ECE391 Exam 1, Fall 2021, CONFLICT

Wednesday 29 September

Name and NetID:

- **Write your name at the top of each page.**
- **This is a closed book exam.**
- **You are allowed TWO 8.5×11" sheet of notes.**
- **Absolutely no interaction between students is allowed.**
- **Show all of your work.**
- **Reference page(s) are attached at the end of the exam.**
- **Don't (kernel) panic, and good luck!**

Problem 1	20	points	
Problem 2	17	points	
Problem 3	22	points	
Problem 4	21	points	
Problem 5	18	points	
Total	98	points	

Name: ____

Problem 1 (20 points): Short Answer

Part A (4 points): List **TWO** roles of system software. Please use ONE WORD for each role. We mentioned three such roles in the lecture.

(i) Abstraction

(ii) Protection

Vistualization

Part B (3 points): The x86 uses a single vector table called the Interrupt Descriptor Table, or IDT, for **THREE** types of ”interruptions”. Please list them.

(i) System Call

(ii) Exception

(iii) Interruption

Part C (3 points): Assume the following code is being executed on a uniprocessor, **USING NO MORE THAN TWENTY WORDS**, explain what are the unnecessary parts of the following code.

```
cli();
spin_lock(&the_lock);
/* critical section code */
spin_unlock(&the_lock);
sti();
```

Using the lock is unnecessary as there is no other processors and cli and sti is enough for masking interruptions.

Part D (4 points): Recall the user-level test harness provided to you for MP1 (the test harness that simulates the test machine's environments). **USING NO MORE THAN FIFTEEN WORDS EACH**, explain one advantage and one disadvantage of developing and using such a harness compared with debugging fully inside the Linux kernel.

ADVANTAGE:

can have faster development cycle
easier to fix bugs, which may crash kernel

DISADVANTAGE:

may not expose all bugs

may not work in kernel

Part E (2 points): IDENTIFY the bug(s) in the following snippet of code for the dispatcher function from MP1 and CORRECT the bug(s).

```
mp1_ioctl:
    cmpl $4, 8(%ebp)
    ja invalid_cmd
    movl 8(%ebp), %eax
    jmp *jump_table(,%eax,4)
invalid_cmd:
    movl $-1, %eax
    leave ①
    ret
```

in the line ①
should not "LEAVE" as no %ebp is
pushed onto stack
remove line ①

Part F (4 points): Assume that in order to access shared data your program needs to acquire a spinlock and a semaphore at the same time. **USING NO MORE THAN THIRTY WORDS**, explain which primitive should you acquire first and why.

Should acquire semaphore first and spinlock then.
Because semaphor allows other programs to run while waiting
and other programs may also want to acquire the same lock,
causing deadlock if order are reversed.

Problem 2 (17 points): MP1

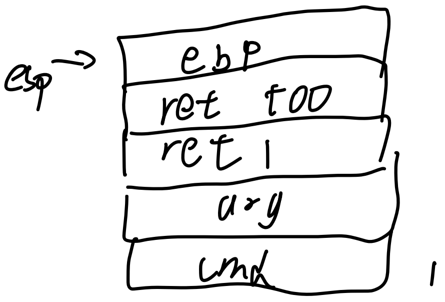
Parts A, **B**, **C** refers to the following function.

Consider a function `foo` which makes use of a jump table similarly to `mp1_ioctl` from MP1. The following is a specification of the function.

```
int foo(int arg, int cmd);
/*
    you can assume that 10 >= arg && arg >= 0
    if cmd==0, return arg
    if cmd==1, return arg + 1
    otherwise, return 0
*/
```

The following implementation of the function `foo` has a bug.

```
foo:
    movl 8(%esp),%eax
    cmpl $1,%eax
    ja cmd_invalid
    ~~call~~ *jump_table(,%eax,4)
cmd_invalid:
    movl $0,%eax
    ret

jump_table:
    .long bar, baz

bar:
    pushl %ebp
    movl %esp, %ebp
    movl 8(%ebp), %ecx
    movl %ecx, %eax
    leave
    ret

baz:
    pushl %ebp
    movl %esp, %ebp
    movl 8(%ebp), %ecx
    addl $1, %ecx
    movl %ecx, %eax
    leave
    ret
```



Problem 2, continued:

Part A (3 points): Suppose you called the function `foo` in the `main` function with the following line without fixing the bug.

```
int result=foo(1, 1);
```

What is the value stored in `result` after returning from `foo`?

0

Result: ~~address of the return address of function foo plus one~~

Part B (3 points): Fix the bug by modifying one line. Indicate the number of the line you would fix and write the modified instruction.

Line Number: 5

Instruction: jmp *jump_table(,%eax,4)

Problem 2, continued:

Part C (4 points): After fixing the bug, you were asked to extend the functionality of `foo` so that it satisfies the following specification.

```
int foo(int arg, int cmd);
/*
if cmd==2, return arg + 7
for any other cmd, behave exactly the same as the previous version of foo
*/
```

To extend the functionality of `foo`, you implemented the following function `quz` and attached it below the implementation of `baz`.

```
quz:
    pushl %ebp
    movl %esp, %ebp
    movl 8(%ebp), %ecx
    addl $7, %ecx
    movl %ecx, %eax
    leave
    ret
```

Besides implementing and attaching `quz`, what are the two changes you should make in `foo`? Indicate the line number where you would make the change and the modified instruction.

Line Number: 3

Instruction: cmpl $2, %eax

Line Number: 11

Instruction: .long bar, baz, quz

Problem 2, continued:

Part D, E are independent of the previous parts.

Part D (4 points): Your friend is struggling with an issue on MP1 where the missiles are flickering occasionally. You suspect that the faulty implementation of `mp1_addmissile` is causing the bug. Place the following steps in the right order so that your friend can resolve the issue(write down the alphabet in the right order).

(a) Copy missile data from the user space to the kernel space

(b) Set the head pointer to the new node

(c) Allocate memory for the missile structure

(d) Set the next pointer of the new node to the head

First step: c

Second step: a

Third step: d

Fourth step: b

7

Part E (3 points): Consider an alternative implementation of missilecommand game as in MP1, but with 10 rows, each row being 20 characters wide. That is, the screen has row 0 through row 9 and column 0 through column 19. Suppose you wish to modify the ascii code located at row 6, column 12. If the video memory starts at 0x00000000, what is the memory address you would modify?

Address: 0x 0000108

$\overset{120}{} \; \overset{132}{}$

$(6 \cdot 20 + 12) \cdot 2$

264

$= 256 + 8$

$2^8 + 2^3$

0001 0000 1000

Problem 3 (20 points): x86 Assembly

The k-means Clustering algorithm is used widely to partition N data points into k clusters. The algorithm can be divided into two steps that are repeated until desired. They are:

1. **Assignment:** Assign each data point to a cluster based on a distance metric to the nearest mean
2. **Update:** Update the mean point for each cluster

The C code below implements the k-means algorithm.

Note that the arrays `data`**,** `means` **and** `assignments` **are declared globally and visible to all functions.**

```
#define ITERATIONS
#define K 10
#define N 200

// HINT: the packed attribute ensures no extra padding. In other words,
//       size of the struct will be the sum of the size of all data
//       that are defined in the struct

typedef struct __attribute__((__packed__)) point_t {
    uint8_t data;
} point_t;

point_t data[N];
point_t means[K];
int32_t assignments[N];

void kmeans() {
    register int32_t i; // register keyword insures the variable
                           is stored in the register, not on the stack
    for (i = 0; i < ITERATIONS; i++) {
        assign_clusters();
        update_means();
    }
}
```

indicate sizeof(point_t)=1

Problem 3, continued:

```
void assign_cluster() {
   int32_t i, j;
   int32_t min_dist, dist;

   for (i = 0; i < N; i++) {
       min_dist = 2147483647;  # 0x7FFF FFFF

       for (j = 0; j < K; j++) {
           dist = distance(data + i, means + j);
           if (dist < min_dist) {
               assignments[i] = j;
               min_dist = dist;
          }
       }
   }
}

void update_means() {
   int32_t i;
   uint32_t j, mean_data, count;

   for (i = 0; i < K; i++) {
       mean_data = 0;
       count = 0;

       for (j = 0; j < N; j++) {
           if (assignments[j] == i) {
               mean_data += data[j].data;
               count += 1;
            }
       }
       // Assume the below quotient is a 8-bit integer.
       means[i].data = (uint8_t) (mean_data / count);
   }
}
```

where the `distance` function has the following interface:

```
/*
 * Calculates the distance between the two points
 * Input: point1, point2 - pointers to the two input points
 * Output: Distance between the two points
*/
int32_t distance(point_t* point1, point_t* point2);
```

Problem 3, continued:

Part A (8 points): **So you think you can Stack?**
The `assign_cluster` function is called from `kmeans` as shown above. **Based on the C function on the previous page**, Complete the diagram below to represent the stack right AFTER the instruction at line 6 has been executed for the FIRST time.
Using the C Calling convention, label each location on the stack, and write their values iff you know them from the information given, otherwise label them as unknown. Leave unused locations blank. **Also indicate the locations to which ESP and EBP point to.**

Assume that `assign_cluster` **uses the registers EDI, ESI, EBX for variables** i, j **and** $dist$ **and no other registers for the variables. Do NOT save unused registers on the stack.** Also assume that the `main` function calls the `kmeans` function.

Address	Label	value
0xBEEF0200	(1)	(2)
0xBEEF0204	(3)	(4)
0xBEEF0208	(5) min-dist	(6) 2147483647
0xBEEF020C	(7) EBX(variable dist)	(8) unknown
0xBEEF0210	(9) ESI(variable j)	(10) unknown
0xBEEF0214	EDI(variable i)	0x0
0xBEEF0218	(11) kmeans's EBP	(12) 0xBEEF0220
0xBEEF021C	Return address to `kmeans`	unknown
0xBEEF0220	main's EBP	unknown
0xBEEF0224	Return address to `main`	unknown

ESP: 0xBEEF0208

EBP: 0xBEEF0218

Problem 3, continued: Part B (12 points): ***¿do you really know ×86 Assembly?***
Fill in the blanks in the code below with **at most one valid instruction per blank** (you may not need all blanks) to complete the translation of the `update_means` function from C to x86 Assembly. **You may not add additional lines, nor cross out existing lines. The registers MUST be used for the purposes indicated below, any changes will results in a loss of points. Adding additional instructions that are not needed will also result in a loss of points.**

```
    # EAX: mean_data
    # EBX: i  对k
    # ECX: j  对N
    # EDI: count
    # ESI: reusable at your will
    # EDX: used for division, other times reusable at your will
    #
    # Unsigned division(DIV):
    #    DIV %EBX     # Unsigned divide (EDX:EAX) by EBX
    #                 # After DIV: EAX stores quotient, EDX stores remainder
    # HINT: (EDX:EAX) represents a 64-bit number, whose higher 32 bits stored
    #       in EDX and lower 32 bits stored in EAX


update_means:
    pushl %ebp
    movl %esp, %ebp

    pushl %ebx
    pushl %edi
    pushl %edx

    xorl %ebx, %ebx

CLUSTER_LOOP:
    cmpl $K, %EBX________________             # Fill in missing instruction (i)
    jge CLUSTER_LOOP_DONE
    xorl %eax, %eax    mean=0
    xorl %edi, %edi    count=0
    xorl %ecx, %ecx    j=0
```

j %ECX
i %EBX
assign[j]==i

Problem 3, continued:

```
DATA_LOOP:
    cmpl $N, %ECX                                   # Fill in missing instruction (ii)
    jge DATA_LOOP_DONE
    movl  assignments(,%ECX,4),%ESI                 # Fill in missing instruction (iii)
元素为int32_t
  4byte
    cmpl  %ESI,%EBX                                 # Fill in missing instruction (iv)
    jne  SKIP
    movzbl data(,%ECX,1),%ESI                       # Fill in missing instruction (v)
    addl  %ESI,%EAX                                 # Fill in missing instruction (vi)
    ~~addl  $1, %EDI~~                              # Fill in missing instruction (vii)
    incl %edi

SKIP:
元素为point_t
    incl %ecx
    jmp DATA_LOOP
1byte


DATA_LOOP_DONE:
    xorl %edx, %edx
    DIV  %EDI                                       # Fill in missing instruction (viii)
    movb   %AL, means(,%EBX,1)                      # Fill in missing instruction (ix)
    incl %ebx
    jmp CLUSTER_LOOP

CLUSTER_LOOP_DONE:
    popl %edx
    popl %edi
    popl %ebx
    leave
    ret
```

Problem 4 (21 points): Synchronization

Read the following code to answer Part A and Part B

```
int a,b,c,d,e;

func1(){
    a++;
    e = a + d;
    b--;
}

func2(){
    e = c + 225;
    a--;
}

func3(){
    d = b - d;
}


func4(){
    c = b + 391;
}
```

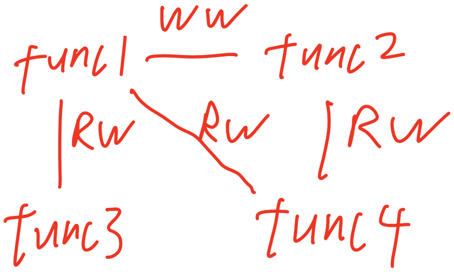
Part A (4 points):
Suppose code using the above functions could be run concurrently (e.g., on a multiprocessor). Draw a dependency graph between functions where each node is a function and edges imply a read-write or a write-write relationship. Name the nodes func1, func2, func3, and func4. It may help to write out read ~~and write sets for each function.~~ ?

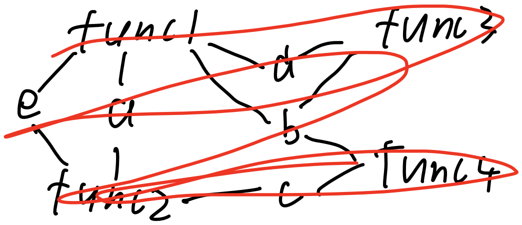
func1 : { read : d
write : a, b, e

func2 : { read : c
write : a, e

Func3 : { read : b
write : d

func4 : { read : b
write : c





Part B (4 points):
Suppose we decide to have one lock for each of the five variables and the code using these locks call their locks in the following orders

T1	T2	T3	T4
lock(a)	lock(c)	lock(b)	lock(b)
lock(d)	lock(e)	lock(d)	lock(c)
lock(e)	lock(a)	unlock(d)	unlock(c)
lock(b)	unlock(a)	unlock(b)	unlock(b)
unlock(b)	unlock(e)	—-	—-
unlock(e)	unlock(c)	—-	—-
unlock(d)	—-	—-	—-
unlock(a)	—-	—-	—-

Indicate in the table below if a deadlock occurs between pairs of executing code. Use a ”D” to indicate a deadlock occurs and leave the cell blank if a deadlock does not occur.

	T1	T2	T3	T4
T1	XX	D	D	
T2	XX	XX		
T3	XX	XX	XX	
T4	XX	XX	XX	XX

Part C (3 points):
Recall that `spin_lock_irqsave` calls CLI first and then locks the lock. Now consider the function `spin_lock_irqrestore`: does it matter in which order it restores flags and unlocks the lock?
First choose ”yes” or ”no”. Then, give a brief reasoning to justify your answer using no more than thirty words.

CIRCLE ONE: [✓ **Yes**] **No**

if restores flags first, interruption may happen right afther it and may acquire the same lock ⇒ deadlock

Part D (10 points): *Suppose a brandly new cryptocurrency called dogcoin has been invented, and there are a number of dogs using dogcoins: they earn dogcoins by running mining programs. Using the mined dogcoins, dogs can pay for their meals at a buffet. These dogs have agreed to pool their money so as many dogs can dine at the buffet as possible.*
These dogs can take 3 possible actions which we express as C functions: **checkin_buffet()**, **exit_buffet()**, and **mine_dogcoins()**. These 3 functions will use the shared variables in the code below. Note: **These 3 functions are the only functions in the universe that modify these shared variables**. Treat each dog as a thread on a multiprocessor system that could be running any of these 3 functions.

Your task is to complete the code below for the two functions **checkin_buffet()** and **mine_dogcoins()**. The behavior of these function is as follows:

- **checkin_buffet()**: Dogs will check if there is enough money to eat at the buffet and if the buffet has space for them. If both of these are true it will take money for a meal at the buffet and return 1 for success. The cost of the buffet is defined in BUFFET_COST, and due to the COVID-19 pandemic the capacity limit for diners at the buffet is defined in CAPACITY. If the dog fails to checkin the buffet, simply return -1.
- **mine_dogcoins()**: Dogs wait until there is enough money to pay for electricity to mine dogcoins. Once there is enough money to pay for electricity, the dog should take the money to pay for electricty and mine the dogcoin by calling **mine_func()**. **mine_func()** a random amount of dogcoins mined. The mined dogcoins are then added to the total dogcoin count. **mine_dogcoins()** should return the amount of dogcoins earned by mining. If there is 0 dogcoin left and no other dogs are mining dogcoin, **mine_dogcoins()** should return -1 for failure. Mining dogcoins is the ONLY way dogs can make money. Note that mining is a very complicated mathematical operation so it takes a long time. To allow other processes run concurrently, we should Not hold the lock when executing the **mine_func()**. The cost of electricity to mine a dogcoin is defined in MINE_COST.

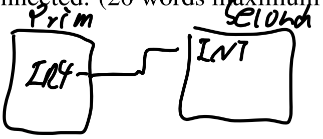
Fill in the blanks in the code below to accomplish the behavior described above. Use synchronization to prevent race conditions while also maintaining maximum parallelism. That means, **do not hold a lock if you don't need it**. You do not need to fill in all blanks. Assume that the given global variables are already initialized including spinlock_t* lock.

```
#define CAPACITY 6
#define BUFFET_COST 100
#define MINE_COST 1

volatile int dogcoin_amount;
volatile int num_dining;
volatile int num_mining;
spinlock_t* lock;

int checkin_buffet(){
    (1) spinlock_lock(lock);
    if( (2) dogcoin_amount >=BUFFET_COST && num_dining < CAPACITY )
    {
        (3)____________________________________
        num_dining++;
        dogcoin_amount -= BUFFET_COST;
        (4) spinlock_unlock(lock);
        return 1;
    }
    else{
        (5) spinlock_unlock(lock);
        return -1;
    }
}

/* *****CODE CONTINUED ON NEXT PAGE**** */
```

```
int mine_dogcoins(){

    int amount_mined = 0;
    while(1){
        (6) spinlock_lock(lock);
        if(dogcoin_amount >= MINE_COST){
            dogcoin_amount -= MINE_COST;
            num_mining++;

            (7) spinlock_unlock(lock);

            amount_mined += mine_func();

            (8) spinlock_lock(lock);

            (9) dogcoin_umount += amount_mined;

            (10) num_mining--;
            spin_unlock(lock);
            return (11) amount-mined ;
        }
        else if (dogcoin_amount <= 0 &&
            (12) num_mining <=0 ){
            spin_unlock(lock);
            return -1;
        }
        spin_unlock(lock);
    }
}
```

Problem 5 (18 points): Programmable Interrupt Controller

Part A (6 points): For each of the following signals on a **slave** 8259A PIC chip in a cascade setup where the slave is connected to IR 4 on the master and they each has some devices connected, explain what will happen if **each of the following signal gets shorted individually** such that it always reads **low**. In each case only the single signal being considered is malfunctioning and all other signals are working properly. For full points, explain how it will impact the functioning of the devices / other PICs connected. (20 words maximum each)



0 1 2 3 4

1. $\bar{CS}$

?

~~PICs always think CPU is communicating to~~ it
~~and may break the data bus~~
Secondary PIC and Primary PIC may
read the same ICWs during initialization.
causing wrong initialization.

2. the **lowest** bit in **CAS**:

100

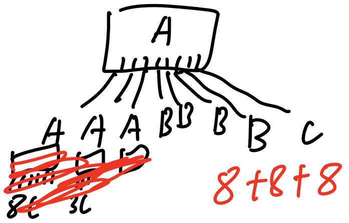
~~works properly as CAS works by setting~~
~~itself to 100, not be influenced~~
If IRQ5 of primary CAS is raised, the CAS of secondary
PIC also receive 100 and write to data bus, breaking data bus

3. $\bar{SP}$

PIC can not work as all PIC thinks
it is primary PIC and race condition
to write to the data bus

Part B (3 points): Ben found a basket of old 8259A PIC chips in the basement. Some of them are only partially working with broken IR pins (Interrupt Request pins). What's the maximum number of devices we can handle, with a cascade scheme, if we have 4 8259A PIC chips with A 8 interrupt ports fully functional, 9 chips with 4 interrupt ports B functional and 11 chips with only 2 interrupt ports C functional? Write down your calculation if applicable.

Show your work:



24+20

24+8+2

$8+8+8+4\cdot4+\overset{4}{\cancel{8}}=34$

$8\times2+(3\times2+5)+8+4+4+4+4+2$

~~16+11+8+16+2~~

~~32+11+10~~

Maximum Number: ~~53~~ ~~32~~ 44

43+10

Part C (3 points): After the calculation, Ben found that he had more devices than the maximum number of devices that his PICs can support. He had an observation: the devices he had all come in pairs, such that both devices in every pair **always send interrupts at the exact same time**. With this observation, he came up with an idea - solder the interrupt pins of each pair of devices to the inputs of an AND gate and connect the output of the AND gate to the same IRQ pin on one of the PICs in the cascade layout.

Given that **both devices in every pair always send interrupts at the exact same time**, will Ben's novel scheme work?

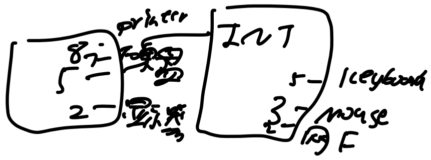
CIRCLE ONE: **Yes** [✓ **No**]

If yes, can you find a way to generalize this idea to support even more devices? e.g., double the number of devices Ben has such that each device can be put in a foursome which always interrupt at the same time within the group of four. If no, explain the reason and include the key signal/mechanism which makes it impossible.

Because two devices in a pair may have different functinality and may want to write different data to data bus, ⇒ don't work

Part D (6 points): Two PICs are put in a cascade setup, both operating in the fixed priority mode (IR0-IR7, high to low priorities), with the slave connected to the IRQ pin 5 on the master. The table below shows how a list of devices are connected to the PICs.

Mouse	IR3 on Slave
Keyboard	IR5 on Slave
Printer	IR8 on Master
Network Card	IR2 on Slave
Hard Drive	IR7 on Master
Monitor	IR2 on Master



At the following timestamps, the devices send interrupts to the system.

Mouse	40ms
Keyboard	55ms
Printer	25ms 被打断 10ms
Network Card	120ms
Hard Drive	35ms 5ms
Monitor	120ms

Assuming each interrupt request takes exactly 20ms to handle, please fill in the table below for the timestamp when the interrupt handling from each device will be finished.

Your answer:

Mouse	60ms
Keyboard	80ms
Printer	105ms
Network Card	160ms
Hard Drive	45ms
Monitor	140ms

Part of the Linux Kernel Synchronization API
Tear off this page, but return it with your exam.

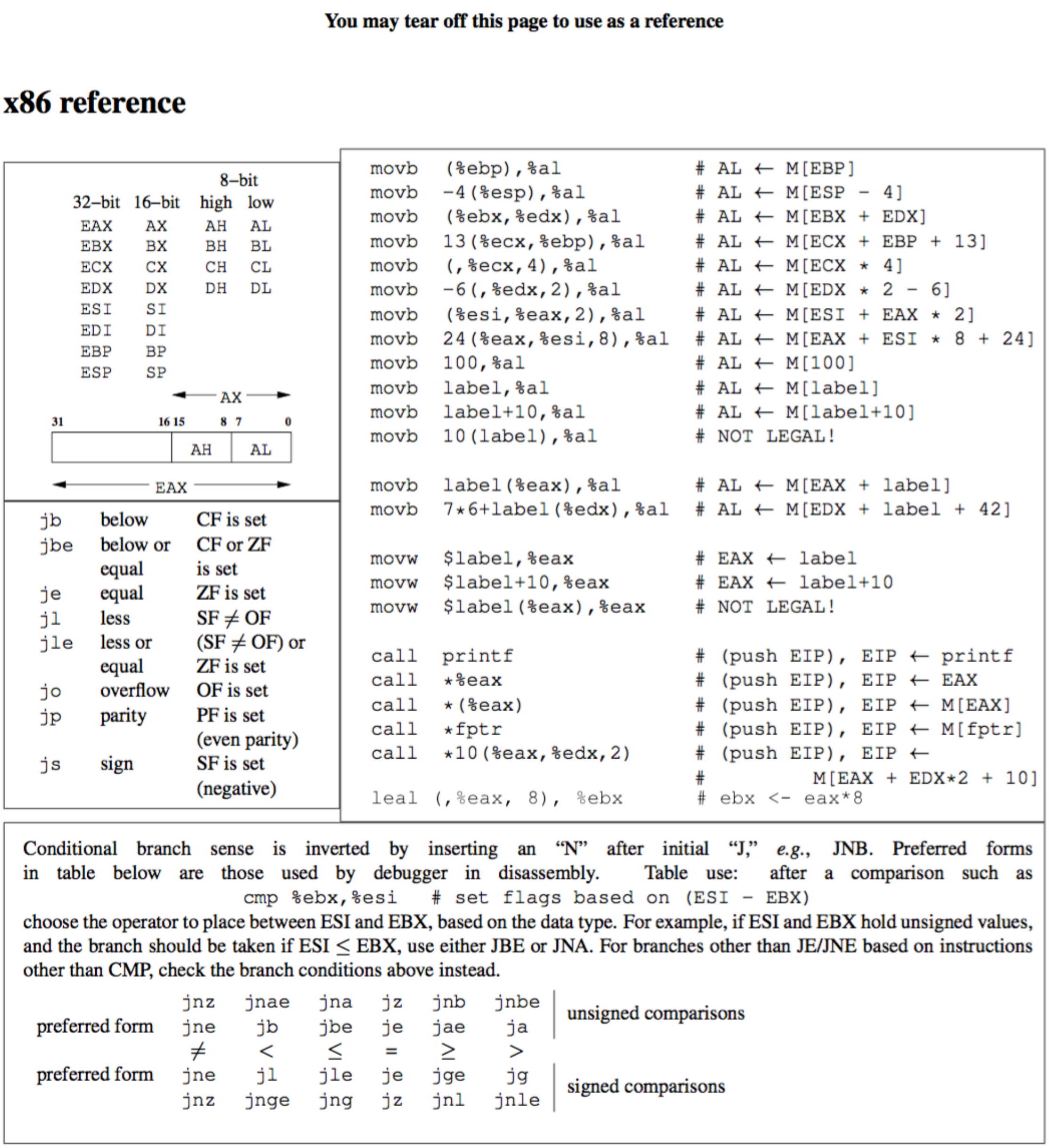
```
void spin_lock (spinlock_t* lock);

void spin_lock_irq (spinlock_t* lock);

void spin_lock_irqsave (spinlock_t* lock, unsigned long& flags);


void spin_unlock (spinlock_t* lock);

void spin_unlock_irq (spinlock_t* lock);

void spin_unlock_irqrestore (spinlock_t* lock, unsigned long flags);


void down (struct semaphore* sem);

void up (struct semaphore* sem);


void read_lock (rwlock_t* rw);

void read_lock_irq (rwlock_t* rw);

void read_lock_irqsave (rwlock_t* rw, unsigned long& flags);


void read_unlock (rwlock_t* rw);

void read_unlock_irq (rwlock_t* rw);

void read_unlock_irqrestore (rwlock_t* rw, unsigned long flags);


void write_lock (rwlock_t* rw);

void write_lock_irq (rwlock_t* rw);

void write_lock_irqsave (rwlock_t* rw, unsigned long& flags);


void write_unlock (rwlock_t* rw);

void write_unlock_irq (rwlock_t* rw);

void write_unlock_irqrestore (rwlock_t* rw, unsigned long flags);


void down_read (struct rw_semaphore* sem);

void down_write (struct rw_semaphore* sem);

void up_read (struct rw_semaphore* sem);

void up_write (struct rw_semaphore* sem);
```

Data Structure and Code for Question3

```
#define ITERATIONS
#define K 10
#define N 200

typedef struct __attribute__((__packed__)) point_t {
    uint8_t data;
} point_t;

point_t data[N];
point_t means[K];
int32_t assignments[N];
```

```
void assign_cluster() {
   int32_t i, j;
   int32_t min_dist, dist;

   for (i = 0; i < N; i++) {
       min_dist = 2147483647;  # 0x7FFF FFFF

       for (j = 0; j < K; j++) {
           dist = distance(data + i, means + j);
           if (dist < min_dist) {
               assignments[i] = j;
               min_dist = dist;
          }
       }
   }
}

void update_means() {
   int32_t i;
   uint32_t j, mean_data, count;

   for (i = 0; i < K; i++) {
       mean_data = 0;
       count = 0;

       for (j = 0; j < N; j++) {
           if (assignments[j] == i) {
               mean_data += data[j].data;
               count += 1;
            }
       }
       // Assume the below quotient is a 8-bit integer.
       means[i].data = (uint8_t) (mean_data / count);
   }
}
```

Figure 1: x86 reference. You must return this sheet with your exam.



You may tear off this page to use as a reference

x86 reference

32–bit	16–bit	8–bit high	8–bit low
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL
ESI	SI		
EDI	DI		
EBP	BP		
ESP	SP		

← AX →

31 16 15 8 7 0

AH AL

← EAX →

jb	below	CF is set
jbe	below or equal	CF or ZF is set
je	equal	ZF is set
jl	less	SF ≠ OF
jle	less or equal	(SF ≠ OF) or ZF is set
jo	overflow	OF is set
jp	parity	PF is set (even parity)
js	sign	SF is set (negative)

```
movb  (%ebp),%al             # AL ← M[EBP]
movb  -4(%esp),%al           # AL ← M[ESP - 4]
movb  (%ebx,%edx),%al        # AL ← M[EBX + EDX]
movb  13(%ecx,%ebp),%al      # AL ← M[ECX + EBP + 13]
movb  (,%ecx,4),%al          # AL ← M[ECX * 4]
movb  -6(,%edx,2),%al        # AL ← M[EDX * 2 - 6]
movb  (%esi,%eax,2),%al      # AL ← M[ESI + EAX * 2]
movb  24(%eax,%esi,8),%al    # AL ← M[EAX + ESI * 8 + 24]
movb  100,%al                # AL ← M[100]
movb  label,%al              # AL ← M[label]
movb  label+10,%al           # AL ← M[label+10]
movb  10(label),%al          # NOT LEGAL!

movb  label(%eax),%al        # AL ← M[EAX + label]
movb  7*6+label(%edx),%al    # AL ← M[EDX + label + 42]

movw  $label,%eax            # EAX ← label
movw  $label+10,%eax         # EAX ← label+10
movw  $label(%eax),%eax      # NOT LEGAL!

call  printf                 # (push EIP), EIP ← printf
call  *%eax                  # (push EIP), EIP ← EAX
call  *(%eax)                # (push EIP), EIP ← M[EAX]
call  *fptr                  # (push EIP), EIP ← M[fptr]
call  *10(%eax,%edx,2)       # (push EIP), EIP ←
                             #         M[EAX + EDX*2 + 10]
leal (,%eax, 8), %ebx        # ebx <- eax*8
```

Conditional branch sense is inverted by inserting an "N" after initial "J," *e.g.*, JNB. Preferred forms in table below are those used by debugger in disassembly. Table use: after a comparison such as

`cmp %ebx,%esi    # set flags based on (ESI - EBX)`

choose the operator to place between ESI and EBX, based on the data type. For example, if ESI and EBX hold unsigned values, and the branch should be taken if ESI ≤ EBX, use either JBE or JNA. For branches other than JE/JNE based on instructions other than CMP, check the branch conditions above instead.

	jnz	jnae	jna	jz	jnb	jnbe	unsigned comparisons
preferred form	jne	jb	jbe	je	jae	ja	
	≠	<	≤	=	≥	>	
preferred form	jne	jl	jle	je	jge	jg	signed comparisons
	jnz	jnge	jng	jz	jnl	jnle	