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 1 月 20 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 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.
```

2.4 プログラムの解説

ここでは 2.6 節に添付したプログラムの_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．

2.5 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．

2.6 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
52 _print_char:
53  subu    $sp,$sp,24
54  sw      $ra,20($sp)
55
56  li      $v0, 11
57  syscall
58
59  lw      $ra,20($sp)
60  addu    $sp,$sp,24
61  j       $ra
62
```

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)
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．

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

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

4.4.1 auto 変数と static 変数

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)
```

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 メモリ空間の節約ができる． （使うときだけ確保する） 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 格納しているので，再利用が容易． スタックを使う場合と比較して， 読み書きを素早く行うことができる． （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， スタックの確保および解放処理を 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 されているため，再帰呼び出しを してしまうと意図せず値を 上書きしてしまう危険性がある．

```
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）.

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.

4.4.2 配列とポインタ

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，プログラマ側が注意する必要がある．

5 課題 2-4

5.1 課題内容

printf など，一部の関数は，任意の数の引数を取ることができる．これらの関数を可変引数関数と呼ぶ．MIPS の C コンパイラにおいて可変引数関数の実現方法について考察し，解説せよ．

5.2 可変引数とは

可変引数とは，引数の数を呼び出す側が任意で変更することができる関数である．例えば，printf 関数を実装する際に，任意の数の引数に対応している場合，引数の数が 2 つで固定されてしまった場合と比較して，関数を呼び出す回数を減らすことができる，というメリットがある．つまり，何か変数の値を表示したい時に，表示したい個数分 printf を呼び出す必要がなくなる．

5.3 可変引数の実現

C 言語で可変引数を扱う際には void printf(char *fmt,...) の様な形式で関数を宣言する．この形式で記述された関数がアセンブリ言語の中でどの様に実現されているのかについて以下に解説する．アセンブリでは可変引数を利用する際に，スタック領域を活用している．課題 2-2 の表 1 でも示している様に，呼び出し側が余分に取っておいたスタック領域に順番に引数を格納しており，もし引数の数が 4 個より多くなったとしても，それに応じてスタックの確保領域を増やし，第 4 引数の次の領域から順に格納していくことで対応する．(関数内で引数に順番にアクセスしやすくするため）この方法を用いることで，スタック領域が埋まらない限りは，何個でも引数の数を増やすことができる．

次に，与えられたそれぞれの引数に対して，どの様にアクセスしているかについて解説する．スタックを用いて引数を順番に確保しているとはいえ，引数の型がそれぞれ異なる場合があるので，アドレスをいくつ加算するかが不明な場合は，正しくデータにアクセスすることができない．そのため，可変引数を用いる際には sizeof 演算子を利用し，与えられた引数に応じて次の引数にアクセスするために，アドレスをいくつ加算すれば良いかを計算している．(例えば，Int 型は 4 バイトであり，char 型は 1 バイト）ちなみに，この sizeof 演算子は 32 ビット CPU と 64 ビット CPU で異なる挙動を示すので，同じソースコードでも動かす環境次第で加算する数を変えなければならない．

5.4 アセンブリでの実装

可変引数についてプログラムを元に説明するため，簡易的なプログラムを作成した．

```
 1 void fun(int a, ...)
 2 {
 3     return;
 4 }
 5
 6 int main()
 7 {
 8
 9     fun(1, 1, 'a', "a", 2, 'b', "b");
10
11     return 0;
12 }
```

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 "2-4.c"
 2
              （中略）
13
14
15 .text
16 .align 2
17 _fun:
18 sw $a0,0($sp)
19 sw $a1,4($sp)
20 sw $a2,8($sp)
21 sw $a3,12($sp)
22 subu $sp,$sp,8
23 sw $fp,0($sp)
24 move $fp,$sp
25 sw $a0,8($fp)
26 move $sp,$fp
27 lw $fp,0($sp)
28 addu $sp,$sp,8
29 j $ra
30 .rdata
31 .align 2
32 $LC0:
33 .asciiz "a"
34 .align 2
35 $LC1:
36 .asciiz "b"
```

表 3: スタックを用いた可変引数の保存

offset	内容	備考
-40	$a0	第 1 引数
-36	$a1	第 2 引数
-32	$a2	第 3 引数
-28	$a3	第 4 引数
-24	$v0	第 5 引数
-20	$v0	第 6 引数
-16	$v0	第 7 引数
-12		
-8	$fp	フレームポインタ
-4	$ra	戻りアドレス

```
37 .text
38 .align 2
39 main:
40 subu $sp,$sp,40
41 sw $ra,36($sp)
42 sw $fp,32($sp)
43 move $fp,$sp
44 li $v0,2 # 0x2
45 sw $v0,16($sp)
46 li $v0,98 # 0x62
47 sw $v0,20($sp)
48 la $v0,$LC1
49 sw $v0,24($sp)
50 li $a0,1 # 0x1
51 li $a1,1 # 0x1
52 li $a2,97 # 0x61
53 la $a3,$LC0
54 jal _fun
55 move $v0,$zero
56 move $sp,$fp
57 lw $ra,36($sp)
58 lw $fp,32($sp)
59 addu $sp,$sp,40
60 j $ra
```

ここで注目すべきは 40-53 行目である．この処理の中でのスタック内の要素について図示すると表 3 の様になり，先ほど説明した様に，スタックを必要に応じて拡張することで，可変引数に対応することが可能となっている，ということが分かる．また，手続き呼び出し規約のために，時系列的には第 1 引数〜第 4 引数よりも先に第 5 引数〜第 7 引数がスタックに格納されているが，第 4 引数の次のアドレスに第 5 引数が格納される様になっているので，引数が並び替えられていいるわけではない．

表 4: 実装したサブセット

サブセット	種別	型	内容
%c	変換指定子	char	任意の一文字を出力
%s	変換指定子	char*	任意の文字列を出力
%d	変換指定子	int	任意の数値を出力（10 進数表記）
%o	変換指定子	int	任意の数値を出力（8 進数表記）
%x	変換指定子	int	任意の整数を出力（16 進数表記）
%-	フラグ	無し	左詰め
%0	フラグ	無し	0 埋め
%+	フラグ	無し	符号を表示
%n（任意の整数）	その他	無し	最小表示文字数
%.n （任意の整数）	その他	無し	最大表示文字数（%s でのみ動作）
%%	その他	無し	”%”を出力

6 課題 2-5

6.1 課題内容

`printf` のサブセットを実装し， SPIM 上でその動作を確認する応用プログラム (自由なデモプログラム) を作成せよ．フルセットにどれだけ近いか，あるいは，よく使う重要な仕様だけをうまく切り出して，実用的なサブセットを実装しているかについて評価する．ただし，浮動小数は対応しなくてもよい (SPIM 自体がうまく対応していない)．加えて，この `printf` を利用した応用プログラムの出来も評価の対象とする．

6.2 プログラムの作成方針

今回のプログラムを作成するに当たって，以下の部分から構成することにした．

- 作成したサブセットそれぞれをテストする部分
- 与えられたフォーマット指定子を元に，引数を適切に出力する部分
- 表示桁数や 0 埋めなどの指定を元に，要素を修飾していく部分

6.3 プログラムおよびその説明

以下では，前節の作成方針における分類に基づいて，プログラムの大まかな構造について解説する．また，プログラムのソースコードについては 6.6 節に添付している．

6.3.1 テスト部（5 行目から 29 行目）

この部分では，表 4 に示された今回実装したサブセットについて，網羅的にテストを行っている．

6.3.2　引数出力部（159 行目から 322 行目）

この部分では，与えられた文字列 fmt を解析し，%があれば，後述する要素修飾部を実行した後，表 4 の変換指定子を元にどの型が可変引数として与えられたかを推定し，システムコールを用いて値を出力する．

6.3.3　要素修飾部（174 行目から 205 行目）

この部分では fmt を解析し，%を発見した後，表 4 のフラグや表示桁数を読み取り，出力する文字を適切に装飾する．

6.4　プログラム作成過程における考察

6.4.1　可変引数の取り扱い

課題のプログラムを作成する上で，まず最初に気をつけた点は可変引数の取り扱いである．前の課題で述べた様に，与えられた引数の型に応じて加算するアドレスの大きさを変えなければならないので，switch 文で分岐し，出力処理が終わったタイミングでアドレスの加算処理を行う様にした（分岐するまでは，想定されている引数の型が不明なので）．

6.4.2　n 進数表記への対応

8 進数表記や 16 進数に対応するために，再帰呼び出しに対応した関数 print_base（138 行目）を作成した．なぜ再帰呼び出しにする必要があったのかという理由については，進数を変換する過程で最後に出力すべき数字から順に値が求まる様になっているからである．

この関数の処理の流れは以下の通りである．

1. 引数として，任意の数字（num）および基数 base を受け取る．
2. num ÷ base を計算し，商と余りを求める．
3. 商が 0 でなければ，その商と基数を引数として，再帰的に関数を呼び出す．
4. 商が 0 になるまで上記の手順を繰り返す．（スタックに表示待ちの数字を積み上げている）
5. 商が 0 になると，スタックに積み上げられた数字を順に取り出し出力する．
6. 出力する際，10 以上の数字はアルファベットに変換して表示する．（'a' との差を求める）

この関数は，引数として基数も受け取れるようにしているので，8 進数や 16 進数以外の表示することも可能である．（今回のプログラムでは 10 + 26 で 36 進数まで対応可能）

6.4.3 各種フラグの指定

0 埋めや左詰めに対応するための処理について解説する．まず，176-197 行目で fmt に各種フラグに対応する文字が含まれているかどうかを解析し，フラグを適切に切り替える．当初，この部分の処理は単一の switch 文を用いて制御していたが，複数のフラグが同時に指定された時に対応できなくなるという不具合が発覚し，while ループで該当部分を囲むことによって，不具合を解消した．

そして，文字列表示の際に各種フラグの値に対応した処理を挟むことで，表示する文字列の装飾が可能となった．フラグに応じた処理の詳細について以下に示す．

- 0：右詰めの際，左側の余白を 0 で埋める．（空白文字を表示する代わりに 0 を表示する）
- -：表示する文字列を左詰めにする．（空白文字出力と引数出力の順序を入れ替える）
- +：正の値の出力時に，前方に「+」を付加する．

6.4.4 表示桁数の指定

最小表示文字数および，最大表示文字数の指定に対応するため，以下の様な処理を実装した．

1. 最小表示文字数，最大表示文字数を格納する変数を初期化する．（leftrange，rightrange）
2. fmt が 1-9 までの数字であれば，現在格納している数を 10 倍した後*fmt と'0' の差分を leftrange に加算する．
3. fmt が「.」であれば加算する対象を rightrange に変更し 2 の処理をもう一度行う，
4. 出力する際，左詰めフラグの値に応じて，文字列の左右どちらかに leftrange から文字数または桁数を引いた値の数だけ空白文字または 0 を出力する．
5. %s の時のみ，表示する際に rightrange の値を出力できる文字数の上限とする．

この処理を実装するに当たって，当初は数字を 10 で割り続けて桁数を取得していたが（55 行目の get_digit 関数），そのままでは一般性がなく 8 進数や 16 進数に対応できていない，という不具合があったため，print_base 関数同様，引数に基数を受け取る様にし，正しい桁数を取得できる様にした．

6.5 実行結果

```
one character [a] = [a]
string [hello] = [hello]
decimal [100] = [100]
octal [100] = [144]
hexadecimal [100] = [64]
escape [%] = [%]
max 10 characters [hello] = [     hello]
limit two characters [hello] = [he]
max 5 digits decimal [100] = [  100]
max 5 digits octal [100] = [  144]
```

```
max 5 digits hexadecimal [100] = [   64]
zero padding max 5 digits decimal [100] = [00100]
zero padding max 5 digits octal [100] = [00144]
zero padding max 5 digits hexadecimal [100] = [00064]
left max 5 digits decimal [100] = [100  ]
left max 5 digits octal [100] = [144  ]
left max 5 digits hexadecimal [100] = [64   ]
signed decimal [100] = [+100]
signed octal [100] = [+100]
signed hexadecimal [100] = [+100]
```

6.6 作成したプログラム

```
 1 #define ROUNDUP_SIZEOF(a) ((sizeof(a) + 3) / 4 * 4)
 2
 3 void myprintf(char *fmt, ...);
 4
 5 int main()
 6 {
 7     myprintf("one character [a] = [%c]", 'a');
 8     myprintf("string [hello] = [%s]", "hello");
 9     myprintf("decimal [100] = [%d]", 100);
10     myprintf("octal [100] = [%o]", 100);
11     myprintf("hexadecimal [100] = [%x]", 100);
12     myprintf("escape [%%] = [%%]");
13     myprintf("max 10 characters [hello] = [%10s]", "hello");
14     myprintf("limit two characters [hello] = [%.2s]", "hello");
15     myprintf("max 5 digits decimal [100] = [%5d]", 100);
16     myprintf("max 5 digits octal [100] = [%5o]", 100);
17     myprintf("max 5 digits hexadecimal [100] = [%5x]", 100);
18     myprintf("zero padding max 5 digits decimal [100] = [%05d]", 100);
19     myprintf("zero padding max 5 digits octal [100] = [%05o]", 100);
20     myprintf("zero padding max 5 digits hexadecimal [100] = [%05x]", 100);
21     myprintf("left max 5 digits decimal [100] = [%-5d]", 100);
22     myprintf("left max 5 digits octal [100] = [%-5o]", 100);
23     myprintf("left max 5 digits hexadecimal [100] = [%-5x]", 100);
24     myprintf("signed decimal [100] = [%+d]", 100);
25     myprintf("signed octal [100] = [%+d]", 100);
26     myprintf("signed hexadecimal [100] = [%+d]", 100);
27
28     return 1;
29 }
30
31 int max(int a, int b)
```

```
32 {
33     if (a > b)
34     {
35         return a;
36     }
37     else
38     {
39         return b;
40     }
41 }
42
43 int min(int a, int b)
44 {
45     if (a < b)
46     {
47         return a;
48     }
49     else
50     {
51         return b;
52     }
53 }
54
55 int get_digit(int a, int base)
56 {
57     int digit = 0;
58
59     if (a == 0)
60     {
61         return 0;
62     }
63     else
64     {
65         while (a != 0)
66         {
67             a /= base;
68             digit++;
69         }
70         return digit;
71     }
72 }
73
74 int get_range(char *str)
75 {
```

```
 76     int range = 0;
 77
 78     while ((*str >= '0' && *str <= '9'))
 79     {
 80         range = range * 10 + (*str - '0');
 81         str++;
 82     }
 83     return range;
 84 }
 85
 86 char get_char_for_fill(int is_zero)
 87 {
 88     if (is_zero)
 89     {
 90         return '0';
 91     }
 92     else
 93     {
 94         return ' ';
 95     }
 96 }
 97
 98 void print_plus(int is_plus)
 99 {
100     if (is_plus)
101         print_char('+');
102 }
103
104 void print_fill(int size, int is_zero)
105 {
106     int i;
107     for (i = 0; i < size; i++)
108     {
109         print_char(get_char_for_fill(is_zero));
110     }
111 }
112
113 void print_limited_string(int limit, char *str)
114 {
115     int i;
116     for (i = 0; i < limit; i++)
117     {
118         if (*str == '\0')
119         {
```

```
120             break;
121         }
122         print_char(*str++);
123     }
124 }
125
126 int strlen(char *str)
127 {
128     int length = 0;
129
130     while (*str++ != '\0')
131     {
132         length++;
133     }
134
135     return length;
136 }
137
138 void print_base(int num, int base)
139 {
140     int surplus = num % base;
141     int quotient = num / base;
142
143     if (quotient != 0)
144     {
145         print_base(quotient, base);
146     }
147
148     if (surplus >= 10)
149     {
150         print_char('a' + surplus - 10);
151     }
152     else
153     {
154         print_int(surplus);
155     }
156     return;
157 }
158
159 void myprintf(char *fmt, ...)
160 {
161     int leftrange = 0;
162     int rightrange = 0;
163     int read_zero = 0;
```

```
164     int is_left = 0;
165     int is_plus = 0;
166
167     char *p = (char *)&fmt + ROUNDUP_SIZEOF(fmt);
168     while (*fmt)
169     {
170         if (*fmt == '%')
171         {
172             fmt++;
173
174             leftrange = 0;
175             rightrange = 0;
176             read_zero = 0;
177             is_left = 0;
178             is_plus = 0;
179
180             while (*fmt == '0' || *fmt == '-' || *fmt == '+')
181             {
182                 switch (*fmt)
183                 {
184                 case '0':
185                     read_zero = 1;
186                     break;
187                 case '-':
188                     is_left = 1;
189                     break;
190                 case '+':
191                     is_plus = 1;
192                     break;
193                 default:
194                     break;
195                 }
196                 fmt++;
197             }
198
199             leftrange = get_range(fmt);
200             fmt += get_digit(leftrange, 10);
201             if (*fmt == '.')
202             {
203                 rightrange = get_range(++fmt);
204                 fmt += get_digit(rightrange, 10);
205             }
206
207             switch (*fmt)
```

```
208             {
209             case 'c':
210                 print_char(*(char *)p);
211                 p += ROUNDUP_SIZEOF(char);
212                 break;
213             case 's':
214
215                 if (rightrange == 0)
216                 {
217                     rightrange = strlen(*(char **)p);
218                 }
219
220                 if (is_left == 1)
221                 {
222                     print_limited_string(rightrange, *(char **)p);
223                     print_fill(
224                         max(leftrange - strlen(*(char **)p), 0), 0);
225                 }
226                 else
227                 {
228                     print_fill(
229                         max(leftrange - strlen(*(char **)p), 0), 0);
230                     print_limited_string(rightrange, *(char **)p);
231                 }
232                 p += ROUNDUP_SIZEOF(char *);
233                 break;
234             case 'd':
235                 if (is_left == 1)
236                 {
237                     print_plus(is_plus);
238                     print_int(*(int *)p);
239                     print_fill(
240                         max(
241                             leftrange -
242                                 get_digit(*(int *)p, 10) - is_plus,
243                             0),
244                         0);
245                 }
246                 else
247                 {
248                     print_fill(
249                         max(
250                             leftrange -
251                                 get_digit(*(int *)p, 10) - is_plus,
```

```
252                         0),
253                     read_zero == 1);
254                 print_plus(is_plus);
255                 print_int(*(int *)p);
256             }
257             p += ROUNDUP_SIZEOF(int);
258             break;
259         case 'o':
260             if (is_left == 1)
261             {
262                 print_plus(is_plus);
263                 print_base(*(int *)p, 8);
264                 print_fill(
265                     max(
266                         leftrange -
267                             get_digit(*(int *)p, 8) - is_plus,
268                         0),
269                     0);
270             }
271             else
272             {
273                 print_fill(
274                     max(
275                         leftrange -
276                             get_digit(*(int *)p, 8) - is_plus,
277                         0),
278                     read_zero == 1);
279                 print_plus(is_plus);
280                 print_base(*(int *)p, 8);
281             }
282             p += ROUNDUP_SIZEOF(int);
283             break;
284         case 'x':
285             if (is_left == 1)
286             {
287                 print_plus(is_plus);
288                 print_base(*(int *)p, 16);
289                 print_fill(
290                     max(
291                         leftrange -
292                             get_digit(*(int *)p, 16) - is_plus,
293                         0),
294                     0);
295             }
```

```
296                 else
297                 {
298                     print_fill(
299                         max(
300                             leftrange -
301                                 get_digit(*(int *)p, 16) - is_plus,
302                             0),
303                         read_zero == 1);
304                     print_plus(is_plus);
305                     print_base(*(int *)p, 16);
306                 }
307                 p += ROUNDUP_SIZEOF(int);
308                 break;
309             case '%':
310                 print_char('%');
311                 break;
312             default:
313                 break;
314             }
315         }
316         else
317         {
318             print_char(*fmt);
319         }
320         fmt++;
321     }
322     print_char('\n');
323     return;
324 }
```

7 感想

今回の課題を進めていく中で，システムコールの実態や printf の内部実装等，今までブラックボックスだった部分の内部を詳しく知ることができた．今後エンジニアとして生きていく上で，この様な内部実装を詳しく知っているのといないのとでは，プログラムでエラーが発生した時の解決スピードに大きな差が出てくると思うので,「とりあえず動くから大丈夫」で終わるのではなく「実際にプログラムがどの様な処理を実行しているのか」を，ライブラリの内部実装などを含めてきちんと理解しながらプログラムを実装していくのが重要だなと思った．