情報工学科　レポート

実験演習記録			判定・指示
	年月日時	共同作業者	
1			
2			
3			
4			

レポート提出記録		
	提出年月日	期限年月日
初	2018/11/20	2018/11/20
再		

科目名	テーマ担当教員	学年	学期	単位
情報工学実験1	山田先生	2	後期	2

テーマ番号　テーマ名	学籍番号　氏　名
(6) プログラムの動作原理	16268062 SALIC ERTUGRUL

1. 課題 1

課題 1-1

課題 1-1 の出力結果を図 1 に示す．図 1 より結果が a = 6 だったことが分かる．

```
[s160062@pc4k024 ~]$
[s160062@pc4k024 ~]$ gcc assignment1.c
[s160062@pc4k024 ~]$
[s160062@pc4k024 ~]$ ./a.out
a = 6
[s160062@pc4k024 ~]$
[s160062@pc4k024 ~]$
[s160062@pc4k024 ~]$
```

図 1　課題 1-1 の実行結果

課題 1-2

課題 1-2 では comb(4, 2)によって comb()が何回呼ばれるかを求める．そのためにプログラムに，グローバル変数 control を定義し， comb()が呼ばれた度に一つ増やし，結果を出力するコードを追加する．

```
#include <stdio.h>
int counter = 0;
int comb(int n, int r){
   int ret;
   counter++;
   if (n == 0) ret = 1;
   else if (r == 1) ret = n;
   else if (r == 0 || n == r) ret = 1;
   else ret = comb(n-1, r-1) + comb(n-1, r);
   return ret;
}
int main(){
   int a;
   a = comb(4,2);
   printf("a = %d¥n", a);
   printf("comb が%d 回呼ばれている¥n", counter);
   return 0;
   }
```

つまり，課題 1-2 では control という変数を定義し，comb 関数が呼ばれるたびに control を 1 増やすというアルゴリズムを考えた．そのプログラムの実行結果を図 2 に示す．

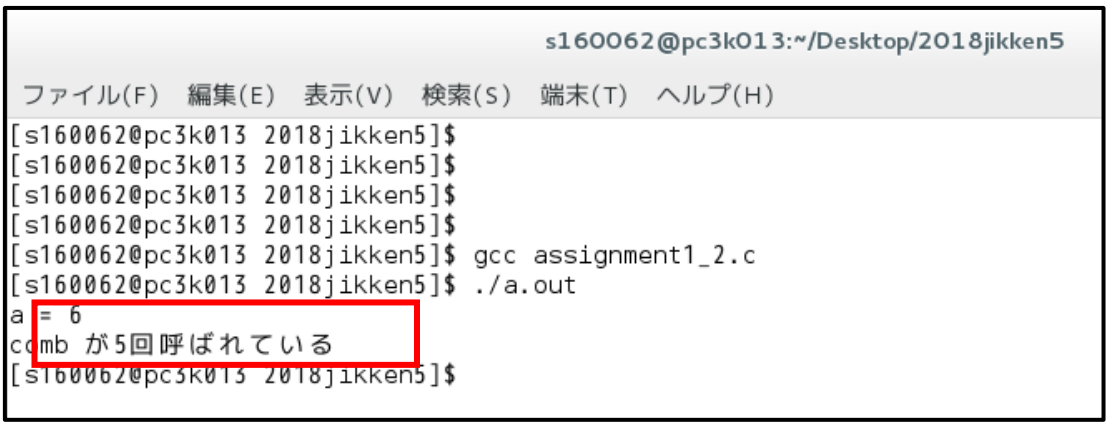

図 2　課題 1-2 の結果

図 2 より，comb(4, 2)によって，comb( )が 5 回呼ばれることが分かる．

課題 1-3

ここでは，comb(4, 2)を実行した時のスタックの様子について説明する．

comb 関数が 5 回呼び出される．（課題 1-2 を参照）comb 関数の頭に，次のコードを追加することによって comb()関数の呼び出される順を求める．その結果を図 3 に示す．

```
printf("comb(%d, %d)が呼び出された¥n", n, r); //追加した行
```

comb 関数の呼び出しが図 3 に示した順になっており，その順でスタックに格納されることになる．

```
[s160062@pc3k013 2018jikken5]$ gcc assignment1_3.c
[s160062@pc3k013 2018jikken5]$ ./a.out
comb(4,2)が呼び出された
comb(3,1)が呼び出された
comb(3,2)が呼び出された
comb(2,1)が呼び出された
comb(2,2)が呼び出された
a = 6
comb が5回呼ばれている
[s160062@pc3k013 2018jikken5]$
```

図 3　comb 関数が呼び出される順

図 3 より comb 関数の呼び出し順番がわかる．その順に従ってスタックの様子を考えた結果を図 4 に示す．簡単のため，レジスタの退避・復元は省略してある．プログラムが起動すると，main()が呼ばれてスタック上に a という局所変数の領域が設けられる．次に，comb(4, 2)が呼ばれる．こ

こで，comb()にジャンプするわけだが，このときに関数の戻りアドレスとなるアドレスと引数として渡した 4 と 2 の値がスタック上に積まれる．その後，comb()を実行する前に局所変数 ret の領域が同様にスタック上に確保される．今，n = 4 そして r = 2 なので，else の方が実行され，comb(3,1)が呼び出される．すると，同様に引数の値，戻り番地，そして局所変数 ret の領域がスタックに設けられる．comb(3,1)を実行するときに ret に 3 が代入されて，それが返り値になる．comb(3,1)へジャンプするときにスタックに積んだ戻りアドレスに戻り，comb(3,2)を呼び出す．またスタックに領域が設けられ，これが，comb(2,1)と繰り返される．comb(2,1)を実行するときに ret に 1 が代入されて，それが返り値になる．すると，comb(2,1)が終了し，comb(2,2)が呼び出される．comb(2,2)を実行するときに ret に 1 が代入されて，それが返り値になる．すると，comb(2,2)が終了し，comb(2,2)へジャンプするときにスタックに積んだ戻り番地に戻る．すると，ret には（comb(2,1) の返り値+comb(2,2)の返り値）= 3 が入り，comb(3,2)は終了する．すると，次は comb(3,2)へジャンプするときに記録した戻り番地へと戻る．同様に，ret には 3((comb(3,1) の返り値) + 3((comb(3,2)の返り値) = 6 が入り，main()関数に戻り，a に 6 が代入される．

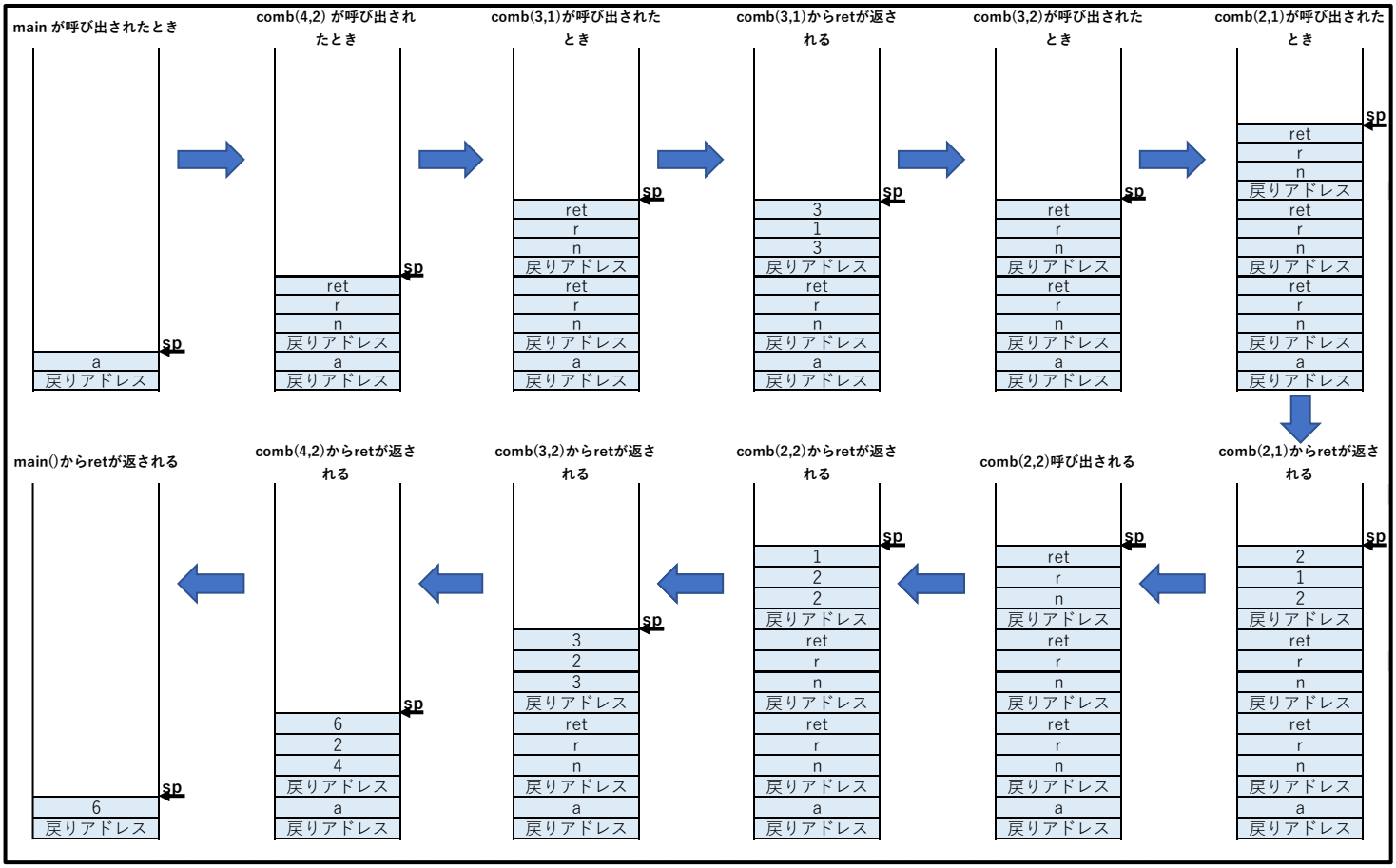

図 4　comb(4, 2)実行時のスタックの様子

課題 1-4

無限に再帰を行うプログラムを動作させる(たとえば f() という関数があり，冒頭で f() を呼び出す)と，Segmentation fault が生じる．本節ではその理由を考察する．セグメンテーション違反の全体的な理由について Alex Allain (2017) がこう説明している．[1]「セグメンテーション違反はアクセスが許可されていないメモリ上の位置にアクセスしようとするときに起こる」．そして，その例として“stack overflow”を挙げる．[1]

問題文によると，課題で扱うプログラムは無限に再帰を行うプログラムである．課題 1，2 と 3 でも触れられた通り，関数が呼び出されると，関数の引数と戻りアドレスが格納されるためにスタックを用いる必要が生じる．しかし，本課題の関数の再帰が無限に行われるので割り当てられたスタックサイズ以上にスタックを使用することになる．よって，アクセス権限のないメモリ領域へのアクセスが生じ，OS が該当プログラムを強制終了する．

2. 課題 2

課題 2-1

課題 2-1 の実行結果を図 5 に示す．プログラムを実行したところ警告が出たが，警告を抑制して実行を続けると*p と*q 両方とも 80 になったという結果が出た．

```
[s160062@pc4k018 ~]$ gcc assignment2.c
assignment2.c: 関数 ‘foo’ 内:
assignment2.c:7:2: 警告: 関数が局所変数のアドレスを返します [-Wreturn-local-addr]
  return &a;
  ^
assignment2.c: 関数 ‘bar’ 内:
assignment2.c:14:2: 警告: 関数が局所変数のアドレスを返します [-Wreturn-local-addr]
  return &b;
  ^
[s160062@pc4k018 ~]$ gcc -w assignment2.c
[s160062@pc4k018 ~]$ ./a.out
*p = 20, *q = 20
*p = 80, *q = 80
[s160062@pc4k018 ~]$
```

図 5　課題 2-1　プログラムの実行結果

課題 2-2

本節では，課題 2-1 の実行結果はなぜそういう結果になったのかを考える．

foo() を呼び出すと，p が foo() の局所変数 a のアドレス，つまり foo() のスタックフレーム内のアドレスを指すことになる．その後，foo() が完了してそのスタックフレームは開放される．次に q = bar() 実行すると，foo() と同位置にそのスタックフレームを作成するため，p は引き続き bar() のスタックフレームのアドレスを指すことになる．具体的には bar() の局所変数 a または b のアドレスとなる．p は bar()の局所変数 a と b のどちらを指しているのかを，プログラム assignment2 のアセンブラコードを確認することによって知ることができる．アセンブラコードの foo()と bar()に当たる部分を次に示す．

---

[1] Debugging Segmentation Faults and Pointer Problems ，<https://www.cprogramming.com/debugging/segfaults.html>，閲覧日：2018 年 11 月 17 日

foo()と bar() のアセンブラコード

```
foo:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $10, -4(%rbp)
        leaq    -4(%rbp), %rax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   foo, .-foo
        .globl  bar
        .type   bar, @function
```

```
bar:
.LFB1:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $20, -4(%rbp)
        leaq    -4(%rbp), %rax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE1:
        .size   bar, .-bar
        .globl  baz
        .type   baz, @function
```

上記のコードより，for()関数の局所変数 a（%rbp-4）と bar()関数局所変数 b（%rbp-4）が同じアドレスを指していることがわかる．よって，p が bar()局所変数 b のアドレスを指していることがわかる．

本プログラムのスタックの様子を図 6 に示す．

bar()を呼び出すと q が bar() の局所変数 b のアドレス，つまり bar() のスタックフレーム内のアドレスを指すことになる．スタックフレームは特に初期化等されないため，p と q の指す先の値を出力すると，p と q 両方とも b の値である 20 が出力する．

次に baz()が呼び出される．すると，foo()と bar() と同位置にそのスタックフレームを作成するため，p と q が引き続き baz() のスタックフレームのアドレスを指すことになる．p と q が baz()局所変数 a と b のどれを指しているのかを，また上記と同様にアセンブラコードで確認できる．アセンブラコードで確認したところ，baz()関数の中では p と q が指している（%rbp-4）のアドレスに 80 が代入されることが分かる．bar() 終了後に p と q の指す先の値を出力すると，baz() の a の値，つまり 80 が出力される．

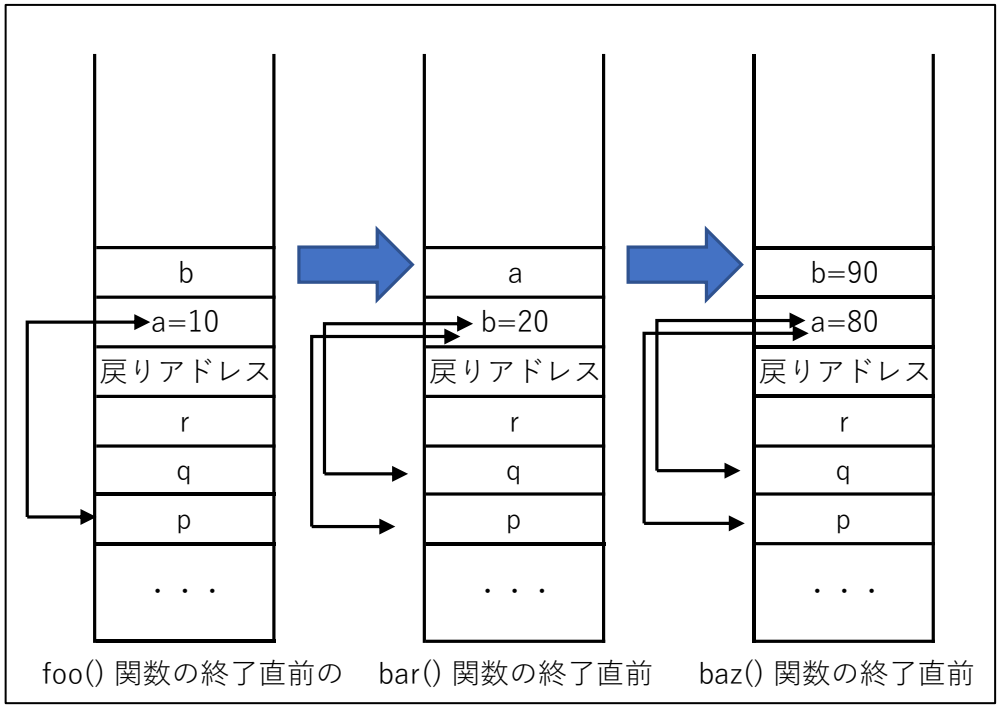

図 6　foo 関数の呼び出し時・終了時のスタックの様子

課題 2-3

課題 2-2 で説明したように（図 6 を参照），ある関数を呼び出すと，関数の引数の値と関数実行後に実行する機械語のアドレスがスタックに格納される．しかし関数が終了すると呼び出すときに生成されたスタックフレームが開放される．したがって，関数の局所変数もスタック上に存在しなくなる．

```
#include <stdio.h>
void swap(int a, int b){
	int temp;		/*  作業用  */
	temp = a;		/* a の値を一時保存  */
	a = b;		/* a の値を b の値に書き換える  */
	b = temp;		/* b の値を a の値に書き換える  */
}
int main(void){
	int	x = 10, y = 20;		/* int 型の変数の宣言  */
	swap(x, y);
	printf("交換後: x = %d	y = %d¥n", a, b);
	return 0;
}
```

上記のプログラムは局所変数を用いた swap 関数で入れ替えを行うプログラムである． swap 関数が呼び出された時のスタックの様子を図 7 に示す.

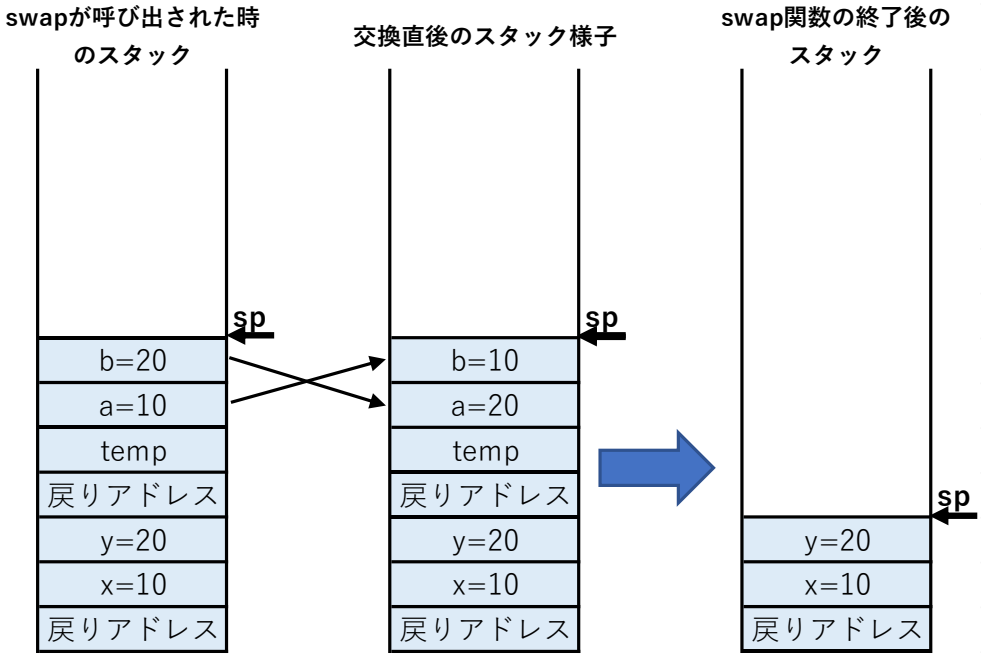

図 7　swap(int a, int b)関数が呼び出された時・終了の直後のスタックの様子

swap 関数が呼び出された時 main 関数の x と y が swap 関数の a と b にコピーされる． そして swap 関数内において，swap 関数の a と b 同士の交換が行われる．しかし swap 関数の終了によってスタックフレームが開放され，交換していない変数同士 x,y が出力される．つまり交換は swap(int a, int b) のスタックフレーム内の操作のみで完結してしまい，引数で渡した変数の値の交換は行われない.

しかし，swap(int *a, int *b)では，swap 関数に変数の値をではなく変数のアドレスを渡しているため，変数本体を直接操作することができる．そのため，ポインタ操作を行うことで，x と y の値を入れ替えることができる．その処理を行う時のスタック様子の変化を図 8 に示す.

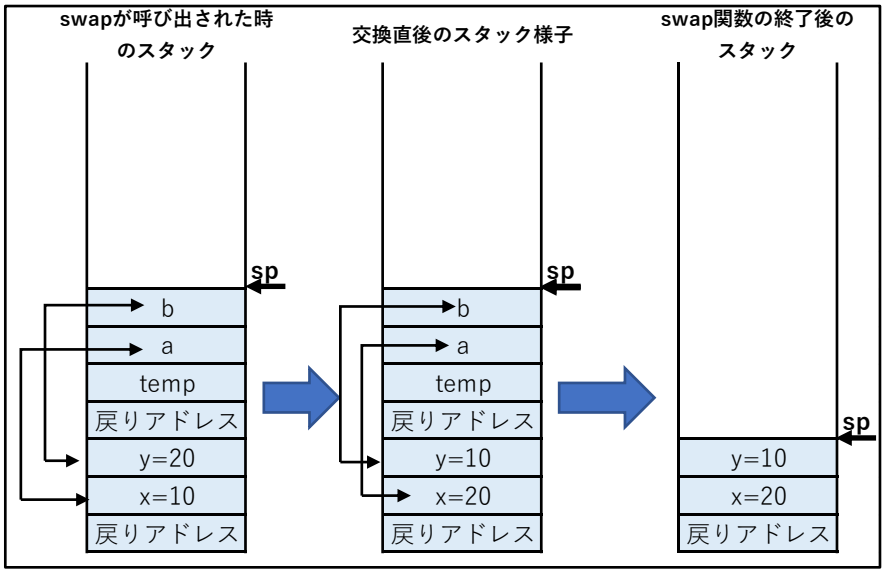

*図 8　swap(int *a, int *b) 関数が呼び出された時・終了の直後のスタックの様子*

3. 課題 3

課題 3-1

課題 1 の assignment1.c に対して gcc -S を実行し comb() に当たるアセンブラコードを得た. 以下に示す.

comb() のアセンブラコード

```
1 comb:
2 .LFB0:
3         .cfi_startproc
4         pushq   %rbp
5         .cfi_def_cfa_offset 16
6         .cfi_offset 6, -16
7         movq    %rsp, %rbp
8         .cfi_def_cfa_register 6
9         pushq   %rbx
10        subq    $40, %rsp
11        .cfi_offset 3, -24
12        movl    %edi, -36(%rbp)
13        movl    %esi, -40(%rbp)
14        cmpl    $0, -36(%rbp)
15        jne     .L2
16        movl    $1, -20(%rbp)
17        jmp     .L3
18 .L2:
19        cmpl    $1, -40(%rbp)
20        jne     .L4
21        movl    -36(%rbp), %eax
22        movl    %eax, -20(%rbp)
23        jmp     .L3
24 .L4:
25        cmpl    $0, -40(%rbp)
26        je      .L5
27        movl    -36(%rbp), %eax
28        cmpl    -40(%rbp), %eax
29        jne     .L6
30 .L5:
31        movl    $1, -20(%rbp)
32        jmp     .L3
33 .L6:
34        movl    -40(%rbp), %eax
35        leal    -1(%rax), %edx
36        movl    -36(%rbp), %eax
37        subl    $1, %eax
38        movl    %edx, %esi
39        movl    %eax, %edi
40        call    comb
41        movl    %eax, %ebx
42        movl    -36(%rbp), %eax
43        leal    -1(%rax), %edx
44        movl    -40(%rbp), %eax
45        movl    %eax, %esi
46        movl    %edx, %edi
47        call    comb
48        addl    %ebx, %eax
49        movl    %eax, -20(%rbp)
50 .L3:
51        movl    -20(%rbp), %eax
52        addq    $40, %rsp
53        popq    %rbx
54        popq    %rbp
55        .cfi_def_cfa 7, 8
56        ret
57        .cfi_endproc
```

課題 3-2

comb() を基本ブロックに区切り，それぞれについて説明する．3-1 で得られたアセンブラは以下の 7 つの基本ブロックに区切ることができる．

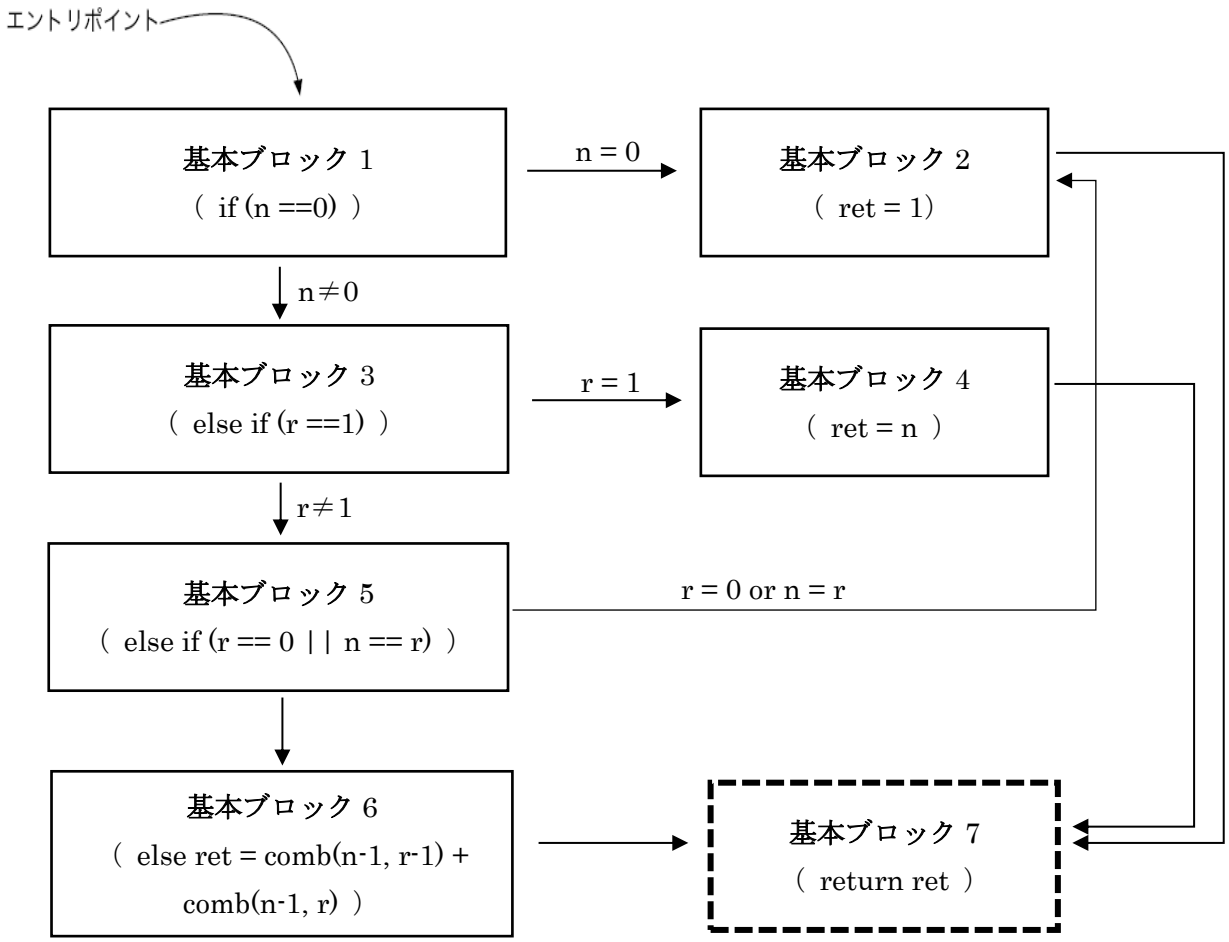

図 9　基本ブロックにもとづいた動作の流れ

・基本ブロック 1: 1 行目から 15 行目．if(n==0) に相当．
・基本ブロック 2: 16 行目から 17 行目と 30 行目から 33 行目．ret = 1 に相当．
・基本ブロック 3: 18 行目から 20 行目．else if(r==1) に相当．
・基本ブロック 4: 21 行目から 23 行目．ret = n に相当．
・基本ブロック 5: 24 行目から 29 行目．else if (r == 0 || n == r) に相当．
・基本ブロック 6: 33 行目から 49 行目．else ret = comb(n-1, r-1) + comb(n-1, r) に相当．
・基本ブロック 7: 50 行目から 57 行目．return ret に相当．

基本ブロックに基づいた動作フローを図 9 に示す．基本ブロック 1 がエントリポイントであり，各ブロックへと遷移していく．基本ブロック 7 につくと終了する．以降，基本ブロックごとにそれぞれの挙動を説明していく．

基本ブロック 1

```
1 comb:
2 .LFB0:
3         .cfi_startproc
4         pushq   %rbp
5         .cfi_def_cfa_offset 16
6         .cfi_offset 6, -16
7         movq    %rsp, %rbp
8         .cfi_def_cfa_register 6
9         pushq   %rbx
10        subq    $40, %rsp
11        .cfi_offset 3, -24
12        movl    %edi, -36(%rbp)
13        movl    %esi, -40(%rbp)
14        cmpl    $0, -36(%rbp)
15        jne     .L2
```

comb() を call すると最初に実行される基本ブロックである．本ブロックは comb() の if (n == 0) に相当する処理である．スタックにベースポインタを保存，現在のベースポインタ値をセットしてスタックポインタをずらす(3～10 行目). %rbp － 36 番地の値(引数 n) と比較し，0 でなければラベル.L2(基本ブロック 3) へ，そうであれば基本ブロック 2 へ飛ぶ.

基本ブロック 2

```
16        movl    $1, -20(%rbp)
17        jmp     .L3

30 .L5:
31        movl    $1, -20(%rbp)
32        jmp     .L3
```

本ブロックは comb( ) の if 文中の ret = 1 に相当する処理である．%rbp － 20 番地には ret 変数があり，そこに 0 を代入し，ラベル.L3(基本ブロック 7) に飛ぶ．16～17 行目のコードと 31～32 行目のコードが同じコードである.

基本ブロック 3

```
18 .L2:
19        cmpl    $1, -40(%rbp)
20        jne     .L4
```

if 文中の else if (r == 1) に相当する処理である．%rbp－40 番地の値(引数 r) の値が 1 と等しいか比較し，そうでなければラベル.L4(基本ブロック 5) へ，そうであれば基本ブロック 4 へ飛ぶ.

基本ブロック 4

```
21        movl    -36(%rbp), %eax
22        movl    %eax, -20(%rbp)
23        jmp     .L3
```

本ブロックは comb( ) の if 文中の ret = n に相当する処理である．%rbp － 36 番地にある n 変数を， %rbp － 20 番地にある ret 変数に代入し，ラベル.L3(基本ブロック 7) に飛ぶ.

基本ブロック 5

```
24 .L4:
25      cmpl    $0, -40(%rbp)
26      je      .L5
27      movl    -36(%rbp), %eax
28      cmpl    -40(%rbp), %eax
29      jne     .L6
```

if 文中の else if (r == 0 || n == r) に相当する処理である．%rbp－40 番地の値(引数 r) の値が 0 と等しいか比較し，そうであれば基本ブロック 2（30 行目（L5)） へ飛ぶ．そうでなければさらに%rbp－36 番地の n と比較する．n と等しくなければラベル.L6(基本ブロック 6) へ，等しければ基本ブロック 2（30 行目（L5)） へ飛ぶ．

基本ブロック 6

```
33 .L6:
34      movl    -40(%rbp), %eax
35      leal    -1(%rax), %edx
36      movl    -36(%rbp), %eax
37      subl    $1, %eax
38      movl    %edx, %esi
39      movl    %eax, %edi
40      call    comb
41      movl    %eax, %ebx
42      movl    -36(%rbp), %eax
43      leal    -1(%rax), %edx
44      movl    -40(%rbp), %eax
45      movl    %eax, %esi
46      movl    %edx, %edi
47      call    comb
48      addl    %ebx, %eax
49      movl    %eax, -20(%rbp)
```

再帰呼び出し行う，ret = comb(n-1, r-1) + comb(n-1, r) に相当する処理を行っている．eax に%rbp－36 番地（引数 n）の値を代入し 1 をマイナスする(n－1 の実行)．同じ処理を%rbp－40 番地（引数 r）についても行い，edx に代入する．その後 edi に eax の値を，esi に edx の値を代入し(引数として渡し)，comb() を call し(comb(n-1,r-1))，その返り値を ebx に保存している．42 から 47 行目にかけては，comb(n-1, r)を呼び出すために同様の処理を行っている．その後，comb(n-1,r)の返り値は eax に格納されているため，ebx の値を足し込み，%rbp－20 番地(ret) に演算結果を代入する．

基本ブロック 7

```
50 .L3:
51      movl    -20(%rbp), %eax
52      addq    $40, %rsp
53      popq    %rbx
54      popq    %rbp
55      .cfi_def_cfa 7, 8
56      ret
57      .cfi_endproc
```

最後の return ret に相当する処理を行う．ret が返り値なので，%rbp－20 の値を eax に格納する．その後，スタックポインタを 40 ずらし(呼び出し元の rbx が保存されている番地にセット)，スタック上に退避しておいた rbx と rbp の値を復元して呼び出し元へ戻る．

課題 3-3

スタックの動作は課題 1-3 で説明したとおりである．関数が呼ばれるたびにスタックフレームが積まれていく．スタックフレームの具体的な内容を図 10 に示す．図 10 では，comb(2,2) を呼び出したときのスタックフレームを表示している．呼び出し元の comb()のアドレスが戻りアドレスとして記述されている．rbp と rbx の値が格納されており，引数 n,r と局所変数 ret はそれぞれ%rbp－36 番地，$rbp－40 番地，%rbp－20 番地に割り当てられている．

スタックポインタは rbx の入っているアドレスから 40 番地分マイナスしたアドレスを指している．comb (2,2) を実行すると，ret に 1 が代入される．各関数呼び出しで積まれるスタックフレームは同じ構造をとっており，n,r と ret に格納される値が異なっていく．

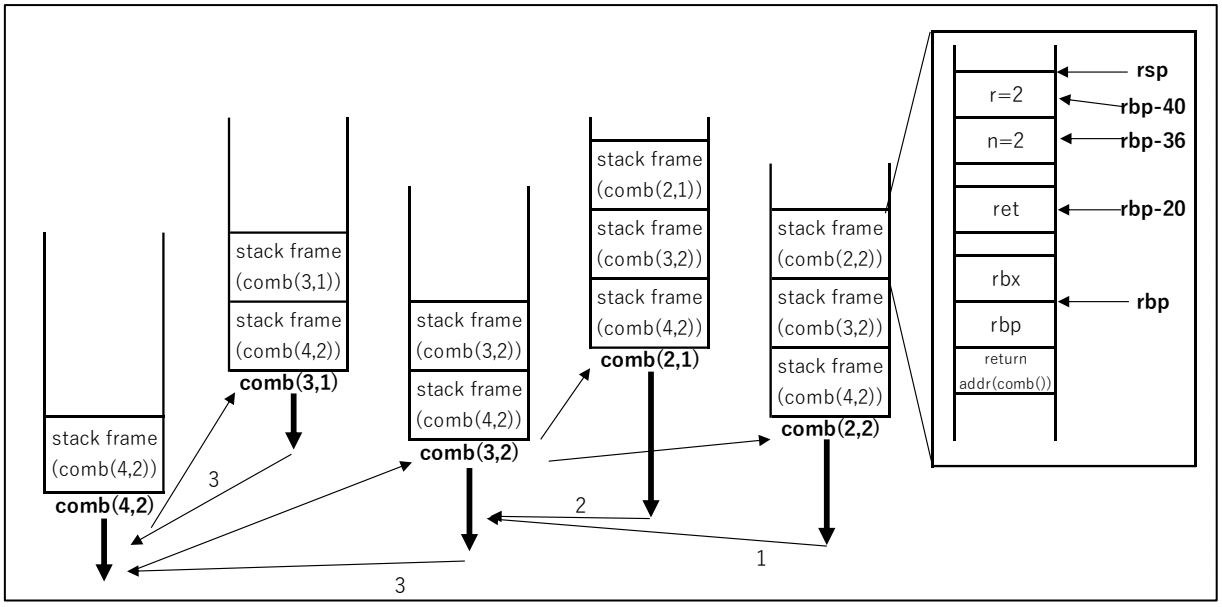

図 10　comb(4,2)を実行した時のスタックの様子とスタックフレーム中身

課題 3-4

RFLAGS について Jun Mizutani がこう述べている．

[“*フラグレジスタ RFLAGS：演算や比較の結果が正負，同じか否かなどで変化するビットが集まっている．上位の 42 ビットは使用しない（予約領域）．条件分岐（Jcc），条件転送（CMOVcc）で使う．*

byte 7	byte 6	byte 5	byte 4	byte 3	byte 2	byte 1	byte 0
upper 32bit				7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
reserved (0)				reserved (0)	ID VIP VIF AC VM RF 0	NT IOPL OF DF IF TF SF ZF 0 AF 0 PF 1 CF	

図 11　RFLAGS（<http://www.mztn.org/lxasm64/amd04.html>より，> 2018 年 11 月 20 日）

フラグレジスタの各ビットの働きを示す．算術ステータスフラグ(CF, PF, AF, ZF, SF, OF)以外の詳細は省略する．

ID：
Identification フラグ
VIP：
バーチャル割り込み保留
VIF：
バーチャル割り込み
AC：
アラインメントチェック
VM：
仮想 86 モード
RF：
Resume フラグ
NT：
Nested Task
IOPL：
I/O 特権レベルフィールド
OF：
オーバーフローフラグ． 算術演算の結果が 2 の補数で表せない大きさの場合にセットされる．
DF
方向フラグ．ストリング命令（MOVS, CMPS, SCAS, LODS, STOS)の自動インクリメント(set)とデクリメント(clear)を指定して文字列操作の方向を決める．
IF：
割り込み許可フラグ
TF：
トラップフラグ．デバッグ用のシングルステップモードに移行する．
SF：
符号フラグ．演算結果の最上位ビットと同じにセットされる．2 の補数の正負を表す．
ZF：
ゼロフラグ．演算結果がゼロのときにセットされる．
AF：
ビット 3 からのキャリーやボローが発生した場合にセットされる．
PF：
パリティフラグ．演算結果の下位 8 ビットのパリティを示す．下位 8 ビットの 1 の数が偶数なら 1，奇数なら 0 になる．
CF：
キャリーフラグ．

このビットは算術演算，シフト，ローテイト命令で最上位ビットまたは最下位ビットから桁あふれが発生した場合にセットされる．”］

上記の引用より，RFLAGS には様々なフラグが存在することが分かる．キャリーフラグ(CF) やパリティフラグ(PF)，ゼロフラグ(ZF)，オーバーフローフラグ(OF) などが演算に関わるフラグとして挙げられる．

4. 課題 4

課題 4-1

プログラムの実行結果を図 12 に示す．プログラムは無限なループになっているように動き，終了しなかった．その原因について 4－2 において説明する．

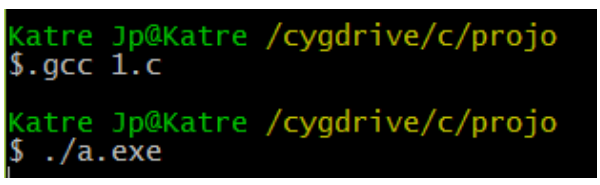

図 12　課題 4-1　プログラムの実行結果

課題 4-2

プログラムの挙動をもっと具体的に見るには, for 文の中に i を出力させる printf を追加する. よって，課題 4-1 のプログラムは以下の通りとなる.

```
#include<stdio.h>
int main(){
        int a[4], i;
        for (i = 0; i < 8; i++){
                a[i] = 0;
                printf("i=%d¥n", i);
        }
        return 0;
```

出力結果を以下に示す.

```
$ gcc assignment4.c
$ ./a.out
i=0
i=1
i=2
i=3
i=4
i=5
i=6
i=0
i=1
i=2
i=3
i=4
i=5
i=6
i=0
i=1
・
・
・
```

出力より，変数 i の値が定期的に 0 になってしまうことがわかる．よって，プログラムは for 文の中から抜けなくなり，無限ループになってしまう.

プログラムで定義した a[4]配列の範囲を超えられている．よって，範囲を超えた領域への代入によるスタック破壊が原因で 4-1 のような結果になったと考えられる.

スタック破壊が原因だと考えられるが，それを確かめるにはプログラムのアセンブラコードとスタックの様子を確認する．プログラムの該当アセンブラコードを次に示す.

```
1 .L3:
2       movl -4(%rbp), %eax
3       cltq
4       movl $0, -32(%rbp,%rax,4)
5       addl $1, -4(%rbp)
6 .L2:
7       cmpl $7, -4(%rbp)
8       jle .L3
```

左にあるアセンブラコードの 4 行目より 0 がスタックの%rbp-20 から%rbp-32 までの範囲に格納されることがわかる．よって，その範囲は a[4]配列が格納されている範囲だと判断できる．そして，i が%rbp-4 に格納される．(5 行目からわかる，5 行目は i=1 のアセンブラコードである．

変数の格納されるスタック上のアドレスを上記のコードより求めた．それを用いてスタックの様子とその変化を図 13 に図示する．i=7 の時に，a[7]=0 が実行されるが，a[7]がスタック上で指しているアドレスと i が指しているアドレスが同じアドレス（rbp-4）であるため，i に 0 が代入されてしまう．それによって，i が定期的に 0 になってしまい，for 文を抜ける条件である i < 8 を満たすことがないため，プログラムは無限ループになってしまう．

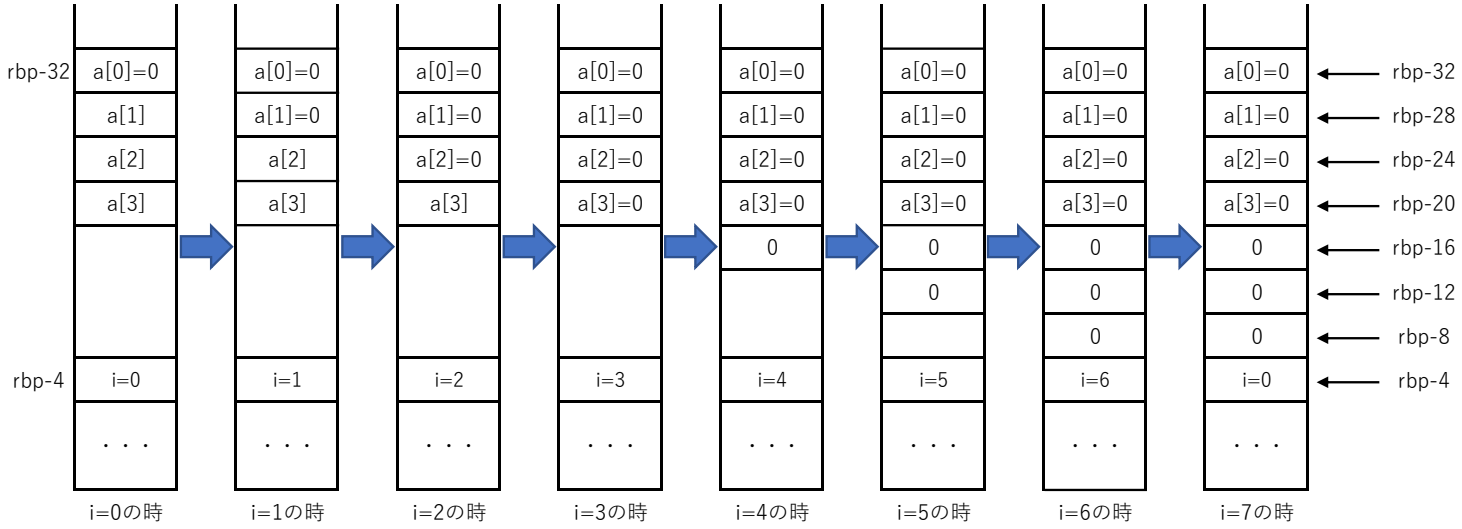

図 13　課題 4-1 のプログラムのスタックの様子

課題 4-3(発展課題)

下記は IBM developerWorks の「バッファー・オーバーフローに対抗する新しい細工」という記事から引用し，参考にしたものである．

「研究者である Crispen Cowan は StackGuard と呼ばれる面白い手法を作り出した．StackGuard は C コンパイラ（gcc）を変更し，戻りアドレスの前に「カナリア値（canary value)」と呼ばれる値を挿入する．この「カナリア」は炭坑でのカナリアと同じような役割を果たし，何かがおかしくなった時には警告する．カナリア変数を図 14 に示す．

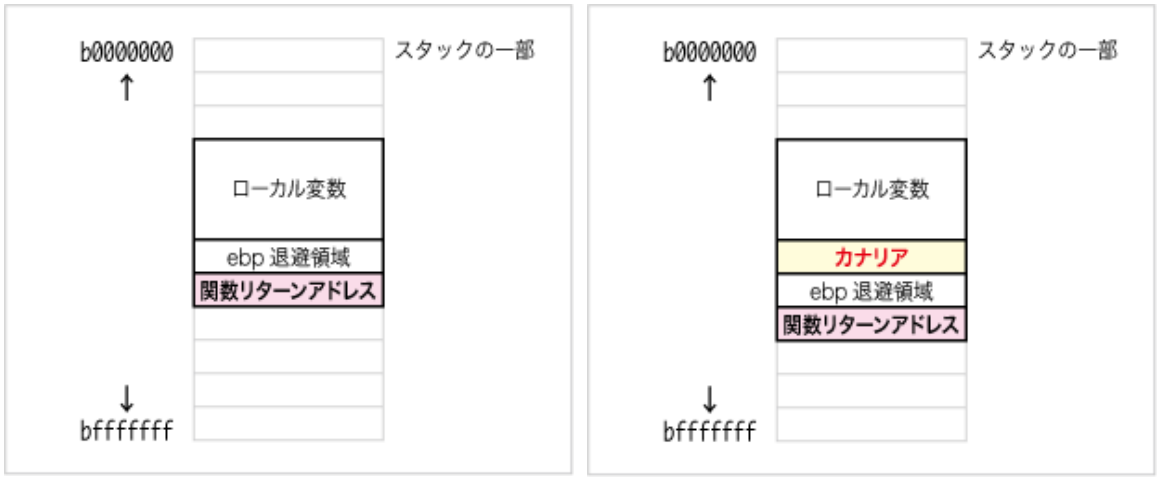

図 14　カナリア変数を追加する前（左）そして後（右）のスタックの様子
（<https://www.ipa.go.jp/security/awareness/vendor/programmingv2/contents/c905.html> より引用，2018/11/20 閲覧）

StackGuard はファンクションが戻る前に，カナリア値が変更されていないかをチェックする．攻撃者が（スタック破壊攻撃の一部として）戻りアドレスを上書きすると，おそらくカナリアの値が変わるので，システムが停止する．これは有効な手法であるが，バッファー・オーバーフローが他の値を上書きするのを防ぐ事はできないという点には注意すべきことはきである（上書きされたその値は，システムへの攻撃のためにまだ使えるかも知れないのである）．この手法を拡張して他の値も保護できるようにする作業も行われている．　」

5. 参考文献

[1] Debugging Segmentation Faults and Pointer Problems ，https://www.cprogramming.com/debugging/segfaults.html，閲覧日：2018 年 11 月 17 日

[2] RFLAGS（< http://www.mztn.org/lxasm64/amd04.html>より 2018 年 11 月 20 日）

[3] https://www.ibm.com/developerworks/jp/linux/library/l-sp4/index.html,閲覧日：2018 年 11 月 20 日

[4]山田浩史，東京農工大学・情報工学科・情報工学実験 1 テキスト 2018，第 6 回実験

[5]山田浩史，東京農工大学・情報工学科・2017 学科掲示板
< https://board.cs.tuat.ac.jp/2017/boards/kyomu/body/00158.html>