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

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 have 165 minutes (1:00 PM – 3:45 PM) to complete your exam. 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) 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.

   Answer:

   Value A: 0xFFFFF8CA, decimal: -1846

   Value B: 0xFFFFF8EB, decimal: -1813

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.

   Answer:

   Value A: 0xCAF8FFFF, decimal: -889651201

   Value B: 0xEBF8FFFF, decimal: -336003073

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.

   Answer:

   ZF: 0 , SF: 0, CF: 0, OF: 0

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. A binary image consists of a number of sections. They include are .bss, .data, .text and etc. Some of the symbols may not have been assigned a section by the assembler. For the symbols that are not assigned a section, mark the section name of those symbols as 'X'.

solution:

label	section	Local?
msg	.rodata	local
main	.text	global
skip	.text	local
getchar	???	global
prinf	???	global

b. (3pt) Which of the symbol references need relocation? Specify the line numbers.

Answer: All references to the symbols that are not defined within the .text section need relocation.

Reference to getchar: line 9

Reference to mgs: line 12

Reference to prinf: link 13

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.

Answer:

getchar, prinf

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?

Answer:

The compiler fails at the pass 1 of the link (symbol resolution). The compiler fails to locate the symbol `prinf` in the symbol table of the `libc` library.

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

We write a function to sum all variables in the array. Assume IA32 architecture. The assembly code generated by the compiler varies widely dependent upon the internal algorithm of the compiler. Modern compiler adopts highly sophisticated algorithm to make the binary image faster and smaller.

Assume that all local variables are allocated in the stack. In the course of evaluating an expression, assume that all temporary values are stored in the six general purpose registers (`EBX`, `ESI`, `EDI`, `EAX`, `ECX` and `EDX`) and the temporary variables do not occupy the stack space.

```
/* 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) Think about the assembly code of the sum1.c. In `main()`, the caller will push the two parameters to the stack to call `sum`. The parameters pushed to the stack correspond 100 and the start address of the `arr` array. After pushing the two parameters, the caller `main()` will execute the following call instruction to jump to function `sum`.

```
...
call sum
...
```

Show the stack contents after the first two calls to `sum`; sum (arr, 100), and sum (arr, 99). Show the stack contents till just before `call` sum with parameter 98. Assume that the caller saves EAX, ECX and EDX registers. The callee saves EBX, ESI and EDI registers. Following are the system state just before executing `call sum` in `main()`. At this point, parameters 100 and &arr[0] have been pushed to the stack. Please make the assumption if necessary.

Assume that address of `&arr` is `0xffffd0cc`.

The current values of `esp` and `ebp` registers just before `call` to sum (100) are `0xffffd0b0` and `0xffffd268`, respectively.

Address	Value
0xffffd05c	
0xffffd060	
0xffffd064	
0xffffd068	
0xffffd06c	
0xffffd070	
0xffffd074	
0xffffd078	
0xffffd07c	
0xffffd080	
0xffffd084	
0xffffd088	
0xffffd08c	
0xffffd090	
0xffffd094	
0xffffd098	
0xffffd09c	
0xffffd0a0	
0xffffd0a4	
0xffffd0a8	
0xffffd0ac	
0xffffd0b0	0xffffd0cc, address of arr

0xffffd0b4	100

Answer:

Address	Value	Note
0xffffd05c	0xffffd0b4	&arr[2]
0xffffd060	98	
0xffffd064		pushed edx
0xffffd068		pushed ecx
0xffffd06c		pushed eax
0xffffd070		pushed edi
0xffffd074		pushed esi
0xffffd078		pushed ebx
0xffffd07c		pushed ebx
0xffffd080	0xffffd0a8	old ebp
0xffffd084		old eip
0xffffd088	0xffffd0b0	&arr[1]
0xffffd08c	99	
0xffffd090		pushed edx
0xffffd094		pushed ecx
0xffffd098		pushed eax
0xffffd09c		pushed edi
0xffffd0a0		pushed esi
0xffffd0a4		pushed ebx
0xffffd0a8	0xffffd268	old ebp
0xffffd0ac		old eip
0xffffd0b0	0xffffd0cc	&arr[0]
0xffffd0b4	100	

b. (8pt) Compute the stack size for `main()`. Fill out the values in the table below. The stack removes the parameters from the stack it the function returns. When the main() calls multiple functions, the stack space used for passing the parameters 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

Answer:

stack of `main()`	Total size (Byte)	Variable names
Local variables	404	`int arr[100];` `int x;`
Caller saved registers	12	`ebx; esi; edi;`
Callee save registers	12	`eax; ecx; edx`
old ebp	4	`ebp`
return address (old eip)	4	`eip`
`arr_init`: parameters	8	`int *arr;` `int size;`
`sum`: parameters	8	`int *arr;` `int size;`
`printf`: parameters	8	`char *str`
Stack size total	444	N/A

c. (10pt) Compute total amount of stack used by `sum(arr, 100)` until it returns. Fill the table in the below.

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:

Answer (sum(100) ~ sum(2)):

stack of `main()`	Total size (Byte)	Variable names
Local variables	0	

Caller saved registers	12	ebx; esi; edi;
Callee save registers	12	eax; ecx; edx
old ebp	4	ebp
return address (old eip)	4	eip
sum: parameters	8	int *arr; int size;
Stack size total	40	N/A

Answer (sum(1)):

stack of main()	Total size (Byte)	Variable names
Local variables	0	
Caller saved registers	0	
Callee save registers	12	eax; ecx; edx
old ebp	4	ebp
return address (old eip)	4	eip
sum: parameters	0	
Stack size total	20	N/A

Total size: 40*99 + 20 = 3960 + 20 = 3980byte

d. (2pt) Compute total amount of stack used by sum1.c (Hint: Summation of stack size for `main` and the stack size of `sum(arr, 100))`

Answer:

(main's stack size) + (sum's stack size) = 444byte + 3980byte = 5024byte

e. (5pt) We like to use the for-loop instead of the recursion in taking the sum. Refer to the code below. Compute the stack size required to run sum1.c with the sum function in the below. To get the full credit, provide the detailed step of your computation.

```
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)		
`sum`: parameters		
Stack size total		N/A

Total size:

Answer (new sum()):

stack of `sum()`	Total size (Byte)	Variable names
Local variables	8	`int s;` `int i;`
Caller saved registers	0	
Callee save registers	12	`eax; ecx; edx`
old ebp	4	`ebp`
return address (old eip)	4	`eip`
`sum`: parameters	0	
Stack size total	28	N/A

Total size: (main size) + (new sum() size) = 444 + 28 = 472byte

4.(25pt) fork and exec

Assume that all fork()'s always succeed. 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") ;
}
```

Answer: 30

b. (7pt) List all possible outputs of function `foo3()`. To get the full credit, provide detailed reasoning. Assume that `execvp()` succeeds. Do not consider the output of execvp. In enumerating the output sequence, please consider only A, B, C and D and exclude the output of `execvp()`

```
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") ;
}
```

Answer: 3

CDA
CAD
ACD

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") ;
}
```

Solution

Step 0. Consider the following code.

```
void foo0(){
        printf("A\n") ;
        fork() ;
        printf("B\n") ;
}
```

Possible output

```
 ABB
```

Step 1. Consider the following code.

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

There are two processes. Each will print `ABB`. Possible output sequence: 3 cases.

AABBBB
ABABBB
ABBABB
ABBBAB → this cannot happen.

Step 2:

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

The total number of C's that exist until a certain position `Pi` cannot be greater than twice the number of B's that exist until that position.

Consider string S. Let N(S) be the number of different strings from foo2() that contains S as its substring.

1. AABBBB: N(AABBBB) = 55 (25+18+12)

   AAB(P1)B(P2)B(P3)B(P4)

   All three conditions below should always hold.

   a. P1 <= 2
   b. P1 + P2 <= 4
   c. P1 + P2 + P3 <= 6

      (P1,P2,P3)
      Case 1: P1 = 0, 25 cases

         (P2 <=4) and (P2 + P3 <= 6) = 3+…+7

         (0,0,P3) → P3 <=6: 7 cases
         (0,1,P3) → P3 <=5: 6 cases
         (0,2,P3) → P3 <=4: 5 cases
         (0,3,P3) → P3 <=3: 4 cases
         (0,4,P3) → P3 <=2: 3 cases

      Case 2: P1 = 1, 18 cases

         (P2 <=3) and (P2 + P3 <= 6) = 3+…+6

         (1,0,P3) → P3 <=5: 6 cases
         (1,1,P3) → P3 <=4: 5 cases
         (1,2,P3) → P3 <=3: 4 cases
         (1,3,P3) → P3 <=2: 3 cases

      Case 3: P2 = 2, 12 cases

         (P2 <=2) and (P2 + P3 <= 6) = 3+…+5

         (2,0,P3) → P3 <=4: 5 cases
         (2,1,P3) → P3 <=3: 4 cases
         (2,2,P3) → P3 <=2: 3 cases

2. ABABBB: N(ABABBB) = 97 (25 + 36 + 36)

AB(P1)A(P2)B(P3)B(P4)B(P5)

All four conditions below should always hold.

a. P1 <= 2
b. P1 + P2 <= 2
c. P1 + P2 + P3 <= 4
d. P1 + P2 + P3 + P4 <= 6

Let's consider the following (P1,P2,P3,P4).

Case 1: P1+P2 = 0, 25 cases
(0,0,0,P4) → P4 <=6: 7 cases
(0,0,1,P4) → P4 <=5: 6 cases
(0,0,2,P4) → P4 <=4: 5 cases
(0,0,3,P4) → P4 <=3: 4 cases
(0,0,4,P4) → P4 <=2: 3 cases

Case 2: P1 + P2 = 1, 18*2 = 36 cases
(0,1,0,P4) → P4 <=5: 6 cases
(0,1,1,P4) → P4 <=4: 5 cases
(0,1,2,P4) → P4 <=3: 4 cases
(0,1,3,P4) → P4 <=2: 3 cases
Same for (1,0,*,*)

Case 3: P1 + P2 = 2, 12*3 = 36 cases
(1,1,0,P4) → P4 <=4: 5 cases
(1,1,1,P4) → P4 <=3: 4 cases
(1,1,2,P4) → P4 <=2: 3 cases
Same for (2,0,*,*) and (0,2,*,*)

3. ABBABB: N(ABBABB) = 65 + 40 + 22 = 127

AB(P1)B(P2)A(P3)B(P4)B(P5)

All four conditions below should always hold.

a. P1 <= 2
b. P1 + P2 <= 4
c. P1 + P2 + P3 <= 4
d. P1 + P2 + P3 + P4 <= 6

Let's consider the following (P1,P2,P3,P4).

Case 1: P1 = 0, 65 cases
(0,0,0,P4) → P4 <=6: 7 cases
(0,0,1,P4), (0,1,0,P4) → P4 <=5: 6 cases *2 = 12
(0,0,2,P4) → P4 <=4: 5 cases *3 = 15
(0,0,3,P4) → P4 <=3: 4 cases *4 = 16

```
        (0,0,4,P4) → P4 <=2: 3 cases *5 = 15

    Case 2: P1 = 1, 40 cases
        (1,0,0,P4) → P4 <=5: 6 cases
        (1,0,1,P4) → P4 <=4: 5 cases *2 = 10
        (1,0,2,P4) → P4 <=3: 4 cases *3 = 12
        (1,0,3,P4) → P4 <=2: 3 cases *4 = 12

    Case 3: P1 = 2, 22 cases
        (2,0,0,P4) → P4 <=4: 5 cases
        (2,0,1,P4) → P4 <=3: 4 cases *2 = 8
        (2,0,2,P4) → P4 <=2: 3 cases *3 = 9

Total       = N(AABBBB) + N(ABABBB) + N(ABBABB)
            = 55 + 97 + 127 = 279
```

채점기준

계산의 정확성보다 접근방식이 제대로 되었는지를 중점적으로 체크.

- 아래 조건, 혹은 유사한 조건을 포함한 답안: 7점
  The total number of C's that exist until a certain position `Pi` ***cannot be greater than twice the number of B's that exist until that position.***

- 각 케이스에 대해서 아래와 같은 조건이나 유사한 조건을 제대로 정의한 답안: 10점

```
Case 1:
  a. P1 <= 2
  b. P1 + P2 <=4
  c. P1 + P2 + P3 <= 6

Case 2:
  a. P1 <= 2
  b. P1 + P2 <= 2
  c. P1 + P2 + P3 <= 4
  d. P1 + P2 + P3 + P4 <= 6

Case 3:
  a. P1 <= 2
  b. P1 + P2 <= 4
  c. P1 + P2 + P3 <= 4
  d. P1 + P2 + P3 + P4 <= 6
```

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 `setitimerval` 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 process is blocked, only one signal is delivered to the process. In the above program, we like to count the total number of sleep calls, the number of SIGNIT's delivered to the process and SIGALRM's that are delivered to the process. `value = i + sigINTcount + sigALARMcount`. Very rarely, the program does not behave correctly and the above condition does not hold. Explain the reason.

Answer: The main program and the signal handlers share the global variable. There can arise a race condition among them.

d. (15pt) Modify signal1.c to fix the problem stated in the above question. Please refer to the following signal manipulation functions.

Followings are the function prototypes and keyword you may use.

```
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.

Answer: in the main program, you have to block the two signals. In each signal hander, it needs to block the other signal of sigINT and sigALRM.

```
/* signal1.c */
int value = 0 ;
int sigINTcount = 0 ;
int sigPROFcount = 0 ;

void SigINTHandler(int sig) {
      sigINTcount++ ;
      sigset_t mysigset ;

      sigemptyset (&mysigset) ;
      sigaddset (&mysigset, SIGALRM) ;

      sigprocmask (SIG_BLOCK, &mysigset, NULL) ;
       value ++ ;
      sigprocmask (SIG_UNBLOCK, &mysigset, NULL) ;
}

void SigALARMHandler(int sig) {
      sigPROFcount++ ;
      sigset_t mysigset ;

      sigemptyset (&mysigset) ;
      sigaddset (&mysigset, SIGINT) ;

      sigprocmask (SIG_BLOCK, &mysigset, NULL) ;
      value ++ ;
      sigprocmask (SIG_UNBLOCK, &mysigset, NULL) ;
}

int main(void) {
```

```
    int i = 0 ;
    struct itimerval MyTimer ;

    signal(SIGINT, SigINTHandler);
    signal(SIGALRM, SigALARMHandler) ;

sigset_t mysigset ;
sigemptyset (&mysigset) ;
sigaddset (&mysigset, SIGINT) ;
sigaddset (&mysigset, SIGALRM) ;

/* 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) ;
      sigprocmask (SIG_BLOCK, &mysigset, NULL) ;
      value++ ;
      sigprocmask (SIG_UNBLOCK,&mysigset, NULL) ;
}
}
```

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.

Answer: Eliminate the overhead of making a function call which include copying the values to and from the stack.

b. (3pt) Unroll the loops.

Answer: Exploit pipelining and superscalar feature of the CPU

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

Answer: There can be many answers but if the answer contains that “the code size becomes larger”, give the full credit.

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.

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;
 }
```

Answer: three times. Each fread() causes read().

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

Answer: three times. Each read() gets 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.

Answer: using read() is faster than using fread(). fread() requires the memory copy from the kernel buffer to the stream buffer and from the stream buffer to the use buffer. Whereas, read() copies the data from the kernel buffer to the user buffer. In read(), the memory copy from the kernel buffer to the stream buffer is omitted.

```
/*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 get into the kernel?

   Answer: once. First one calls read() system call. Second and the third one are serviced from the stream buffer.

e. (4pt) Consider read buffer4.c How many times, do the read()'s in readbuffer4.c get into the kernel?

   Answer: three times. Each of the read() goes 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.

Answer: readbuffer3.c will run faster since it does not go into kernel and has less amount of system call overhead.

```
/* 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;
}
```