Spring Semester 2021 EE209 Final Exam

Pledging of No Cheating

Note: Please write down your student ID and name, sign on it (draw your signature), and date it. If you do not fill out this page, we won't grade your final exam.

저는 이번 시험을 온라인으로 치르면서 이 과목에서 금지한 어떤 부정행위도 저지르지 않을 것임을 서약합니다. 추후에 위반사항이 발견되었을 경우 합당한 모든 불이익을 감수하겠습니다.

I pledge that I will not participate in any activity of cheating disallowed by this course while taking this exam online. I will assume full responsibility if any violation is found later.

Student ID: ___________________ Name:

Signature: ___________________ Date:

Spring Semester 2021

KAIST EE209

Programming Structures for Electrical Engineering

Final Exam

Wed. June 16, 2021, 13:00 ~ 15:45 (zoom)

Name:

Student ID:

Class: A , B

This exam is closed book and notes. Read the questions carefully and focus your answers on what has been asked. You are allowed to ask the instructor/TAs for help only in understanding the questions, in case you find them not completely clear. Be concise and precise in your answers and state clearly any assumption you may have made. You can submit/upload your answers early but you are allowed to leave the zoom session only after 3:00PM. The submission format can be either MS word file format (.docx) or a PDF file (You can save it in PDF format in MS-Word) You can insert the space in the original exam file if necessary. Good luck!

Question 1		/ 6
Question 2		/ 15
Question 3		/ 35
Question 4		/ 25
Question 5		/ 24
Question 6		/ 10
Question 7		/ 24
Total:		/ 139

1.(6pt) cmp

Consider the following memory layout. Assume 32 bit CPU with 2's complement representation.

Value A:

```
Address      content
2000         1111 1111 (FF)
2001         1111 1111 (FF)
2002         1111 1000 (F8)
2003         1100 1010 (CA)
```

Value B

```
Address      content
2010         1111 1111 (FF)
2011         1111 1111 (FF)
2012         1111 1000 (F8)
2013         1110 1010 (EB)
```

a. (2pt) Assume CPU is Big Endian architecture. Represent the value A and value B in signed integer (2's complement). Write in hexadecimal form.

b. (2pt) Assume CPU is Little Endian architecture. Represent the value A and value B in signed integer (2's complement). Write in hexadecimal form.

c. (2pt) Load value A and value B to register A and register B.

```
cmpl regA,  regB
```

Show the output of ZF, SF, CF and OF for little endian CPU.

2. (15pt) Assemble and Link

We assemble a simple C program.

```
#include <stdio.h>
int main(void) {
   if (getchar() == 'A')
      prinf("Hi\n");
   return 0;
}
```

Please refer to the assembly code below. Note that the developer mis-spelled the `printf()` to `prinf()`.

```
    .section ".rodata"
    msg:
            .asciz  "Hi\n"
            .section ".text"
            .globl  main
    main:
            pushl   %ebp
            movl    %esp, %ebp
            call    getchar
            cmpl    $'A', %eax
            jne     skip
            pushl   $msg
            call    prinf
            addl    $4, %esp
    skip:
            movl    $0, %eax
            movl    %ebp, %esp
            popl    %ebp
        ret
```

a. (5pt) Show the contents of the symbol table after the assembler generates the object file. Fill the blanks of the table below with the proper contents. A binary image consists of a number of sections. They include .bss, .data, .text and etc. Some of the symbols may not be assigned a section by the assembler. For the symbols that are not assigned a section, mark the section name of those symbols as 'X'.

label	section	local / global
	.	

b. (3pt) Consider the symbols obtained in question 2.a. Which of the symbol references need relocation? Specify the line numbers.

c. (3pt) Linker combines the several object files and the libraries, resolving references and generates the final binary image. Which of the symbols in question 2.a need to be resolved by the linker.

d. (4pt) The above C program fails to compile. Assume that the compiler has failed to compile this code because the programmer has mis-spelled the printf() with prinf(). After preprocessing, the compiler goes through four phases in creating the final binary image; Assemble-pass1 (symbol table generation), Assemble-pass2 (code generation), Link-pass1 (symbol resolution) and Link-pass2 (relocation). In which phase of the compilation, does this program fail to compile?

3. (15pt) Call stack setup and parameter setting

We write a function to sum all values in the array. Assume IA32 architecture CPU. The assembly code generated from C code varies widely dependent upon the internal algorithm of the underlying compiler. Modern compiler adopts sophisticated algorithm to make the binary image faster and smaller. In this question, assume that we use very naïve compiler that does not use any optimization techniques. Assume that all local variables are allocated in the associated stack. In evaluating an expression, assume that the temporary values are stored in the six general purpose registers (`EBX, ESI, EDI, EAX, ECX` and `EDX`) and do not occupy the stack space. Assume that when calling a function, the caller always saves EAX, ECX and EDX registers no matter whether they have been used or not. When a function is called, the callee always saves EBX, ESI and EDI registers no matter whether the callee is going to use them or not.

```
/* sum1.c */

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

int sum (int* arr, int n)
{
  if ( n == 1 )
      return arr[n-1] ;
  return arr[n-1] + sum (arr+1, n-1) ;
}

int arr_init(int *arr, int n) {
   for (int i = 0 ; i < n ; ++i)
      arr[i] = rand() % 100;
   return 0 ;
}

int main()
{
  int arr[100] ;
  int x ;

  arr_init(arr, 100) ;

  x = sum (arr, 100) ;

  printf("sum : %d", x) ;
  return 0 ;
}
```

a. (10pt) Assume we have generated the assembly code for sum1.c. The function `main()` pushes two parameters, 100 and the start address of the `arr` array, to the stack and will execute `call sum`. The code will look something like the one in the below.

```
…
call sum
…
```

Function sum() is a recursive function. The main() calls sum(arr, 100). The sum(arr,100) will call sum(arr+1, 99) and so on. The recursion will stop when sum(arr+99,1) is called. Remind that EAX, ECX and EDX registers are saved by the caller and EBX, ESI and EDI registers are saved by the callee. Show the stack contents after the first two calls to `sum`; sum (arr, 100), and sum (arr+1, 99), i.e. till just before executing `call sum` with parameter 98. Please fill out the table below. Following are the system state just before executing `call sum` in `main();` 100 and &arr[0] have been pushed to the stack. `&arr[0]` corresponds to `0xffffd0cc` and the values of `esp` and `ebp` registers are `0xffffd0b0` and `0xffffd268`, respectively. For EAX, ECX, EDX, EBX, ESI and EDI, you do not have to write the register values. Instead, just write the register name at the value (or notes) field of the table.

Address	Value (4byte)	Notes
0xffffd05c		
0xffffd060		
0xffffd064		
0xffffd068		
0xffffd06c		
0xffffd070		
0xffffd074		
0xffffd078		
0xffffd07c		
0xffffd080		
0xffffd084		
0xffffd088		
0xffffd08c		
0xffffd090		
0xffffd094		
0xffffd098		
0xffffd09c		
0xffffd0a0		
0xffffd0a4		
0xffffd0a8		
0xffffd0ac		
0xffffd0b0	0xffffd0cc	&arr[0]
0xffffd0b4	100	

b. (8pt) Compute the stack size for `main()`. Fill out the values in the table below. The assembly code removes the parameters from the stack when the function returns. When the main() calls multiple functions, the stack space used for passing the parameters for each function can be reused across the function call.

stack of `main()`	Total size (Byte)	Variable names
Local variables		
Caller saved registers		
Callee save registers		
old ebp		
return address (old eip)		
`arr_init`: parameters		
`sum`: parameters		

`printf`: parameters		
Stack size total		N/A

c. (10pt) Compute total amount of stack used by `sum(arr, 100)` **until it returns**. Fill the table below to compute the stack size for each execution of `sum`. The last call in the recursion, `sum(arr+99,1)`, does not call any other function. The stack size for `sum(arr+99,1)` can be different from the preceding calls to `sum`.

For sum(100) ~ sum(2):

stack of `main()`	Total size (Byte)	Variable names
Local variables		
Caller saved registers		
Callee save registers		
old ebp		
return address (old eip)		
`sum`: parameters		
Stack size total		N/A

For sum(1):

stack of `main()`	Total size (Byte)	Variable names
Local variables		
Caller saved registers		
Callee save registers		
old ebp		
return address (old eip)		
`sum`: parameters		
Stack size total		N/A

Total size:

d. (2pt) Compute total amount of stack used by sum1.c, which corresponds to the summation of stack size for `main` (question 3.b) and the total stack size required for returning from `sum(arr, 100)` (question 3.c).

e. (5pt) We like to use the for-loop instead of the recursion in implementing the `sum`. Refer to the code below. Compute the stack size required to run sum1.c with the for-loop based `sum`. The stack size to run sum1.c will be the summation of the stack size of the `main()` and the stack size of for-loop based `sum()`. You can reuse the stack size of `main()` from question 3.b.

```
int sum (int* arr, int n)
{
```

```
    int s = 0 ;
      for (int i = 0 ; i < n ; ++i)
        s += arr [i] ;
    return s ;
}
```

new sum():

stack of `sum()`	Total size (Byte)	Variable names
Local variables		
Caller saved registers		
Callee save registers		
old ebp		
return address (old eip)		
function parameters		
Stack size total		N/A

Total size:

4.(25pt) fork and exec

Assume that fork() always succeeds. Assume that there is no stack overflow and no memory bloating.

a. (3pt) How many 'A' will the `foo1()` print?

```
void foo1(){
    fork() ;
    printf("A\n") ;
    fork() ;
    printf("A\n") ;
    fork() ;
    printf("A\n") ;
    fork() ;
    printf("A\n") ;
}
```

b. (7pt) List all possible different output sequences of function `foo3`. Assume that `execvp()` succeeds. In enumerating the output sequence, consider only A, B, C and D and exclude the output of `execvp()`. For example, "`A B outputofexecvp D`" and "`A outputofexecvp B D`" are the same sequence since both become "`A B D`" when you ignore `outputofevecvp. ()`. To get the full credit, provide the detailed reasoning.

```
void foo3(){
    int pid ;
    pid = fork() ;

    if (pid == 0) {
        printf ("A\n") ;
        char *argv[] = {"ls","-1", NULL};
        execvp ("ls", argv) ;
        printf ("B\n") ;
    }
    else {
        printf ("C\n") ;
    }
    printf("D\n") ;
}
```

c. (15pt) What is the number of different outputs that the function `foo2()` can generate?

```
void foo2(){
    fork() ;
    printf("A\n") ;
    fork() ;
```

```
    printf(“B\n”) ;
    fork() ;
    printf(“C\n”) ;
}
```

5.(24pt) Signal

a. (2pt) Which signal is generated as a result of pressing “Ctrl-C”?

(a) SIGKILL (b) SIGINT (c) SIGSTOP (d) SIGSEGV

b. (2pt) Select all functions that generate the signal.

(a) signal() (b) kill() (c) raise() (d) alarm()

Consider the following code `signal1.c`. The main function and the two signal handlers share the same global variable value. This program counts the number of SIGINT signals and the number of SIGALARM signal delivered to the process. We set the `setitimer` function to generate the `SIGALARM` in every 2 sec (wall clock time).

```
/* signal1.c */

#include <stdio.h>
#include <unistd.h>
#include <assert.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/time.h>
int value = 0 ;
int sigINTcount = 0 ;
int sigALARMcount = 0 ;
void SigINTHandler(int sig) {
   sigINTcount++ ;
   value ++ ;
}
void SigALRMHandler(int sig) {
   sigALARMcount ++ ;
   value++;
   signal(SIGALRM, SigALRMHandler);
}
int main(void) {
   int i = 0 ;
   struct itimerval MyTimer ;
   signal(SIGINT, SigINTHandler);
   signal(SIGALRM, SigALRMHandler);
   /* Send first signal in 1 second, 0 microseconds. */
   MyTimer.it_value.tv_sec = 1;
   MyTimer.it_value.tv_usec = 0;
   /* Send subsequent signals in 1 second,
      0 microseconds intervals. */
   MyTimer.it_interval.tv_sec = 2 ;
   MyTimer.it_interval.tv_usec = 0;
   setitimer(ITIMER_REAL, &MyTimer, NULL);
   while(++i) {
      sleep(1) ;
      value++ ;
   }
}
```

c. (5pt) When there arrive multiple signals of the same type while the signal is blocked, only one signal is delivered to the process after the signal is unblocked. In the above program, we like to count the total number of sleep calls, the number of SIGINT's delivered to the process and SIGALRM's that are delivered to the process using the variable `value`, i.e. `value = i +`

sigINTcount + sigALARMcount. Very rarely, the program does not behave correctly and the above condition does not hold. Explain the reason.

d. (15pt) Modify signal1.c to fix the problem stated in the above question. Please refer to the following signal manipulation functions. You may use some of these functions if necessary.

```
int sigemptyset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

You can use SIG_BLOCK or SIG_UNBLOCK in how field of sigprocmask to block or unblock the signal, respectively.

6. (10pt) Optimization

Followings are the techniques to make the program run faster. Please explain the reason why it makes the program run faster

a. (3pt) Use inline function (or macro) instead of using the normal function call.

b. (3pt) Unroll the loops.

c. (4pt) What is the disadvantage of using “inline function” or “unroll loops”.

7. (20pt) IO

Consider two ways to read the data from the file: `fread()` and `read()`. Assume that “sample.txt” contains enough amount of data to read and `fread()` and that `fread()` and `read()` always succeed.

a. (4pt) Consider readbuffer1.c. How many times, do the fread()’s in readbuffer1 get into the kernel? You can assume that stream buffer size is 8 Kbyte.

```
/* readbuffer1.c */

#include <stdio.h>

 int main(void)
 {
     FILE *file_ptr;
     char arr[8192];

     file_ptr = fopen("sample.txt", "rb");
     if(file_ptr==NULL) return 1;

     fread(arr, sizeof(char), 8192, file_ptr);
     fread(arr, sizeof(char), 8192, file_ptr);
     fread(arr, sizeof(char), 8192, file_ptr);

     fclose(file_ptr);

     return 0;
 }
```

b. (4pt) Consider readbuffer2.c. How many times, do the read()’s in readbuffer2 get into the kernel?

c. (4pt) Which of readbuffer1.c and readbuffer2.c runs faster? Please explain the reason. Provide detailed reasoning to get the full credit.

```
/*readbuffer2.c */

#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

 int main(void)
 {
     int fd;
     char arr[8192];


     fd = open("sample.txt", O_RDONLY);
     if(fd<0) return 1;

     read(fd, arr, 8192);
     read(fd, arr, 8192);
     read(fd, arr, 8192);

     close(fd);

     return 0;
 }
```

d. (4pt) Consider `readbuffer3.c`. How many times, do the `fread()`'s in `readbuffer3.c` get into the kernel?

e. (4pt) Consider `readbuffer4.c` How many times, do the `read()`'s in `readbuffer4.c` get into the kernel?

f. (4pt) Which of the `readbuffer3.c` and `readbuffer4.c` do you think runs faster? Provide the detailed reasoning for the answer.

```
/* readbuffer3.c */

#include <stdio.h>

 int main(void)
 {
```

```
    FILE *file_ptr;
    char arr[8192];


    file_ptr = fopen("sample.txt", "rb");
    if(file_ptr==NULL) return 1;

    fread(arr, sizeof(char), 1, file_ptr);
    fread(arr, sizeof(char), 1, file_ptr);
    fread(arr, sizeof(char), 1, file_ptr);

    fclose(file_ptr);

    return 0;
}
```

```
/*readbuffer4.c */

#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

int main(void)
{
    int fd;
    char arr[8192];


    fd = open("sample.txt", O_RDONLY);
    if(fd<0) return 1;

    read(fd, arr, 1);
    read(fd, arr, 1);
    read(fd, arr, 1);

    close(fd);

    return 0;
}
```