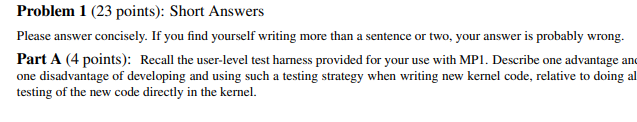
**+ Spring 2012**

****

1.a answer

One advantage of using a test harness is development time. Using a test harness makes testing quicker than having to build and install into the kernel. Testing on the kernel also comes with the risk of crashing the kernel and corrupting it. One disadvantage of using a test harness is that it doeerfectly emulate the kernel so although functionality of your code may appear to work in the test harness it may not perform as expected in the kernel.

****

1.b answer

This answer seemed to be endorsed by an instructor on Piazza, and so here goes - If what was described happens, then ICW2 (initialization control word 2) was given wrong ( probably given as OUTB(x0, 0x21) instead of OUTB (0x20, 0x21)) and this would have mapped the PIC to 0x00- 0x07 in the IDT, which is the region where ISA specific interrupts reside. To fix it, you would redo the initialization sequence, but with the correct ICW2.

~thank you but how the F were we supposed to know that



1.c answer

When an interrupt is made the state of caller saved registers needs to be saved because interrupts may occur between any instructions. In this sense, interrupts do not have call procedures, they happen abruptly and caller saved registers need to be saved by the interrupt handler to preserve their values.



1.d answer

Tasklets allow for deferrable functions, where interrupts can be scheduled in a work queue and taken care of at a later time. Hardware interrupts on the other hand are not deferrable and often reserved for critical tasks that need priority. (If someone has anymore insight please expand on this, this answer can be more detailed)

Maybe more detailed answer:

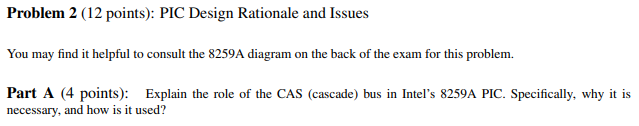
Interrupt handlers should be as short as possible to ensure the swift handling of other interrupts. For non-urgent needs, a tasklet is better because it is itself interruptible, but will still run every time an interrupt occurs.

Tasklets allow for programs that take a lot of resources (like I/O) to be deferred until critical and time sensitive tasks are completed. Doing so also improves the responsiveness of the system. Implementing in hardware would not allow for this flexibility.



1.e answer

Multiple processors have their own corresponding IF. Calling CLI will only clear the IF flag for that one processor, not for any of the others. Calling CLI/STI for all the processors is too time consuming and ruins the performance of the processors. In addition to CLI/STI, we must use locks to preserve shared memory and resources when multiple processes are going on at the same time as multiple processors could be trying to use the same resource.



2.a answer

The cascade bus allows us to string together multiple PIC’s in a master-slave relationship. It enables us to increase the number of device interrupts we can address.

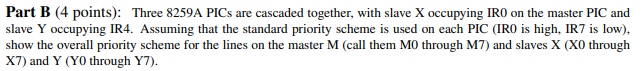
**In more detail:** (this is mostly to understand, you prolly need a shorter answer in the test)

When an interrupt occurs on one of the slave PICs (refered to as sPIC hereforth), the sPIC knows everything about the device generating the interrupt (specifically its interrupt vector), but the mPIC (master PIC) does not. However, the sPIC does not know what’s going on with the mPIC (maybe the mPIC is dealing with a higher priority interrupt), so it does not know whether or not it it is its turn to send the interrupt vector.

Basically mPIC knows when the sPIC can go, but it doesn’t know it’s vector, and the sPIC knows the vector but doesn’t know when to go.

So we need a way for the sPIC and mPIC to talk to another, and this is done through the CAS bus. The way the CAS bus works is as follows:

When the mPIC is ready to service the sPIC’s interrupt, it puts the 3-bit ID associated with the sPIC on the CAS bus (usually this ID is just the IR port number the slave is connected to). When the sPIC sees its number on the CAS bus, that’s its go signal, and it puts its vector onto the Data bus. Now we get the correct vector, and at the correct time. Problem solved :)

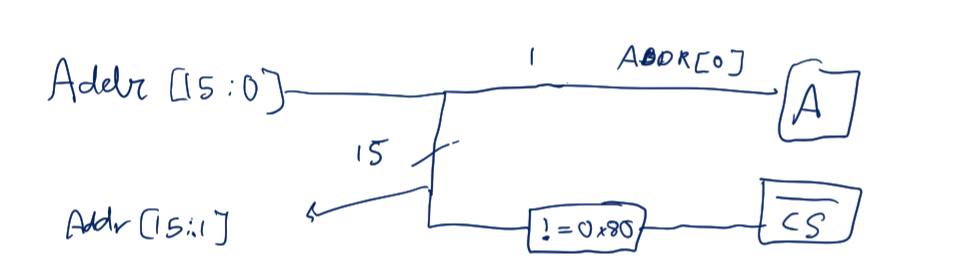


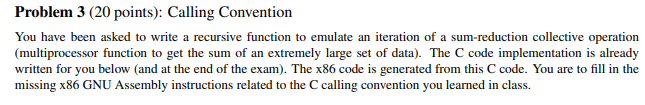
So since here we have slave X attached to master’s IR0 and slave Y attached to master’s IR4, is it just X0, X1, X2, X3, X4, X5, X6, X7, M1, M2, M3, Y0, Y1, Y2, Y3, Y4, Y5, Y6, Y7, M5, M6, M7 -- where M0 and M4 cascade to slaves X and Y respectively? (I’m not sure about this…)

I think your answer is correct. Yeah me too.



x100=0001 0000 0000, if we shift everything 1 to the right (because we’re ignoring bit 0) we get 0000 1000 0000 which is 0x80 in hexadecimal. It’s != because CS is active low.





recursive\_reduce\_iter:

pushl %ebp

movl %esp, %ebp

*# ADD YOUR CODE HERE TO...*

*# blocksize => ecx*

*# start => eax*

*# new\_list\_len => local var on stack*

movl 8(%ebp), %ecx # move blocksize into ecx

movl 12(%ebp), %eax # move start into eax

subl $4, %esp # allocate space on stack for new\_list\_len

*# ###############################*

movl %eax, %edx

cmpl $0, %eax

je base\_case

block\_loop:

cmpl $0, %ecx

je next\_block

movl NEXT(%edx), %edx

decl %ecx

jmp block\_loop

next\_block:

movl NEXT(%edx), %edx

*# ADD YOUR CODE HERE TO...*

*# Do recursive call*

*# followed by sum\_nodes call*

# Caller-save eax and edx

pushl %eax

pushl %edx

# push arguments for recursive call

pushl %edx

pushl %ecx # this is weird ecx must be zero by that time

call recursive\_reduce\_iter

# pop function arguments from the stack

addl $8, %esp

movl %eax, %esi

popl %edx

popl %eax

pushl %eax

pushl %edx

pushl %edx

pushl %eax

call sum\_nodes

addl $8, %esp

movl %eax, %ecx

popl %edx

popl %eax

*# ###############################*

movl %ecx, VALUE(%eax)

movl %edx, NEXT(%eax)

*# ADD YOUR CODE HERE TO....*

*# Set return value to new\_list\_len+1*

*# leal 1(%esi), %eax*

movl %esi, %eax

addl $1, %eax

done:

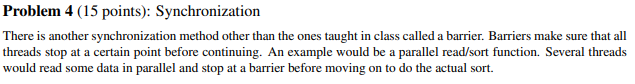
leave

ret

base\_case:

movl $0, %eax

jmp done







typedef struct {

spinlock\_t lock;

volatile int num\_t\_waiting;

} barrier\_t;

void barrier\_init(barrier\_t \*b) {

If (b == NULL) return;

spin\_lock\_init(&(b->lock));

}

void barrier\_wait(barrier\_t \*b) {

if (b == NULL) return;

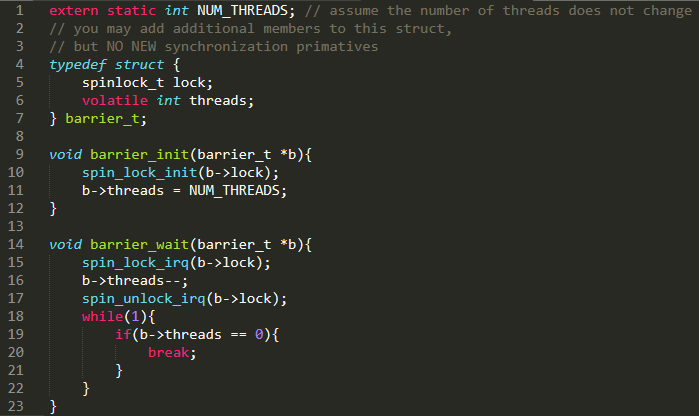
spin\_lock(&(b->lock));

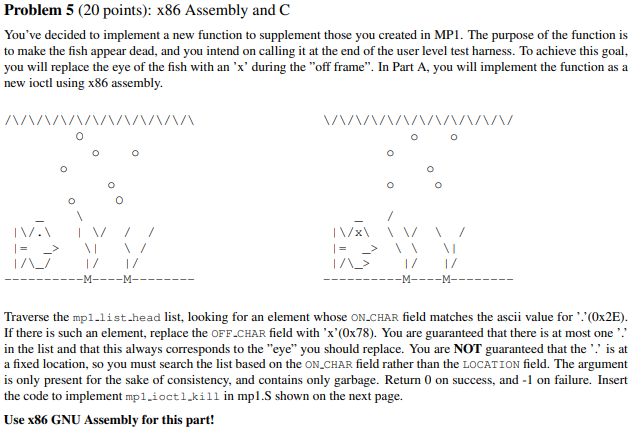
b->num\_t\_waiting++;

spin\_unlock(&(b->lock));

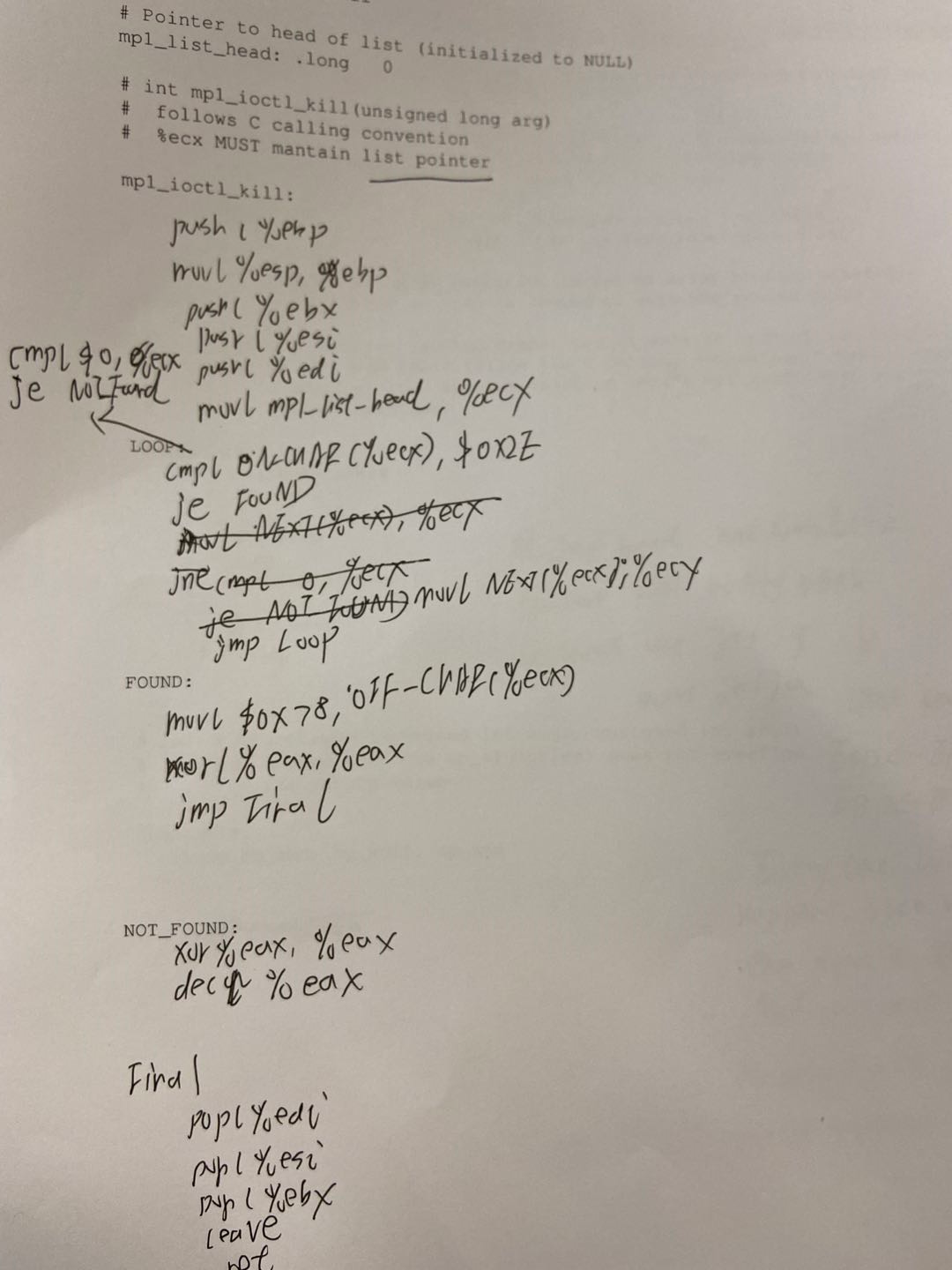
while (b->num\_t\_waiting != NUM\_THREADS) ; //Once the barrier count is reached and the thread is in a non-blocking state, the barrier can be reused.

}





I provide a piece of solution.



mp1\_ioctl\_kill:

pushl %ebp

movl %esp, %ebp

pushl %esi

pushl %edi

pushl %ebx

movl mp1\_list\_head, %ecx

LOOP:

cmpl $0, $ecx

je NOT\_FOUND

xorl %esi, %esi # clear esi

movb ON\_CHAR(%ecx), %si

cmpb $0x2E, %si

je FOUND

movl NEXT(%ecx), %ecx

jmp LOOP

FOUND:

movb $0x78, OFF\_CHAR(%ecx)

movl $0, %eax

popl %ebx

popl %edi

popl %esi

leave

ret

NOT\_FOUND:

movl $-1, %eax

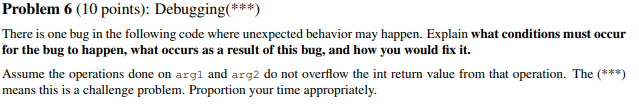
popl %ebx

popl %edi

popl %esi

leave

ret



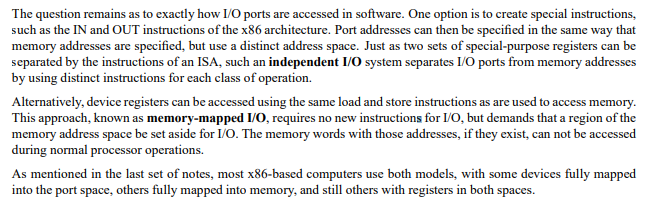
When the operation input is invalid, the function will attempt to leave (move the ebp) in this dispatcher function. Since this is a dispatcher function, the stack isn’t changed to include the old ebp. Including leave would mess up the stack causing the rest of the program to fail.

I think the problem comes from the “leave” operation. If we meet some error situation, eg. ecx=3, we will modify the value of esp and value of ebp. However, firstly we cannot modify the stack, we do not know where the ebp points to. Therefore, the ret operation will go to the wrong place.

Spring 2011

1a. One advantage of using a test harness is development time. Using a test harness makes testing quicker than having to build and install into the kernel. Testing on the kernel also comes with the risk of crashing the kernel and corrupting it. One disadvantage of using a test harness is that it does not perfectly emulate the kernel so although functionality of your code may appear to work in the test harness it may not perform as expected in the kernel.

1b. Memory Mapped (VGA) & Independent IO (PICs)



1c. When accessing functions, top to bottom would correspond to left to right.

More details:

Parameters are pushed from right to left to allow for a variable number of parameters without requiring additional space for parameter counts or sentinels.

1d. Caller saved - eax, ecx, edx, eflags

Callee saved - ebx, esi, edi

1e. CLI is first then gain the spin lock. Since interrupt may occur between them, in opposite order it will lead to deadlock (lec 7)

1f. Spin\_lock\_irqsave must be used rather than spinlock\_irq or spinlock when there is the possibility that the spinlocked code segment could be interrupted by a handler that needs to acquire that same lock to access shared memory. This would cause a deadlock because that interrupt would wait forever for the interrupted segment to release the lock. Spin\_lock\_irqsave prevents this from happening because it cannot be interrupted but will still allow that interrupt to take place afterwards.

2a.

**lock:**

**.byte 0**

**spin\_lock:**

**loop:**

**cmpl $0, lock**

**jne loop**

**bts $0, lock**

**ret**

**spin\_unlock:**

**movl $0, lock**

**ret**

2b. X = 2 in all cases. Y = 1 or 2 because of race conditions.(wait why is Y not always 2? Are global variables only changed at the end of a function?)

Since Y++ is not inside the lock both threads could be reading and writing Y value at the same time. For example thread1 reads y = 0 then adds 1 but before writing it back thread2 reads y and it is still equal to 0 therefore thread2 will add 1 to 0 and when it writes it back y = 1.

3.

.long mp1\_list\_head

convert\_to\_judy:

pushl %ebp

movl %esp,%ebp

pushl %ebx

pushl %esi

pushl %edi

call judy\_init

movl mp1\_list\_head, %edx

xorl %ecx, %ecx

CHECK\_NEXT:

cmpl $0, %edx

je DONE\_INSERTING

movw LOCATION(%edx), %cx

*# Insert call to add\_to\_judy below*

pushl %eax

pushl %ecx

pushl %edx *# Caller Save store*

pushl %ecx

pushl %eax

pushl %edx *# Push Params*

call add\_to\_judy

addl $12, %esp *# Pop Params*

popl %edx

popl %ecx

popl %eax *# Caller Save restore*

movl NEXT(%edx), %edx

jmp CHECK\_NEXT

DONE\_INSERTING:

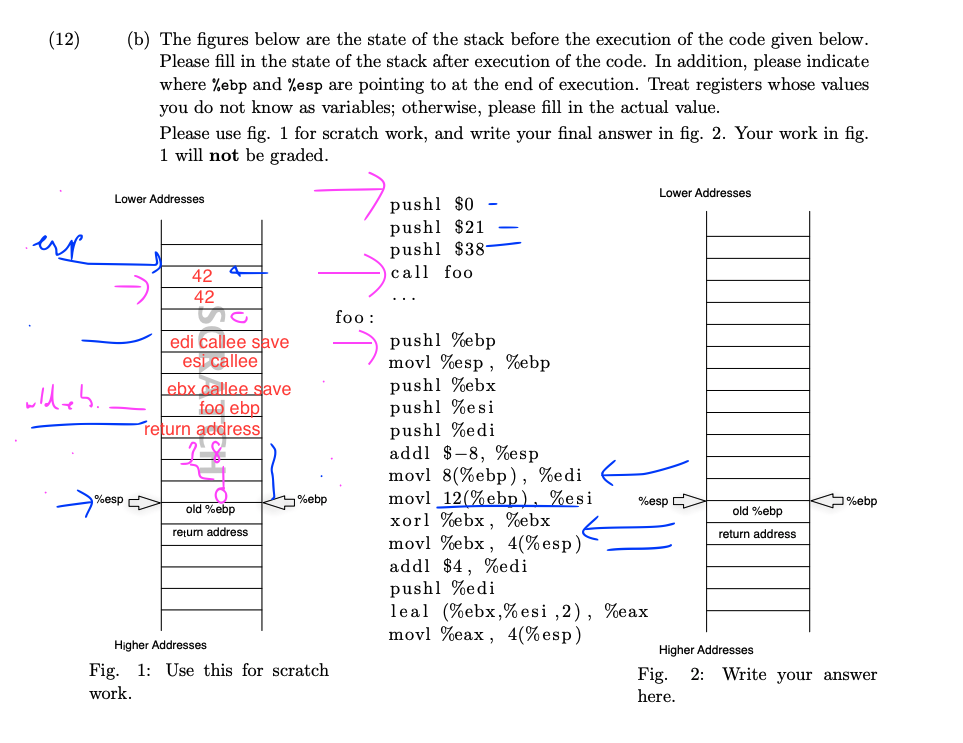
popl %edi

popl %esi

popl %ebx

leave

ret

4. 

5.

C:

ASM:

is\_palindrome:

pushl %ebp

movl %esp, %ebp

pushl %esi

pushl %ebx

pushl %edi

movl 8(%ebp), %esi *# left*

movl 12(%ebp), %edi *# right*

cmpl $0, %esi

je not\_palindrome

cmpl $0, %edi

je not\_palindrome *# null checks*

cmpl %esi, %edi

je palindrome *# base case*

movl LETTER(%esi), %ebx

cmpl LETTER(%edi), %ebx

jne not\_palindrome

movl PREV(%edi), %ebx

pushl %ebx

movl NEXT(%esi), %ebx

pushl %ebx *# make recursive call*

call is\_palindrome

addl $8, %esp

jmp final

palindrome:

movl $0, %eax

jmp final

not\_palindrome:

movl $-1, %eax

final:

popl %edi

popl %ebx

popl %esi

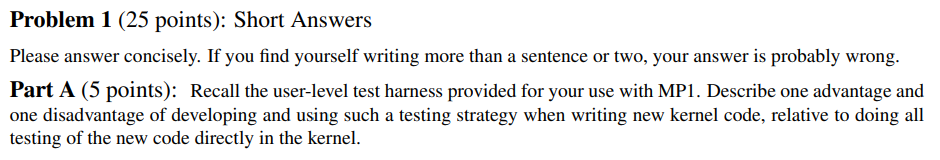
leave

ret

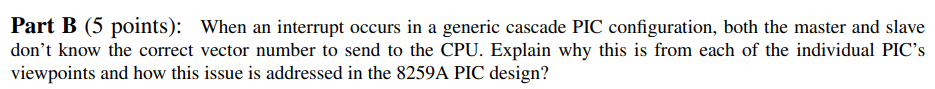


| **is\_palindrome:  //set up  pushl %ebp movl %esp, %ebp  Pushl %ebx Pushl %esi Pushl %edi  //load values   Movl 12(%ebp), %edi # right pointer Movl 8(%ebp), %esi # left pointer Cmpl %edi, %esi #base case, string of length 1 je EQUAL**  **//load first and last letter Movb (%ESI), %ECX Movb (%EDI), %EDX**  **//check if not a palindrome Cmp %Ecx, %Edx Jne NOT\_EQUAL**  **#get value to the left of right most and right of left most Movl 6(%esi), %Esi Movl 2(%edi), %EDI**  **#recursive call Pushl %edi Pushl %esi Call is\_palindrome Addl $8 %esp**  **Jmp Done**  **NOT\_EQUAL :  Movl $-1, %eax Jmp DONE  EQUAL: Movl 0, %eax Jmp DONE  DONE: Popl %EDI Popl %ESI Popl %Ebx  Leave ret** |
| --- |

Fall 2010

****

One advantage of using a test harness is development time. Using a test harness makes testing quicker than having to build and install into the kernel. Testing on the kernel also comes with the risk of crashing the kernel and corrupting it. One disadvantage of using a test harness is that it does not perfectly emulate the kernel so although functionality of your code may appear to work in the test harness it may not perform as expected in the kernel.

****

When the processor sends an active low INTA signal, it gets sent to both of the PICs. From their perspective, it means they should both write onto the data bus, and this will cause issues.

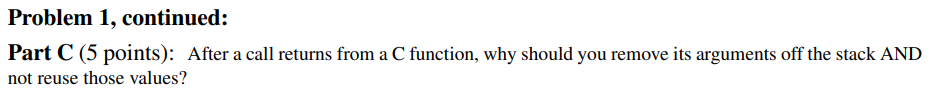
In more detail: (this is mostly to understand, you need a shorter answer in the test)

When an interrupt occurs on one of the slave PICs (refered to as sPIC hereforth), the sPIC knows everything about the device generating the interrupt (specifically its interrupt vector), but the mPIC (master PIC) does not. However, the sPIC does not know what’s going on with the mPIC (maybe the mPIC is dealing with a higher priority interrupt), so it does not know wether or not it it is its turn to send the interrupt vector.

Basically mPIC knows when the sPIC can go, but it doesn’t know it’s vector, and the sPIC knows the vector but doesn’t know when to go.

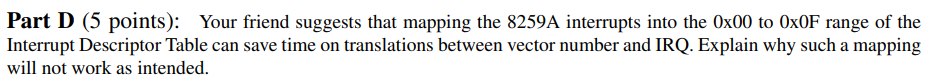
So we need a way for the sPIC and mPIC to talk to another, and this is done through the CAS bus. The way the CAS bus works is as follows:

When the mPIC is ready to service the sPIC’s interrupt, it puts the 3-bit ID associated with the sPIC on the CAS bus (usually this ID is just the IR port number the slave is connected to). When the sPIC sees its number on the CAS bus, that’s its go signal, and it puts its vector onto the Data bus. Now we get the correct vector, and at the correct time. Problem solved :)

****

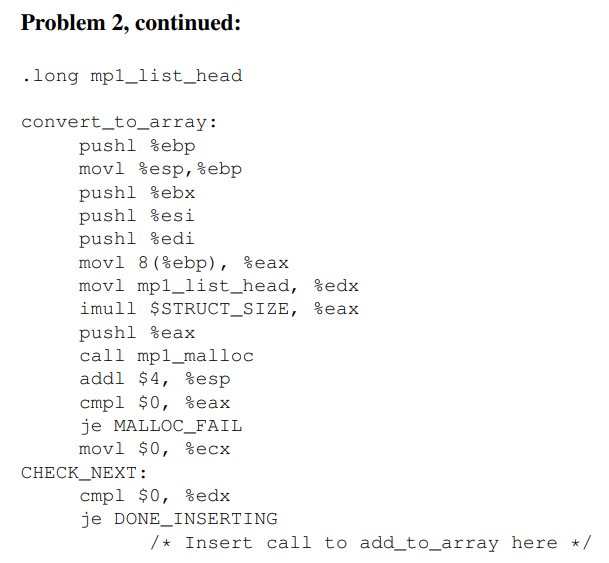
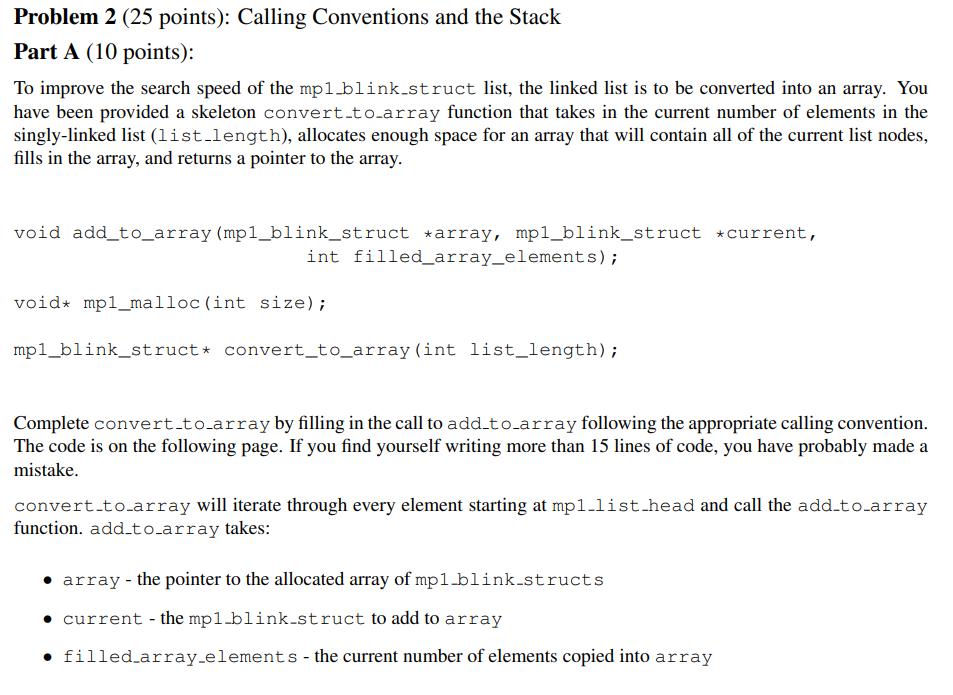
Not sure of the first part. For not reusing, I think you cannot re-use because it is possible that the values you pushed were modified by the function (maybe like a reference variable/parameter?)

I think it may be the variable has been changed, like call-by-value, but they are just copy from callee function scope, so when referred to them when calling back, the code should track the variable in callee scope.

****

See SP2012 1B. Basically the same question.

One additional point I think is that IDT is hardware defined, which we should/could not modify.

****

pushl %edx # save caller-saved registers

pushfl

pushl %ecx # filled\_array\_elements

pushl %edx # current ptr

pushl %eax # array ptr

call add\_to\_array

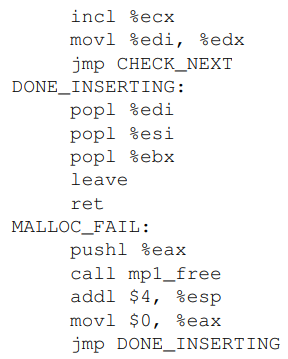
addl $12, %esp # pop arguments off stack

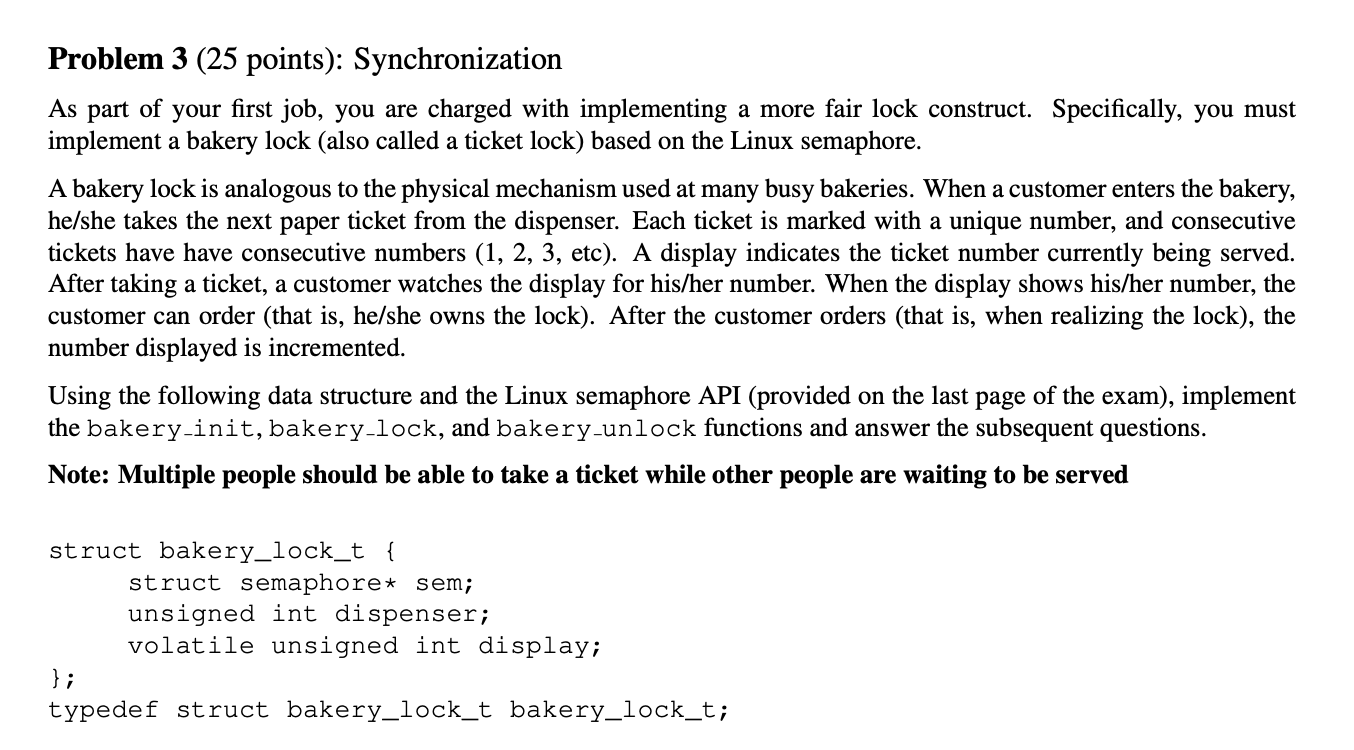
popfl # restore caller-saved registers

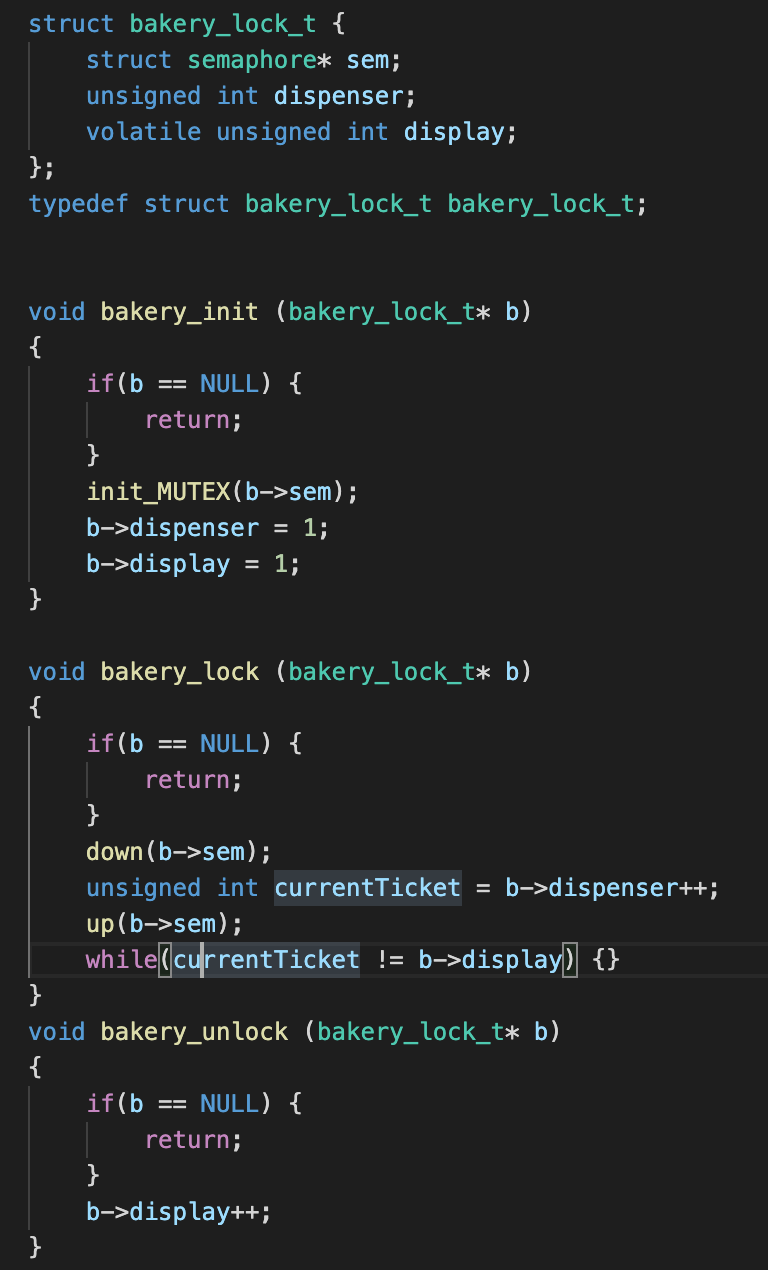
popl %edx

movl %eax, %edi

# where’s %edi used?

****





**Part D**

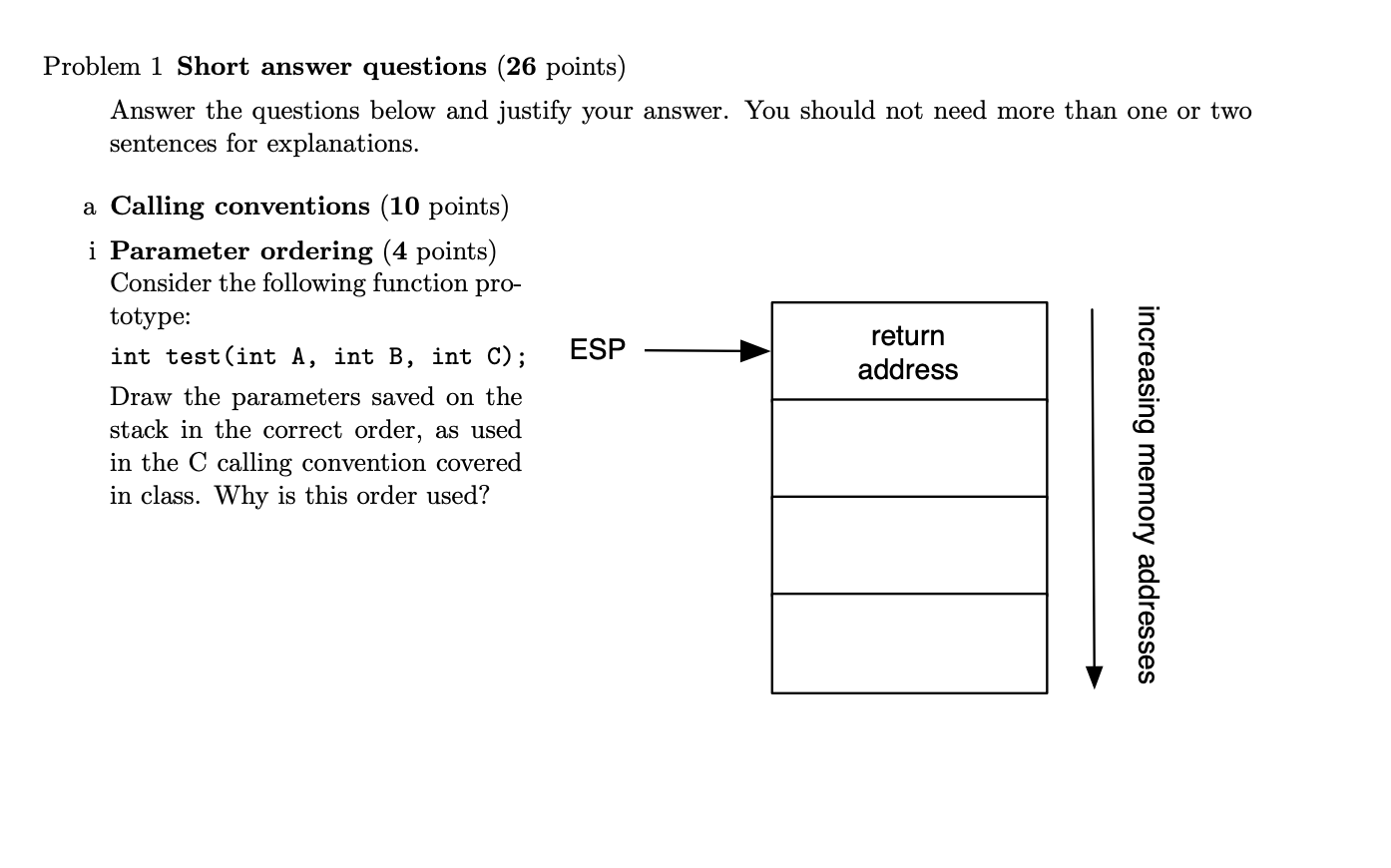
If there was a lock on the display, you would run into issues with having multiple users watching the display at once. If there was an issue where 2 people get a ticket at the same time but person number 2 gets access to the display first you would have a deadlock because person never gets to check the display so they can never go and both people wait forever.

Putting a lock on being able to increment the display is silly because since only one person ever has access to the baker\_lock at a time, they should never unlock it until after they gained access to the lock. It is unnecessary to restrict being able to unlock the bakery lock in this way.

**Part E**

I think it may be an issue, especially if other threads are holding a semaphore that the current owner of the bakery lock needs. If a paused thread that holds a semaphore is waiting for the bakery lock, then the current owner of the bakery lock will have to wait for that semaphore forever, i.e. Deadlock.

Spring 2010

****

**ESP -> Return Addy**

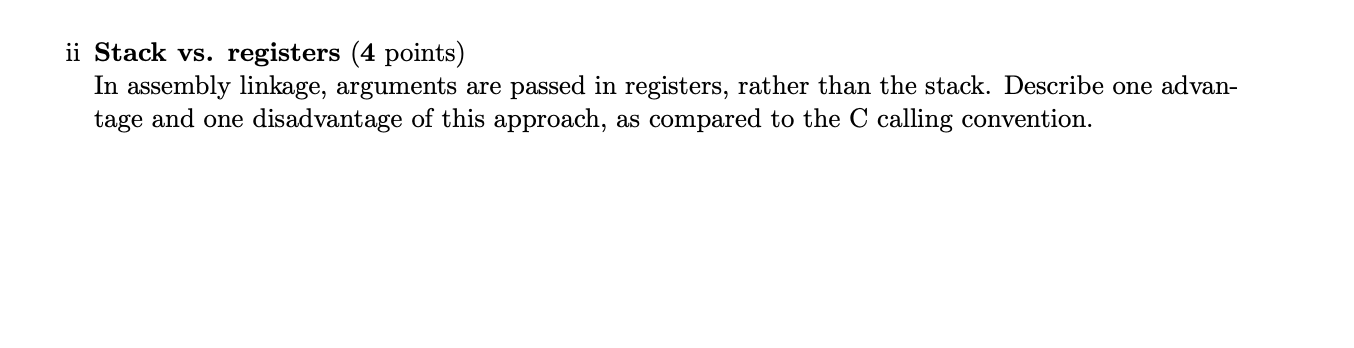
**A**

**B**

**C**

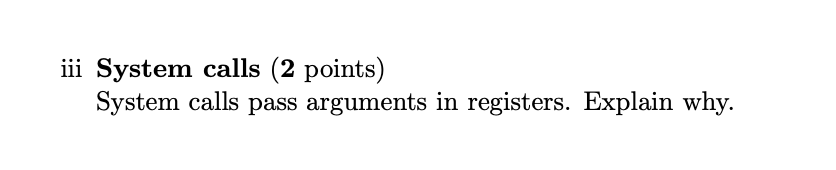
**C calling convention requires arguments to be passed into the stack from right to left. This is to ensure accessibility to arguments in reference to ESP/EBP. (Someone double check me on this reasoning, I don’t think I am right). We push parameters right to left to allow for a variable number of parameters without needing extra space for counts**

**^I believe your reasoning is correct**

****

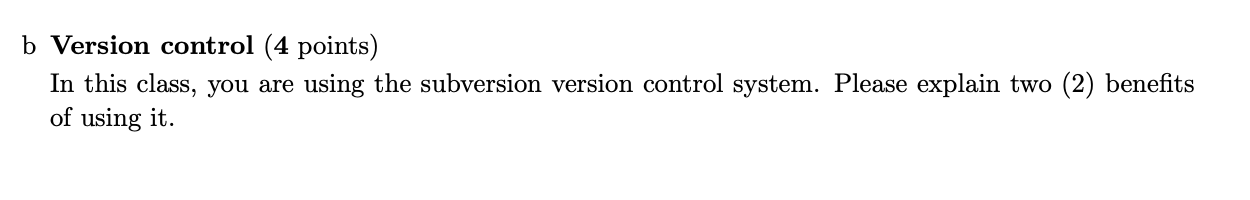
**The advantage of using registers is you don’t use up memory in the stack to store arguments. However, there is a limit to the number of registers you can use, making it difficult when you are using multiple arguments.**

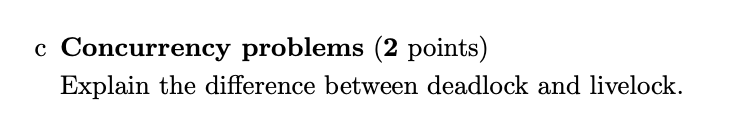
Also it is faster to do register operations than to read from memory.

****

**System calls use kernel stack when programs use user stack. To communicate between the two, need to use registers (Not sure)**

**System call is in kernel space, and has separate different stack, it could not use stack args directly, if desired, it needs to transfer in between, which should be like, user stack -> reg -> kernel stack, complicated and waste of time**

****

****

“a deadlock is a state in which each member of a group waits for another member, including itself, to take action, such as sending a message or more commonly releasing a [lock](https://en.wikipedia.org/wiki/Lock_(computer_science))”

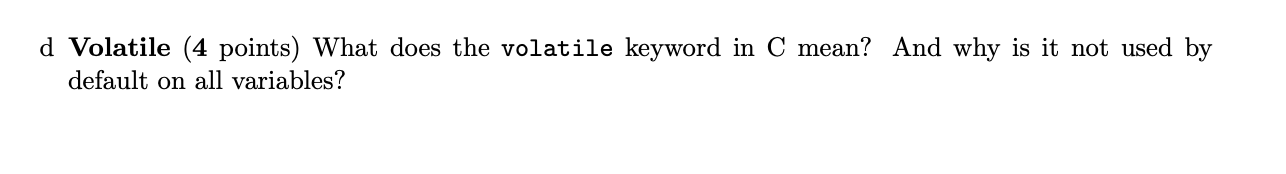
“A livelock is similar to a deadlock, except that the states of the processes involved in the livelock constantly change with regard to one another, none progressing. Livelock is a special case of resource starvation; the general definition only states that a specific process is not progressing.”

To put it into a more concise explanation: Livelock is a special version of a deadlock, in both cases processes experience resource starvation although a deadlock is specific to individual processes being barred from resources. Whereas in a livelock multiple processes are starving because they bar resources from each other and end up in a state where neither process can access resources.

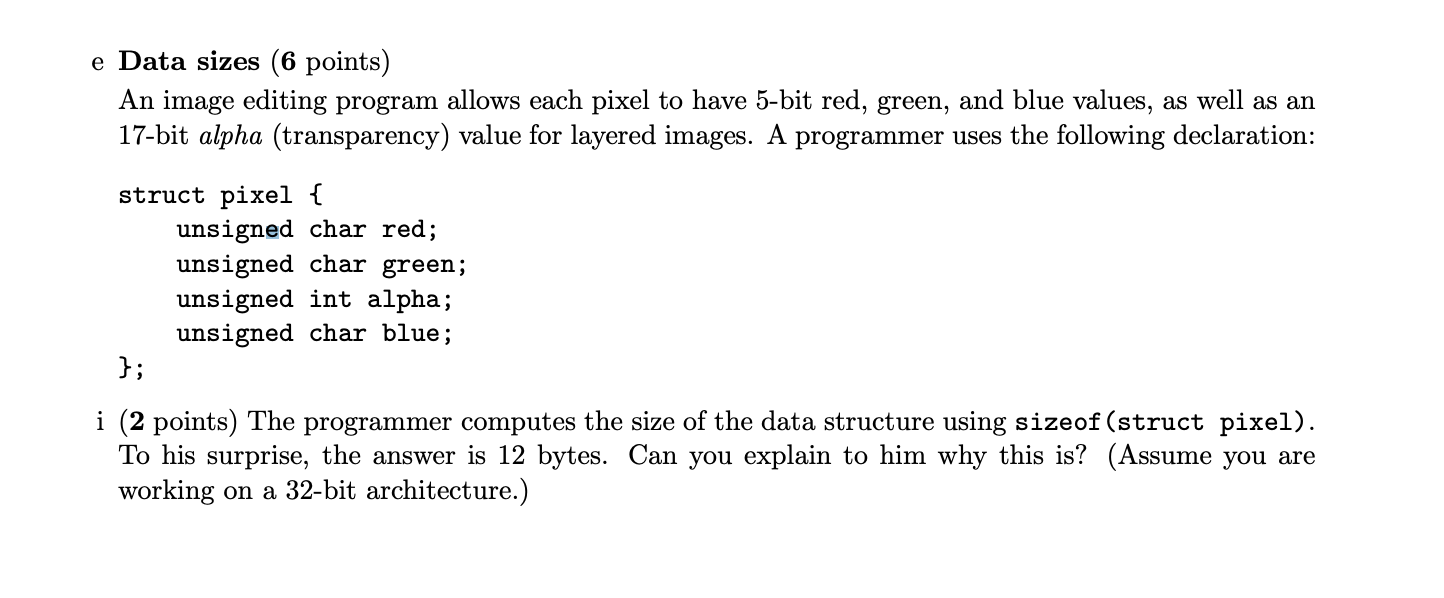
An example of this is:

1. Process A holds resource Y
2. Process B holds resource X
3. Process A requires resource X
4. Process B requires resource Y

This can happen if at first, process A runs and acquires data resource X and then process B runs and acquires resource Y. Then no matter which process runs first, none of them further progress. You fix this by correctly following the principles of conservative sync design.

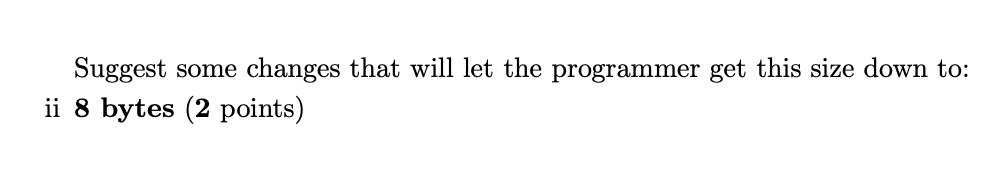


**Makes it so the compiler will always check on the volatile variable on each run. Not used by default because checking everything everytime is too much time. Compiler is optimized for speed, so if it sees the variable cannot change in the scope of the code it no longer checks it, volatile prevents this.**

****

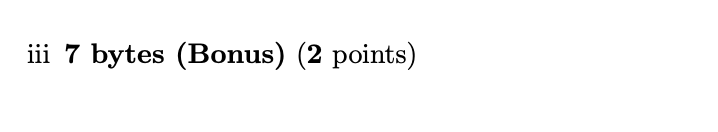
| **red** | **green** | **P** | **P** | **alpha** | **alpha** | **alpha** | **alpha** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **blue** | **P** | **P** | **P** |  |  |  |  |

**P = padding Chars = 1 Byte Int = 4 bytes (Each column represents a byte)**

****

**Switch blue and alpha in the struct:**

| **red** | **green** | **blue** | **P** | **alpha** | **alpha** | **alpha** | **alpha** |
| --- | --- | --- | --- | --- | --- | --- | --- |

****

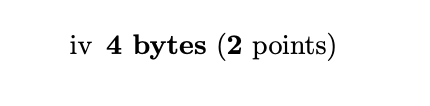
**Put Alpha at the top of the struct, and call the struct as packed.**

| **Alpha** | **Alpha** | **Alpha** | **Alpha** | **Red** | **Green** | **Blue** |  |
| --- | --- | --- | --- | --- | --- | --- | --- |

**I think the above is not enough,**

**Struct \_\_attribute\_\_((\_\_packed\_\_)) pixel {any order of 4 elements}, refer the source on piazza**

**(I basically said that when I said “packed” lol)**

****

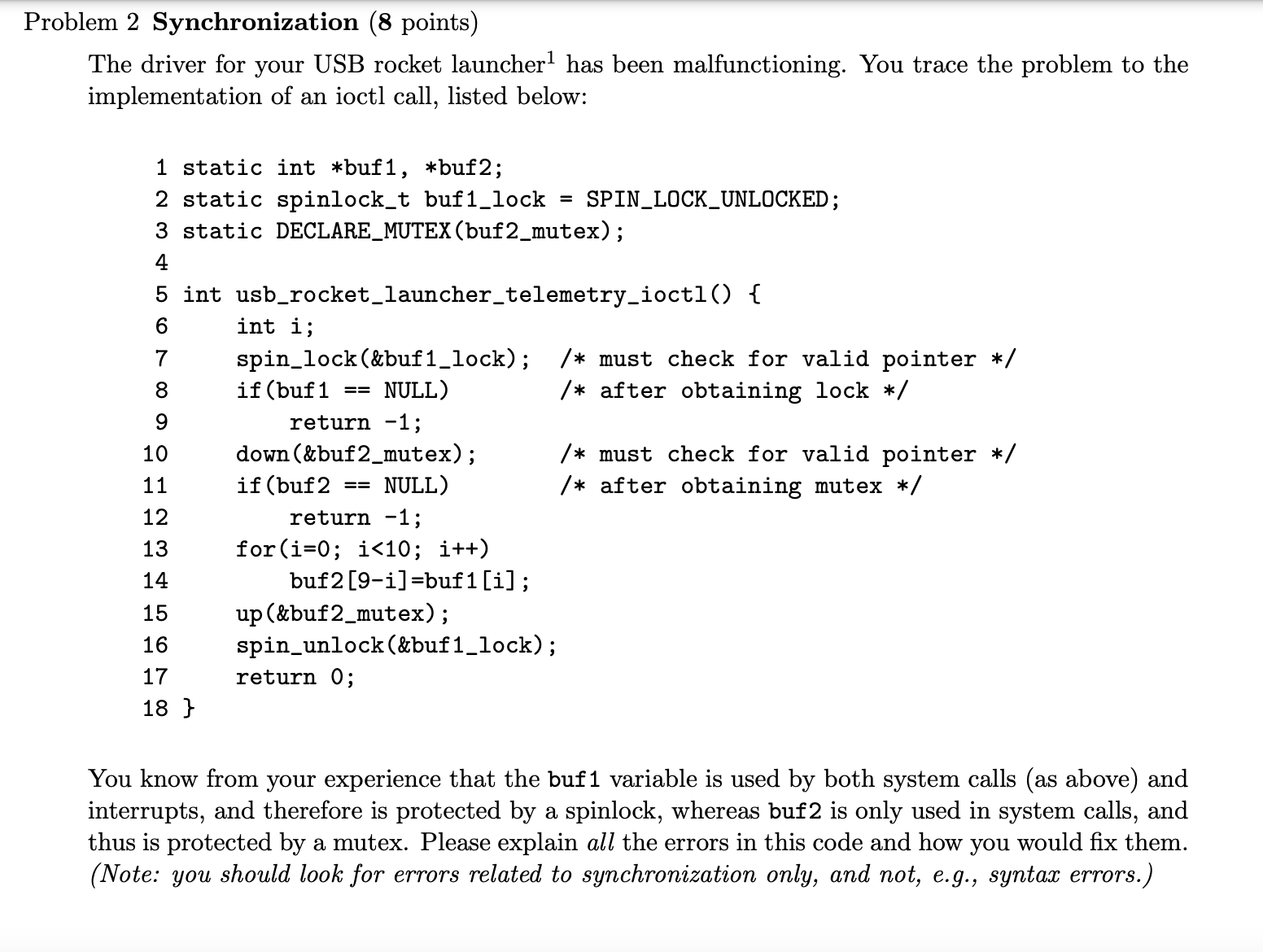
**You can pack these values as so**

**Unsigned int red:5;**

**Unsigned int green:5;**

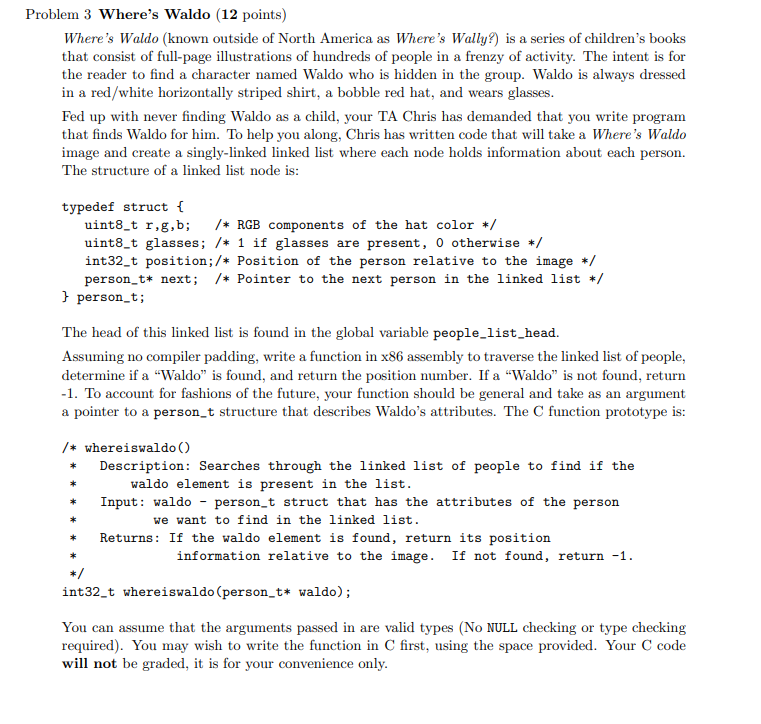
**Unsigned int alpha:17;**

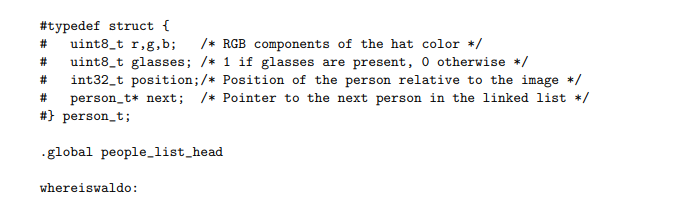
**Unsigned int blue:5;**

****

* **Line 8-9 and 11-12 will cause issues if conditions are met (returning without unlocking).**
* **Line 16 unlocks after a for loop and mutex call. This means any interrupt executed between the lock and unlock will cause a deadlock. Lock and unlock should be called inside the for loop.**
* **It would be appropriate to use spin\_lock\_irq since interrupts could take place in the code which could cause issues, specifically in lines 13-14, which could throw an exception.**
* **Should semaphore first then spin lock.**

**Im not sure if 2nd point is right can someone tell if my reasoning is wrong? If buf2 is only used by other system calls no deadlock is possible cause the other call is set to sleep until the mutex is available again. And also, isnt it better to keep it locked till all 16 writes have happened? What if another system call changed the bit of buf2 we had written a few calls ago? So it is being incremented and decremented in the correct locations right?**

****

****

**I will provide one solution**

.data

/\* Useful offset constants for accessing members of a

struct mp1\_blink\_struct structure\*/

R = 0

G = 1

B = 2

GLASSES = 3

POSITION = 4

NEXT = 8

STRUCT\_SIZE = 12

**# g:1, b:2, glasses:3, position:4,next:8**

**# g:1, b:2, glasses:3, position:4,next:8**

**.global people\_list\_head**

**pushl %ebp**

**movl %esp, %ebp**

**pushl %ebx**

**pushl %esi**

**pushl %edi**

**movl 8(%ebp),%edi # store waldo**

**xorl %ebx,%ebx**

**xorl %esi,%esi**

**xorl %edi,%edi**

**xorl %eax,%eax**

**movl people\_list\_head,%ebx**

**start\_find:**

**cmpl %ebx,$0**

**je none\_find**

**movb (%ebx),%esi**

**movb (%edi),%eax**

**cmpl %esi,%eax**

**jne ct**

**movb 1(%ebx),%esi**

**movb 1(%edi),%eax**

**cmpl %esi,%eax**

**jne ct**

**movb 2(%ebx),%esi**

**movb 2(%edi),%eax**

**cmpl %esi,%eax**

**jne ct**

**movb 3(%ebx),%esi**

**cmpl %esi,$1**

**jne ct**

**find\_wat:**

**movl 4(%ebx),%eax**

**jmp final**

**ct:**

**movl 8(%ebx),%ebx**

**jmp start\_find**

**none\_find:**

**xorl %eax, %eax**

**addl $-1,%eax**

**final:**

**popl %edi**

**popl %esi**

**popl %ebx**

**leave**

**ret**

**Another possibility (could have different issues)**

# typedef struct{

# uint8\_t r, g, b; /\*RGB components of the hat color\*/

# uint8\_t glasses; /\*one if classes are present, 0 otherwise\*/

# int32\_t position; /\*position of the person relative to the image\*/

# person\_t\* next; /\* Pointer to the next person in the linked list \*/

# } person\_t;

.data

/\* Useful offset constants for accessing members of a

struct mp1\_blink\_struct structure\*/

R = 0

G = 1

B = 2

GLASSES = 3

POSITION = 4

NEXT = 8

STRUCT\_SIZE = 12

.global people\_list\_head

/\* whereiswaldo()

Description: seatches through the linked list of people to find if the

waldo element is present in the list

Input: walso - person\_t struct that has the attributes of the person we

want to find in the linked list.

Returns: If the waldo element is found, return its positoin

information relative to the image. If not found, return -1.

\*/

/\*int32\_t whereiswaldo(person\_t\* waldo){

person\_t\* current = people\_list\_head;

while(current->next != NULL){

if(waldo->r != current->r)

break;

if(waldo->g != current->g)

break;

if(waldo->b != current->b)

break;

if(waldo->glasses != current->glasses)

break;

return current->position

}

return -1;

} \*/

whereiswaldo:

# setup stack frame

pushl %ebp

movl %esp, %ebp

# callee saved registers

pushl %ebx

pushl %esi

pushl %edi

# load the person\_t\* pointer named waldo

movl 8(%ebp), %ecx

# load head pointer into edx

movl people\_list\_head, %edx

loop:

/\* check if color is equal if not next loop \*/

movb R(%edx), %al

cmpb R(%ecx), %al

jne next

movb G(%edx), %al

cmpb G(%ecx), %al

jne next

movb B(%edx), %al

cmpb B(%ecx), %al

jne next

/\* check glasses \*/

movb GLASSES(%edx), %al

cmpb GLASSES(%ecx), %al

jne next

/\* if still in loop we found waldo \*/

movl POSITION(%edx), %eax

jmp done

next:

/\* if next is null, jump \*/

movl NEXT(%edx), %eax

cmpl $0, %eax

je eaxminusone

/\* else set to next struct and rejoin loop \*/

movl NEXT(%edx), %edx

jmp loop

eaxminusone:

xorl %eax, %eax

decl %eax

jmp done

done:

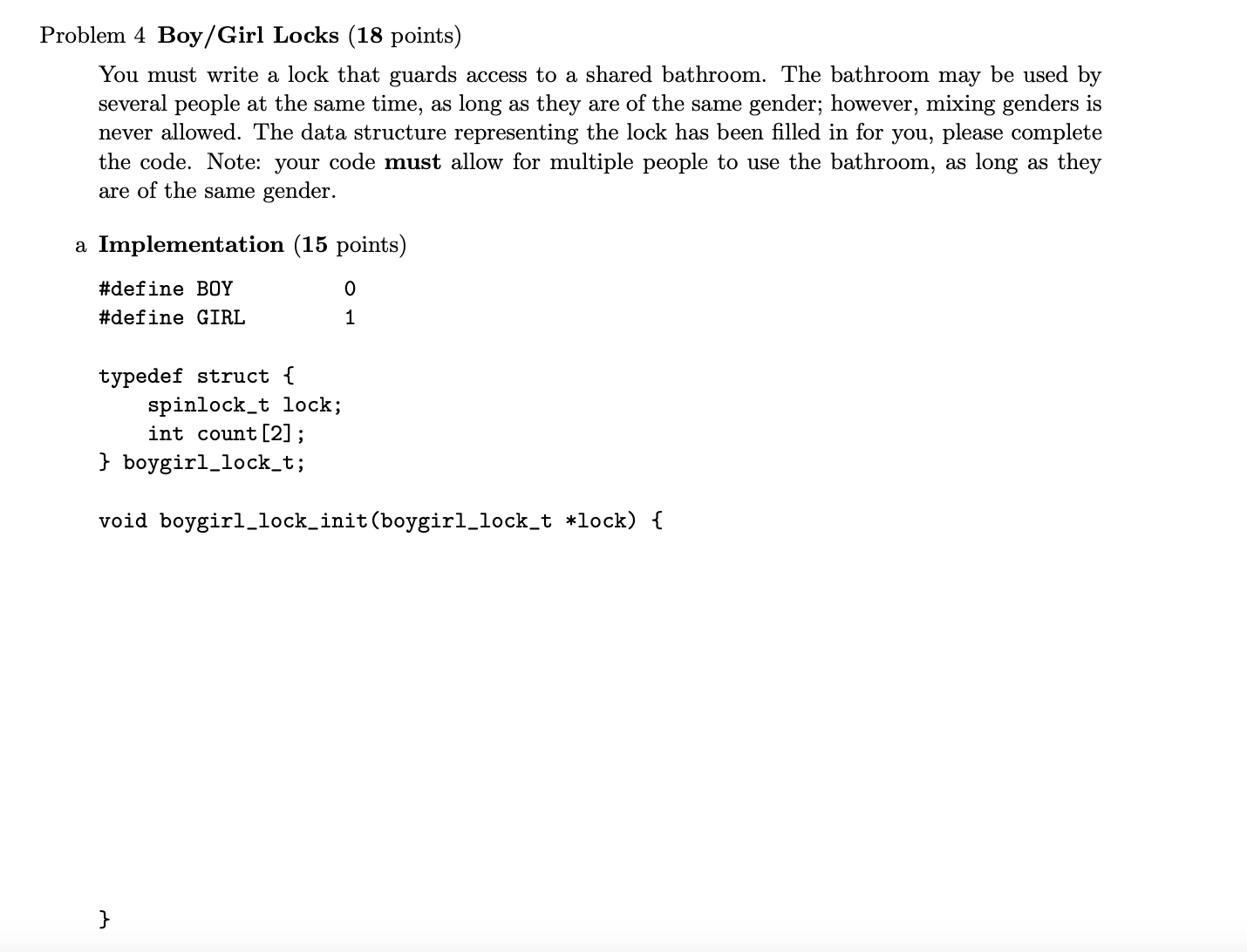
popl %edi

popl %esi

popl %ebx

leave

ret

****

**void boygirl\_lock\_init(boygirl\_lock\_t \*lock) {**

**spin\_lock\_init(lock->lock);**

**lock->count[BOY]=0;**

**lock->count[GIRL]=0;**

**} // Churan Approved Solution**

**void boygirl\_lock(boygirl\_lock\_t \*lock, int gender) {**

**while(1) {**

**spin\_lock\_irq(lock->lock);**

**if(lock->count[1-gender] == 0) {**

**lock->count[gender]++;**

**break;**

**}**

**spin\_unlock\_irq(lock->lock);**

**}**

**spin\_unlock\_irq(lock->lock);**

**} // Churan Approved Solution**

**void boygirl\_unlock(boygirl\_lock\_t \*lock, int gender) {**

**spin\_lock\_irq(lock->lock);**

**if(lock->count[1-gender] == 0) {**

**lock->count[gender]--;**

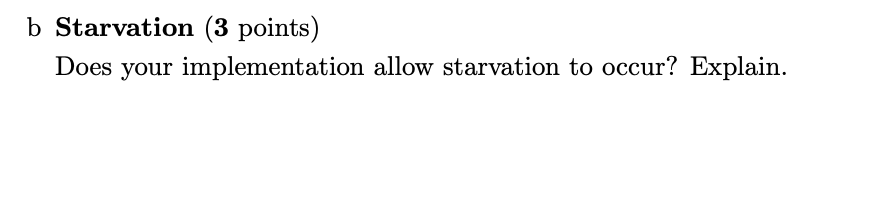
**if(lock->count[gender] < 0) {**

**lock->count[gender] = 0;**

**}**

**spin\_unlock\_irq(lock->lock);**

**} // Churan Approved Solution**

****

**My implementation allows for starvation to occur, given that if one boy enters the bathroom, and a girl come along waiting to use the restroom, the implementation allows for boys to skip ahead of this girl since there is no restriction on the number of guys that can enter this bathroom, meaning someone’s bladder is going to rupture rather than starve.**