Computer Systems Fundamentals

call_return.c

C Function with No Parameters or Return Value

```
#include <stdio.h>

void f(void);

int main(void) {
    printf("calling function f\n");
    f();
    printf("back from function f\n");
    return 0;
}

void f(void) {
    printf("in function f\n");
}
```

call_return.broken.s

simple example of returning from a function loops because main does not save return address

```
main:
    la   $a0, string0   # printf("calling function f\n");
    li   $v0, 4
    syscall

    jal f               # set $ra to following address

    la   $a0, string1   # printf("back from function f\n");
    li   $v0, 4
    syscall

    li   $v0, 0         # fails because $ra changes since main called
    jr   $ra            # return from function main


f:
    la $a0, string2     # printf("in function f\n");
    li $v0, 4
    syscall
    jr $ra              # return from function f


    .data
string0:
    .asciiz "calling function f\n"
string1:
    .asciiz "back from function f\n"
string2:
    .asciiz "in function f\n"
```

call_return.s

simple example of placing return address on stack note stack grows down

```
main:
    sub   $sp, $sp, 4     # move stack pointer down to make room
    sw    $ra, 0($sp)     # save $ra on $stack

    la    $a0, string0    # printf("calling function f\n");
    li    $v0, 4
    syscall

    jal   f               # set $ra to following address

    la    $a0, string1    # printf("back from function f\n");
    li    $v0, 4
    syscall

    lw    $ra, 0($sp)     # recover $ra from $stack
    add   $sp, $sp, 4     # move stack pointer back to what it was

    li    $v0, 0          # return 0 from function main
    jr    $ra             #


f:
    la $a0, string2       # printf("in function f\n");
    li $v0, 4
    syscall
    jr $ra                # return from function f


    .data
string0:
    .asciiz "calling function f\n"
string1:
    .asciiz "back from function f\n"
string2:
    .asciiz "in function f\n"
```

return_answer.c

simple example of returning a value from a function

```
#include <stdio.h>

int answer(void);

int main(void) {
    int a = answer();
    printf("%d\n", a);
    return 0;
}

int answer(void) {
    return 42;
}
```

return_answer.s

simple example of returning a value from a function note storing of return address $ra and $a0 on stack for simplicity we are not using a frame pointer

```
main:
    sub  $sp, $sp, 4     # move stack pointer down to make room
    sw   $ra, 0($sp)     # save $ra on $stack

    jal  answer          # call answer, return value will be in $v0

    move $a0, $v0        # printf("%d", a);
    li   $v0, 1
    syscall

    li   $a0, '\n'       # printf("%c", '\n');
    li   $v0, 11
    syscall



    lw   $ra, 0($sp)     # recover $ra from $stack
    add  $sp, $sp, 4     # move stack pointer back up to what it was when main called

    li   $v0, 0          # return 0 from function main
    jr   $ra             #

answer:
    li $v0, 42           #
    jr $ra               # return from answer
```

more_calls.c

example of function calls

```
#include <stdio.h>

int sum_product(int a, int b);
int product(int x, int y);

int main(void) {
    int z = sum_product(10, 12);
    printf("%d\n", z);
    return 0;
}

int sum_product(int a, int b) {
    int p = product(6, 7);
    return p + a + b;
}

int product(int x, int y) {
    return x * y;
}
```

more_calls.s

example of function calls note storing of return address $a0, $a1 and $ra on stack for simplicity we are not using a frame pointer

```
main:
    sub  $sp, $sp, 4     # move stack pointer down to make room
    sw   $ra, 0($sp)     # save $ra on $stack

    li   $a0, 10          # sum_product(10, 12);
    li   $a1, 12
    jal  sum_product

    move $a0, $v0        # printf("%d", z);
    li   $v0, 1
    syscall

    li   $a0, '\n'        # printf("%c", '\n');
    li   $v0, 11
    syscall

    lw   $ra, 0($sp)     # recover $ra from $stack
    add  $sp, $sp, 4     # move stack pointer back up to what it was when main called

    li   $v0, 0          # return 0 from function main
    jr   $ra             # return from function main



sum_product:
    sub  $sp, $sp, 12    # move stack pointer down to make room
    sw   $ra, 8($sp)     # save $ra on $stack
    sw   $a1, 4($sp)     # save $a1 on $stack
    sw   $a0, 0($sp)     # save $a0 on $stack

    li   $a0, 6          # product(6, 7);
    li   $a1, 7
    jal  product

    lw   $a1, 4($sp)     # restore $a1 from $stack
    lw   $a0, 0($sp)     # restore $a0 from $stack

    add  $v0, $v0, $a0   # add a and b to value returned in $v0
    add  $v0, $v0, $a1   # and put result in $v0 to be returned

    lw   $ra, 8($sp)     # restore $ra from $stack
    add  $sp, $sp, 12    # move stack pointer back up to what it was when main called

    jr   $ra             # return from sum_product


product:                 # product doesn't call other functions
                         # so it doesn't need to save any registers
    mul  $v0, $a0, $a1   # return argument * argument 2
    jr   $ra              #
```

two_powerful.c

recursive function which prints first 20 powers of two in reverse

```
#include <stdio.h>

void two(int i);

int main(void) {
    two(1);
}

void two(int i) {
    if (i < 1000000) {
        two(2 * i);
    }
    printf("%d\n", i);
}
```

two_powerful.s

simple example of placing return address $ra and $a0 on stack for simplicity we are not using a frame pointer

```
main:
    sub $sp, $sp, 4     # move stack pointer down to make room
    sw $ra, 0($sp)      # save $ra on $stack

    li $a0, 1           # two(1);
    jal two


    lw $ra, 0($sp)      # recover $ra from $stack
    add $sp, $sp, 4     # move stack pointer back up to what it was when main called

    jr $ra              # return from function main



two:
    sub $sp, $sp, 8     # move stack pointer down to make room
    sw $ra, 4($sp)      # save $ra on $stack
    sw $a0, 0($sp)      # save $a0 on $stack

    bge $a0, 1000000, print
    mul $a0, $a0, 2     # restore $a0 from $stack
    jal two
print:

    lw $a0, 0($sp)      # restore $a0 from $stack
    li $v0, 1           # printf("%d");
    syscall

    li $a0, '\n'        # printf("%c", '\n');
    li $v0, 11
    syscall

    lw $ra, 4($sp)      # restore $ra from $stack
    add $sp, $sp, 8     # move stack pointer back up to what it was when main called

    jr $ra              # return from two
```

squares.c

store first 10 squares into an array which is a local variable then print them from array

```c
#include <stdio.h>

int main(void) {
    int squares[10];
    int i = 0;
    while (i < 10) {
        squares[i] = i * i;
        i++;
    }
    i = 0;
    while (i < 10) {
        printf("%d", squares[i]);
        printf("%c",'\n');
        i++;
    }
    return 0;
}
```

squares.s

i in register $t0 registers $t1, and $t2, used to hold temporary results

```
main:
    sub  $sp, $sp, 40    # move stack pointer down to make room
                         # to store array numbers on stack
    li   $t0, 0          # i = 0

loop0:
    bge  $t0, 10, end0   # while (i < 10) {

    mul  $t1, $t0, 4     #     calculate &numbers[i]
    add  $t2, $t1, $sp   #
    mul  $t3, $t0, $t0   #     calculate i * i
    sw   $t3, ($t2)      #     store in array

    add  $t0, $t0, 1     #     i++;
    b    loop0           # }
end0:

    li   $t0, 0          # i = 0
loop1:
    bge  $t0, 10, end1   # while (i < 10) {

    mul  $t1, $t0, 4
    add  $t2, $t1, $sp   #    calculate &numbers[i]
    lw   $a0, ($t2)      #    load numbers[i] into $a0
    li   $v0, 1          #    printf("%d", numbers[i])
    syscall

    li   $a0, '\n'        #   printf("%c", '\n');
    li   $v0, 11
    syscall

    add  $t0, $t0, 1     #    i++
    b    loop1           # }
end1:

    add  $sp, $sp, 40    # move stack pointer back up to what it was when main called
    li   $v0, 0          # return 0 from function main
    jr   $ra             #
```

frame_pointer.c

example of function where frame pointer useful because stack grows during function execution

```c
#include <stdio.h>

void f(int a) {
    int length;
    scanf("%d", &length);
    int array[length];
    // ... more code ...
    printf("%d\n", a);
}
```

frame_pointer.broken.s

example stack growing during function execution breaking the function return

```
f:
    sub  $sp, $sp, 8       # move stack pointer down to make room
    sw   $ra, 4($sp)       # save $ra on $stack
    sw   $a0, 0($sp)       # save $a0 on $stack

    li   $v0, 5            # scanf("%d", &length);
    syscall

    mul  $v0, $v0, 4       # calculate array size
    sub  $sp, $sp, $v0     # move stack_pointer down to hold array

    # ...

                          # breaks because stack pointer moved down to hold array
                          # so we won't restore the correct value
    lw   $ra, 4($sp)      # restore $ra from $stack
    add  $sp, $sp, 8      # move stack pointer back up to what it was when main called

    jr   $ra              # return from f
```

frame_pointer.s

using a frame pointer to handle stack growing during function execution

```
f:
    sub  $sp, $sp, 12     # move stack pointer down to make room
    sw   $fp, 8($sp)      # save $fp on $stack
    sw   $ra, 4($sp)      # save $ra on $stack
    sw   $a0, 0($sp)      # save $a0 on $stack
    add  $fp, $sp, 12     # have frame pointer at start of stack frame

    li   $v0, 5           # scanf("%d", &length);
    syscall

    mul  $v0, $v0, 4      # calculate array size
    sub  $sp, $sp, $v0    # move stack_pointer down to hold array

    # ... more code ...

    lw   $ra, -8($fp)     # restore $ra from stack
    move $sp, $fp         # move stack pointer backup  to what it was when main called
    lw   $fp, -4($fp)     # restore $fp from $stack
    jr   $ra              # return
```

pointer.c

demonstrate implementaion of pointers by an address

```c
#include <stdio.h>

int answer = 42;

int main(void) {
    int i;
    int *p;

    p = &answer;
    i = *p;
    printf("%d\n", i); // prints 42
    *p = 27;
    printf("%d\n", answer); // prints 27

    return 0;
}
```

pointer.s

demonstrate implementation of pointers by an address p in register $t0 i in register $t1 $t2 used for temporary value

```
main:
    la   $t0, answer     # p = &answer;

    lw   $t1, ($t0)      # i = *p;

    move $a0, $t1        # printf("%d\n", i);
    li   $v0, 1
    syscall

    li   $a0, '\n'       # printf("%c", '\n');
    li   $v0, 11
    syscall

    li   $t2, 27         # *p = 27;
    sw   $t2, ($t0)      #

    lw   $a0, answer     # printf("%d\n", answer);
    li   $v0, 1
    syscall

    li   $a0, '\n'       # printf("%c", '\n');
    li   $v0, 11
    syscall

    li   $v0, 0          # return 0 from function main
    jr   $ra             #

    .data
answer:
    .word 42             # int answer = 42;
```

strlen_array.c

calculate the length of a string using a strlen like function

```c
#include <stdio.h>

int my_strlen(char *s);

int main(void) {
    int i = my_strlen("Hello Andrew");
    printf("%d\n", i);
    return 0;
}

int my_strlen(char *s) {
    int length = 0;
    while (s[length] != 0) {
        length++;
    }
    return length;
}
```

strlen_array.goto.c

calculate the length of a string using a strlen like function

```c
#include <stdio.h>

int my_strlen(char *s);

int main(void) {
    int i = my_strlen("Hello Andrew");
    printf("%d\n", i);
    return 0;
}

int my_strlen(char *s) {
    int length = 0;
loop:
    if (s[length] == 0) goto end;
        length++;
    goto loop;
end:
    return length;
}
```

strlen_array.s

calculate the length of a string using a strlen like function

```
main:
    sub  $sp, $sp, 4      # move stack pointer down to make room
    sw   $ra, 0($sp)      # save $ra on $stack

    la   $a0, string      # my_strlen("Hello Andrew");
    jal  my_strlen

    move $a0, $v0         # printf("%d", i);
    li   $v0, 1
    syscall

    li   $a0, '\n'         # printf("%c", '\n');
    li   $v0, 11
    syscall

    lw   $ra, 0($sp)      # recover $ra from $stack
    add  $sp, $sp, 4      # move stack pointer back up to what it was when main called

    li   $v0, 0           # return 0 from function main
    jr   $ra              #


my_strlen:                # length in t0, s in $a0
    li   $t0, 0
loop:                     # while (s[length] != 0) {
    add  $t1, $a0, $t0    #   calculate &s[length]
    lb   $t2, 0($t1)      #   load s[length] into $t2
    beq  $t2, 0, end      #
    add  $t0, $t0, 1      #   length++;
    b    loop             # }
end:
    move $v0, $t0         # return length
    jr   $ra

    .data
string:
    .asciiz "Hello Andrew"
```

strlen_pointer.c

```
#include <stdio.h>

int my_strlen(char *s);

int main(void) {
    int i = my_strlen("Hello Andrew");
    printf("%d\n", i);
    return 0;
}

int my_strlen(char *s) {
    int length = 0;
    while (*s != 0) {
        length++;
        s++;
    }
    return length;
}
```

strlen_pointer.s

simple example of placing return address $ra and $a0 on stack for simplicity we are not using a frame pointer

```
main:
    sub $sp, $sp, 4       # move stack pointer down to make room
    sw $ra, 0($sp)        # save $ra on $stack

    la $a0, string        # my_strlen("Hello Andrew");
    jal my_strlen

    move $a0, $v0         # printf("%d", i);
    li $v0, 1
    syscall

    li $a0, '\n'          # printf("%c", '\n');
    li $v0, 11
    syscall

    lw $ra, 0($sp)        # recover $ra from $stack
    add $sp, $sp, 4       # move stack pointer back up to what it was when main called

    jr $ra                # return from function main

my_strlen:                # length in t0, s in $a0
    li $t0, 0
loop:                     #
    lb $t1, 0($a0)        # load *s into $t1
    beq $t1, 0, end       #
    add $t0, $t0, 1       # length++
    add $a0, $a0, 1       # s++
    b loop                #
end:
    move $v0, $t0         # return length
    jr $ra

    .data
string:
    .asciiz "Hello Andrew"
```

stack_inspect.c

```c
#include <stdio.h>
#include <stdint.h>

/*
$ clang stack_inspect.c
$ a.out
 0: Address 0x7ffe1766c304 contains 3            <- a[0]
 1: Address 0x7ffe1766c308 contains 5            <- x
 2: Address 0x7ffe1766c30c contains 2a           <- b
 3: Address 0x7ffe1766c310 contains 1766c330     <- f  frame pointer (64 bit)
 4: Address 0x7ffe1766c314 contains 7ffe
 5: Address 0x7ffe1766c318 contains 40120c       <- f return address
 6: Address 0x7ffe1766c31c contains 0
 7: Address 0x7ffe1766c320 contains 22
 8: Address 0x7ffe1766c324 contains 25
 9: Address 0x7ffe1766c328 contains 9            <- a
10: Address 0x7ffe1766c32c contains 0
11: Address 0x7ffe1766c330 contains 401220       <- main return address
12: Address 0x7ffe1766c334 contains 0
13: Address 0x7ffe1766c338 contains c7aca09b     <- main frame pointer (64 bit)
14: Address 0x7ffe1766c33c contains 7ff3
15: Address 0x7ffe1766c340 contains 0
*/

void f(int b) {
    int x = 5;
    uint32_t a[1] = { 3 };

    for (int i = 0; i < 16; i++)
        printf("%2d: Address %p contains %x\n", i, &a[i], a[0 + i]);
}

int main(void) {
    int a = 9;
    printf("function main is at address %p\n", &main);
    printf("function f is at address %p\n", &f);
    f(42);
    return 0;
}
```

invalid0.c

```
Run at CSE like this


$ clang invalid0.c -o invalid0
$ ./invalid0
42 77 77 77 77 77 77 77 77 77
```

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int a[10];
    int b[10];
    printf("a[0] is at address %p\n", &a[0]);
    printf("a[9] is at address %p\n", &a[9]);
    printf("b[0] is at address %p\n", &b[0]);
    printf("b[9] is at address %p\n", &b[9]);

    for (int i = 0; i < 10; i++) {
        a[i] = 77;
    }

    // loop writes to b[10] .. b[12] which don't exist -
    // with gcc 7.3 on x86_64/Linux
    // b[12] is stored where a[0] is stored
    // with gcc 7 on CSE lab machines
    // b[10] is stored where a[0] is stored

    for (int i = 0; i <= 12; i++) {
        b[i] = 42;
    }

    // prints 42 77 77 77 77 77 77 77 77 77 on x86_64/Linux
    // prints 42 42 42 77 77 77 77 77 77 77 at CSE
    for (int i = 0; i < 10; i++) {
        printf("%d ", a[i]);
    }
    printf("\n");

    return 0;
}
```

invalid1.c

```
Run at CSE like this

$ clang invalid1.c -o invalid1
$ ./invalid1
i is at address 0x7ffe2c01cd58
a[0] is at address 0x7ffe2c01cd30
a[9] is at address 0x7ffe2c01cd54
a[10] would be stored at address 0x7ffe2c01cd58

doesn't terminate
```

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int i;
    int a[10];
    printf("i is at address %p\n", &i);
    printf("a[0] is at address %p\n", &a[0]);
    printf("a[9] is at address %p\n", &a[9]);
    printf("a[10] would be stored at address %p\n", &a[10]);

    // loop writes to a[10] .. a[11] which don't exist -
    // but with gcc 7 on x86_64/Linux
    // i would be stored where a[11] is stored

    for (i = 0; i <= 11; i++) {
        a[i] = 0;
    }

    return 0;
}
```

invalid2.c

```
Run at CSE like this

$ clang -Wno-everything  invalid2.c -o invalid2
$ ./invalid2
answer=42
```

```
#include <stdio.h>

void f(int x);

int main(void) {
    int answer = 36;
    printf("answer is stored at address %p\n", &answer);

    f(5);
    printf("answer=%d\n", answer); // prints 42 not 36

    return 0;
}

void f(int x) {
    int a[10];

    // a[18] doesn't exist
    // with clang at CSE variable answer in main
    // happens to be where a[19] would be

    printf("a[18] would be stored at address %p\n", &a[18]);

    a[18] = 42;
}
```

invalid3.c

```
Run at CSE like this

$ clang invalid3.c -o invalid3
$ ./invalid3

I will never be printed.
argc was 1
$
```

```c
#include <stdio.h>
#include <stdlib.h>

void f(void);

int main(int argc, char *argv[]) {
    f();

    if (argc > 0) {
        printf("I will always be printed.\n");
    }

    if (argc <= 0) {
        printf("I will never be printed.\n");
    }

    printf("argc was %d\n", argc);
    return 0;
}

void f() {
    int a[10];

    // function f has it return address on the stack
    // the call of function f from main should return to
    // the next statement which is:  if (argc > 0)
    //
    // with clang at CSE  f's return address is stored where a[12] would be
    //
    // so changing a[12] changes where the function returns
    //
    // adding 12 to a[12] happens to cause it to return several statements later
    // at the printf("I will never be printed.\n");

    a[12] += 12;
}
```

invalid4.c

```
Run at CSE like this

$ clang invalid4.c -o invalid4
$ ./invalid4
authenticated is at address 0xff94bf44
password is at address 0xff94bf3c

Enter your password: 123456789

Welcome. You are authorized.
$
```

```c
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]) {
    int authenticated = 0;
    char password[8];

    printf("authenticated is at address %p\n", &authenticated);
    printf("password[8] would be at address %p\n", &password[8]);

    printf("Enter your password: ");
    int i = 0;
    int ch = getchar();
    while (ch != '\n' && ch != EOF) {
        password[i] = ch;
        ch = getchar();
        i = i + 1;
    }
    password[i] = '\0';

    if (strcmp(password, "secret") == 0) {
        authenticated = 1;
    }

    // a password longer than 8 characters will overflow the array password
    // the variable authenticated is at the address where
    // where password[8] would be and gets overwritten
    //
    // This allows access without knowing the correct password

    if (authenticated) {
        printf("Welcome. You are authorized.\n");
    } else {
        printf("Welcome. You are unauthorized.  Your death will now be implemented.\n");
        printf("Welcome. You will experience a tingling sensation and then death. \n");
        printf("Remain calm while your life is extracted.\n");
    }

    return 0;
}
```