38          move    $sp,$fp
39          lw      $fp,0($sp)
40          addu    $sp,$sp,8
41          j       $ra
```

上記のプログラムから分かるように，配列とポインタの大きな違いとしては，宣言した変数に再び代入処理ができるかどうか，ということが挙げられる．配列の場合では，変数内に文字列の先頭アドレスが定数として保存されているので，別の値を代入することはできずコンパイルエラーが発生する．（添字付きでアクセスするのは問題ない）これに対してポインタの場合では，変数内に格納されているアドレスを自由に付け替えれる様になっているので，再代入が可能であり，それによって元の値が壊されることもない．（アクセスする場所が変わっているだけ．）ただ，ポインタを使った場合では使わなくなった値を適切に解放する処理をしなければ，どこからもアクセスされないデータが蓄積し，メモリを圧迫する場合もあるので注意しなければならない．

# 5　課題 2-4

## 5.1　課題内容

## 5.2　可変引数

## 5.3　可変引数（C 言語）

## 5.4　アセンブリでの実現方法の違い（C との）

## 5.5　実装

# 6　課題 2-5

## 6.1　課題内容

# 7　感想