CSE351 Spring 2014 – Final Exam (11 June 2014)

Please read through the entire examination first! We designed this exam so that it can be completed in the 110 minutes we have scheduled and, hopefully, this estimate will prove to be reasonable.

There are 6 problems for a total of 220 points. The point value of each problem is indicated in the table below and at every part of every problem. Write your answer neatly in the spaces provided. If you need more space (you shouldn't), you can write on the back of the sheet where the question is posed, but please make sure that you indicate clearly the problem to which the comments apply. Do NOT use any other paper to hand in your answers. If you have difficulty with part of a problem, move on to the next one. They are independent of each other.

The exam is CLOSED book and CLOSED notes. Please do not ask or provide anything to anyone else in the class during the exam. Make sure to ask clarification questions early so that both you and the others may benefit as much as possible from the answers.

Name: ______*Sample Solution*____

ID#: _________________________

Problem	**Max Score**	**Score**
1 (Potpourri)	30	***30***
2 (Stacks)	30	***30***
3 (Caches)	40	***40***
4 (Virtual Memory)	40	***40***
5 (Memory Allocation)	70	***70***
6 (Java)	10	***10***
TOTAL	**220**	***220***

1. Potpourri (True/False Answers) – 30pts total (2pts each)

A. A 2s-complement 2-byte integer can be copied into a 32-bit register using the movzwl instruction.

☐ True ☑ False

B. On a 64-bit architecture, casting a C long int to a double does not lose precision.

☐ True ☑ False

C. A logical shift of a 2s-complement number by 3 bits to the right (>> 3) is the same as dividing by 8.

☐ True ☑ False

D. In C, the length of string is always in an int at the starting address of the string.

☐ True ☑ False

E. In both C and Java it is possible to determine the address of a struct/object within an array of structs/objects.

☐ True ☑ False

F. Total internal fragmentation in a struct can't be more than its largest element.

☐ True ☑ False

G. An instruction cache takes advantage of both spatial and temporal locality.

☑ True ☐ False

H. To be able to write a correct program, a developer needs to know cache sizes.

☐ True ☑ False

I. Caches copy frequently used memory to faster storage to speed-up execution.

☑ True ☐ False

J. On a 32-bit architecture, if a cache block is 128 bytes, and there are 1024 sets in the cache, the tag will be 17 bits.

☐ True ☑ False

K. A process's stack is typically in a segment of memory that is not executable.

☑ True ☐ False

L. When executing a fork, a child process is given the same process ID as its parent.

☐ True ☑ False

M. A TLB is used in an MMU to cache page table entries.

☑ True ☐ False

N. A parent process and its children share the same memory address space.

☐ True ☑ False

O. C generally has better performance than Java.

☑ True ☐ False

2. Stacks – 30 pts total (5/A, 5/B, 5/C, 15/D)

You are running a program on a 64-bit architecture, that uses stack frames to hold local variables but passes arguments in registers. Assume integers are 4 bytes and pointers are 8 bytes.

The program includes the definition for a data_structure type:

```
typedef struct data_struct {
  int a;
  int *b;
  int c;
} data_struct;
```

as well as the definition of a print_structure function:

```
void print_struct(data_struct *y) {
  printf("%p\n", y);
  printf("%d\n", *(y->b + y->c));
<<execution is suspended here>>
}
```

This is a small snippet of code corresponding to `foo`, which has just been called and in turns calls `print_struct`:

```
int foo() {
  data_struct x;
  int n = 13;
  x.a = ???;
  x.b = &n;
  x.c = 3;
  print_struct(&x);
}
```

Execution is suspended after the `printf` statements in `print_struct` but before it returns to `foo`. The stack at this point of the execution of the program is shown below in 4-byte blocks (note that the stack is shown as is tradition, from bottom to top, with the top-most of the stack at the bottom or lowest address):

```
0x7fffffffffffa038:  0x00203748
0x7fffffffffffa034:  0x00000001
0x7fffffffffffa030:  0x0000015f
0x7fffffffffffa02c:  0x00000000
0x7fffffffffffa028:  0x00402741
0x7fffffffffffa024:  0x0000000d
0x7fffffffffffa020:  0x00000003
0x7fffffffffffa01c:  0x7fffffff
0x7fffffffffffa018:  0xffffa024
0x7fffffffffffa014:  0x00000007
0x7fffffffffffa010:  0x00000000
0x7fffffffffffa00c:  0x00402053
```

Note: The alignment is not quite correct in this example. a and b should both be padded for a total size of 24 bytes for the struct.

A. What is the value stored in the stack at the 8-bytes starting at location `0x7fffffffffa00c` to `0x7fffffffffa013` and what does it represent?

`0x0000000000402053` which represents the return address to be used when `print_struct` returns to `foo`.

B. What value was assigned to `x.a` in the function `foo` and at what address is it stored on the stack?

The value `0x7` represents `x.a` and is stored at location `0x7fffffffffa014`.

C. What will the call to `print_struct` output?
(Note: the "`%p`" and "`%d`" format specifiers print the value of a pointer in hex and the value of an int in decimal notation, respectively.)

```
0x7fff…ffa014
351
```

D. The argument `&x` to `print_struct` is stored in register `%rdi` when `print_struct` is called. What are succinct assembly language instructions to obtain the value printed in the second statement of `print_struct`, namely, `*(y->b + y->c)`, and place it `%rdi` for the call to `printf`?

```
movq       4(%rdi), %rax
movslq    12(%rdi), %rcx
shlq            2, %rcx  |
addq         %rcx, %rax  | or   movq  (%rax, %rcx, 4), %rdi
movq        (%rax), %rdi |
```

3. Caches – 40pts total (5/A, 5/B, 5/C, 25/D)

A. If a cache has a block size of 128 bytes, what is the miss rate we expect in a row-major sequential traversal of an array of 16-byte structs (assume we make four accesses to each struct)?

*Since our blocks are 128 bytes, there will be 8 structs in each block loaded from memory. Thus, the first access to a struct will generate a miss, the next three accesses to the same struct will be hits as will the 4 accesses to each of the next 7 structs. Thus, there will be one miss every 4*8 accesses or a miss rate of 1/32 or* ***3.125%****.*

B. How many sets are there in a 64K cache that is 4-way set associative and has a block size of 64 bytes? If the address size is 32 bits, how many bits are in the tag?

*C (size) = S (sets) * E (associativity) * B (block size)*
*64K = S * 4 * 64*
*S = 64K / (4 * 64) = 64k / 256 =* $2^{16}/2^{8} = 2^{8}$ = ***256 sets***

Address bits = tag bits + set bits + block offset bits
32 = tag bits + ($\log_2 256$) + ($\log_2 64$) *= tag bits + 8 + 6*
tag bits = 32 – 8 – 6 = ***18 bits***

C. What are the two types of locality that make caches work well? Describe each in one sentence.

Spatial locality: when one byte is accessed, bytes nearby are likely to also be accessed.

Temporal locality: a byte that is accessed is likely to be accessed again soon.

D. Given the following 2-way set-associative cache and its contents in a system with a 12-bit address:

Index	Tag	V	B0	B1	B2	B3	B4	B5	B6	B7	Tag	V	B0	B1	B2	B3	B4	B5	B6	B7
0	2F	1	99	1F	34	56	99	1F	34	56	11	1	DE	AD	BE	EF	DE	AD	BE	EF
1	2C	0	27	A4	C5	23	00	00	00	01	22	0	FF	FF	FF	FF	FF	FF	FF	FF
2	01	1	54	21	65	78	54	21	65	78	3F	0	FF	FF	FF	FF	FF	FF	FF	BF
3	0F	1	01	02	03	04	05	06	07	08	12	1	CA	FE	12	34	CA	FE	12	34
4	36	1	3E	DE	AD	0F	3E	DE	AD	0F	34	0	FF	FF	FF	F4	FF	FF	FF	FE
5	3D	0	7F	FF	FF	FF	FF	FF	FF	FF	23	1	1F	2E	11	09	1F	2E	11	09
6	23	1	12	5E	67	90	12	5E	67	90	12	0	00	00	00	01	00	00	00	02
7	13	0	00	00	00	00	00	40	20	60	0F	1	12	34	56	78	13	24	57	68

Label the bits of the address with whether they are used as a block offset (CO), set index (CI) or tag (CT).

11	10	9	8	7	6	5	4	3	2	1	0
CT	*CT*	*CT*	*CT*	*CT*	*CT*	*CI*	*CI*	*CI*	*CO*	*CO*	*CO*

What are the results of the following read operations (specify whether it is a hit or miss and the value if is determinable from the information given, otherwise just write ND for non-determinable)? Assume the reads are executed in the order given below and the addresses are given in hex.

Address to be read	Tag	Set	Byte	Hit or Miss (H or M)	Value read (or ND)
0xBC4	*10 1111*	*000*	*100*	*H*	*0x99*
0x498	*01 0010*	*011*	*000*	*H*	*CA*
0x358	*00 1101*	*011*	*000*	*M*	*ND*
0x398	*00 1110*	*011*	*000*	*M*	*ND*
0x498	*01 0010*	*011*	*000*	*M*	*ND*
0x4FD	*01 0011*	*111*	*101*	*M*	*ND*
0x8EA	*10 0011*	*101*	*010*	*H*	*0x11*

4. Virtual Memory – 40pts total (5/A, 5/B, 5/C, 25/D)

We have a system with the following properties:

- a virtual address of 16 bits (4 hex digits),
- a physical address of 11 bits (3 hex digits),
- pages that are 128 bytes,
- a corresponding page table with 512 entries, and
- a TLB with 16 entries that is 4-way set associative.

The current contents of the TLB and Page Table are shown below:

TLB

Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	V
0	03	-	0	07	0	1	06	-	0	3F	3	1
1	05	3	1	0A	-	0	00	B	1	01	F	1
2	07	-	0	0B	-	0	0F	2	1	2B	-	0
3	01	C	1	0C	1	1	02	0	0	1A	1	1

Page Table (only first 16 of the 512 PTEs are shown)

VPN	PPN	Valid	VPN	PPN	Valid	VPN	PPN	Valid	VPN	PPN	Valid
000	3	1	004	-	0	008	3	1	00C	F	1
001	6	1	005	-	0	009	-	0	00D	-	0
002	3	1	006	-	0	00A	1	1	00E	6	1
003	3	1	007	-	0	00B	3	1	00F	A	1

A. Specify which bits correspond to the components of the 16-bit virtual address, namely, the virtual page number (VPN) and the virtual page offset (VPO) by placing "VPN" or "VPO" in each cell.

15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
VPN	*VPN*	*VPN*	*VPN*	*VPN*	*VPN*	*VPN*	*VPN*	*VPN*	*VPO*	*VPO*	*VPO*	*VPO*	*VPO*	*VPO*	*VPO*

B. Now do the same for the TLB by identifying the bits that are used for the TLB set index and the TLB tag, use the labels "TI" and "TT", respectively. Leave any other bits blank.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TT	*TT*	*TT*	*TT*	*TT*	*TT*	*TT*	*TI*	*TI*							

C. Working with the 11-bit physical address, specify which bits correspond to the physical page number (PPN) and the physical page offset (PPO) by using "PPN" and "PPO" labels in each cell.

10	9	8	7	6	5	4	3	2	1	0
PPN	*PPN*	*PPN*	*PPN*	*PPO*	*PPO*	*PPO*	*PPO*	*PPO*	*PPO*	*PPO*

D. Determine the physical address, TLB miss or hit, and whether there a page fault for the following virtual address accesses (write "Y" or "N" for yes or no, respectively, in the TLB Miss? And Page Fault? columns). If you can't determine the PPN and/or physical address and/or TLB miss and/or Page Faulty, simply write ND (for non-determinable) in the appropriate entry in the table.

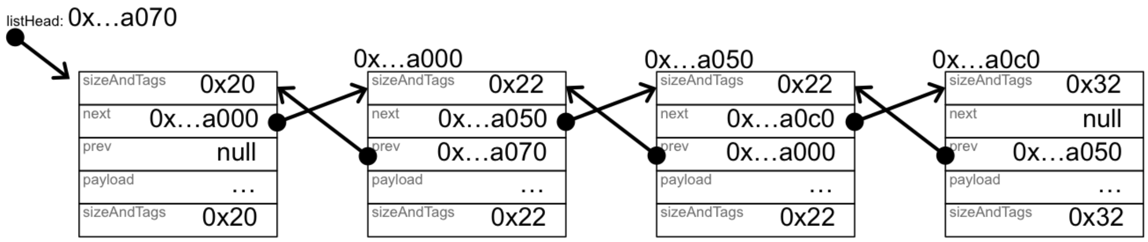
Virtual Address	VPN	TT	TI	PPN	Physical Address	TLB Miss?	Page Fault?
0x1F6A	*000111110*	*0x0F*	*2*	*0010*	*0x16A*	*N*	*N*
0x0EC2	*000011101*	*0x07*	*1*	*ND*	*ND*	*Y*	*ND*
0x05FF	*000001011*	*0x02*	*3*	*0011*	*0x1FF*	*Y*	*N*
0x0C00	*000011000*	*0x06*	*0*	*ND*	*ND*	*Y*	*ND*

5. Memory Allocation – 70pts total (20/A, 20/B, 20/C, 10/D)

A. The following is a map of the heap just after a block was freed and added to the free list. The head of the free list starts at address 0x…a070. Place a check in the "Part of Free Block" column if the 8 bytes represented in that are part of a free block. Place a check mark in the "Size and Tags" column if that row represents a boundary tag for either an allocated or free block.

Address	Original Data	Part of Free Block	Size and Tags	Modified Data
0x…a128:	00000000 00000008			
0x…a120:	00000eee 00000006			
0x…a118:	00000000 00000005			
0x…a110:	00000000 00000004			
0x…a108:	00000000 00000003			
0x…a100:	00000000 00000002			
0x…a0f8:	00000ccc 00000001			
0x…a0f0:	00000000 00000041		▯	
0x…a0e8:	00000000 00000032	▯	▯	
0x…a0e0:	00000000 00000004	▯		
0x…a0d8:	00000000 00000003	▯		
0x…a0d0:	3fffffff ffffa050	▯		*3fffffff ffffa000*
0x…a0c8:	00000000 00000000	▯		
0x…a0c0:	00000000 00000032	▯	▯	
0x…a0b8:	00000000 00000005			
0x…a0b0:	00000eee 00000004			
0x…a0a8:	00000000 00000003			
0x…a0a0:	00000000 00000002			
0x…a098:	00000ddd 00000001			
0x…a090:	00000000 00000031		▯	
0x…a088:	00000000 00000020	▯	▯	*00000000 00000042*
0x…a080:	00000000 00000000	▯		
0x…a078:	3fffffff ffffa000	▯		
0x…a070:	00000000 00000020	▯	▯	
0x…a068:	00000000 00000022	▯	▯	
0x…a060:	3fffffff ffffa000	▯		*00000000 00000000*
0x…a058:	3fffffff ffffa0c0	▯		*3fffffff ffffa000*
0x…a050:	00000000 00000022	▯	▯	*00000000 00000042*
0x…a048:	00000000 00000005			
0x…a040:	00000000 00000004			
0x…a038:	00000000 00000003			
0x…a030:	00000000 00000002			
0x…a028:	00000000 00000001			
0x…a020:	00000000 00000031		▯	
0x…a018:	00000000 00000022	▯	▯	
0x…a010:	3fffffff ffffa070	▯		*3fffffff ffffa050*
0x…a008:	3fffffff ffffa050	▯		*3fffffff ffffa0c0*
0x…a000:	00000000 00000022	▯	▯	

B. Provide a map of the current free list (a doubly-linked list). The first block is shown filled in.



C. The next step is to call "coalesceFreeBlock". In the rightmost column of the table in part A, indicate which values will change – do not bother making entries for data that will not change – and to what value.

See memory map on previous page.

D. What is the new address for the head of the free list?

0x…a050

6. Java – 10pts total

In Java, objects are represented by a struct that includes a header, vtable pointer, and the fields of the object. The vtable, which corresponds to the class of the object provides a jump table to the code for the class's methods. Declarations for two new objects (Vehicle and Car), one a subclass of the other, are shown below as are their data structs and their vtables. Why are additional subclass data fields and methods always put at the end of data struct and vtable? Explain in a sentence or two. HINT: considering the casting of a subclass to a superclass as in:

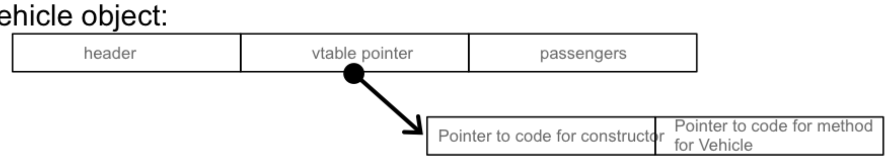
```
Car c1 = new Car();.
Vehicle v1 = (Vehicle) c1;
```

```
class Object {
  …
}
```

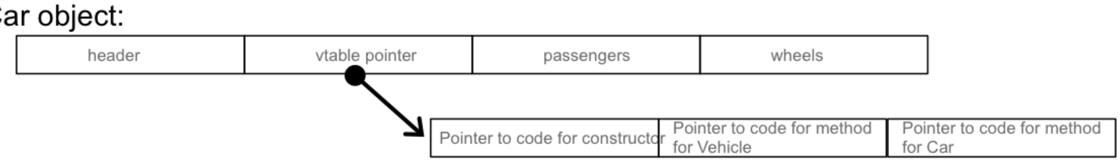
```
class Vehicle {
  int passengers;
}
```

```
class Car extends Vehicle {
  int wheels;
}
```

Vehicle object:



Car object:



The additional fields and methods of subclasses have to go at the end so that field and method offsets can still work as for the superclass without modification.

REFERENCES

Powers of 2:

$2^0 = 1$	
$2^1 = 2$	$2^{-1} = .5$
$2^2 = 4$	$2^{-2} = .25$
$2^3 = 8$	$2^{-3} = .125$
$2^4 = 16$	$2^{-4} = .0625$
$2^5 = 32$	$2^{-5} = .03125$
$2^6 = 64$	$2^{-6} = .015625$
$2^7 = 128$	$2^{-7} = .0078125$
$2^8 = 256$	$2^{-8} = .00390625$
$2^9 = 512$	$2^{-9} = .001953125$
$2^{10} = 1024$	$2^{-10} = .0009765625$

Assembly Code Instructions:

`push` push a value onto the stack and decrement the stack pointer
`pop` pop a value from the stack and increment the stack pointer

`call` jump to a procedure after first pushing a return address onto the stack
`ret` pop return address from stack and jump there

`mov` move a value between registers and memory
`lea` compute effective address and store in a register

`add` add src (1st operand) to dst (2nd) with result stored in dst (2nd)
`sub` subtract src (1st operand) from dst (2nd) with result stored in dst (2nd)
`and` bit-wise AND of src and dst with result stored in dst
`or` bit-wise OR of src and dst with result stored in dst
`shl` shift data in the dst to the left (logical shift) by the number of bits specified in the 1st operand

`jmp` jump to address
`jne` conditional jump to address if zero flag is not set
`cmp` subtract src (1st operand) from dst (2nd) and set flags
`test` bit-wise AND src and dst and set flags

Suffixes for `mov` instructions:
`s` or `z` for sign-extended or zero-ed, respectively
Suffixes for all instructions:
`b`, `w`, `l`, or `q` for byte, word, long, and quad, respectively

Reference from Lab 5:

The functions, macros, and structs from lab5. These are all identical to those in the lab. Note that some of them will not be needed in answering the following questions.

Structs:

```
struct BlockInfo {
  // Size of the block (in the high bits) and tags for whether the
  // block and its predecessor in memory are in use.  See the SIZE()
  // and TAG macros, below, for more details.
  size_t sizeAndTags;
  // Pointer to the next block in the free list.
  struct BlockInfo* next;
  // Pointer to the previous block in the free list.
  struct BlockInfo* prev;
};
```

Macros:

```
/* Macros for pointer arithmetic to keep other code cleaner.  Casting
   to a char* has the effect that pointer arithmetic happens at the
   byte granularity. */
#define UNSCALED_POINTER_ADD …
#define UNSCALED_POINTER_SUB …

/* TAG_USED is the bit mask used in sizeAndTags to mark a block as
   used. */
#define TAG_USED 1

/* TAG_PRECEDING_USED is the bit mask used in sizeAndTags to indicate
   that the block preceding it in memory is used. (used in turn for
   coalescing).  If the previous block is not used, we can learn the
   size of the previous block from its boundary tag */
#define TAG_PRECEDING_USED 2;

/* SIZE(blockInfo->sizeAndTags) extracts the size of a 'sizeAndTags'
   field. Also, calling SIZE(size) selects just the higher bits of
   'size' to ensure that 'size' is properly aligned.  We align 'size'
   so we can use the low bits of the sizeAndTags field to tag a block
   as free/used, etc, like this:

      sizeAndTags:
      +-------------------------------------------+
      | 63 | 62 | 61 | 60 |  . . . .  | 2 | 1 | 0 |
      +-------------------------------------------+
        ^                                       ^
      high bit                               low bit

   Since ALIGNMENT == 8, we reserve the low 3 bits of sizeAndTags for
   tag bits, and we use bits 3-63 to store the size.
   Bit 0 (2^0 == 1): TAG_USED
   Bit 1 (2^1 == 2): TAG_PRECEDING_USED
*/
#define SIZE …
```

```
/* Alignment of blocks returned by mm_malloc. */
# define ALIGNMENT 8

/* Size of a word on this architecture. */
# define WORD_SIZE 8

/* Minimum block size (to account for size header, next ptr, prev ptr,
   and boundary tag) */
#define MIN_BLOCK_SIZE …

/* Pointer to the first BlockInfo in the free list, the list's head.
   A pointer to the head of the free list in this implementation is
   always stored in the first word in the heap.  mem_heap_lo() returns
   a pointer to the first word in the heap, so we cast the result of
   mem_heap_lo() to a BlockInfo** (a pointer to a pointer to
   BlockInfo) and dereference this to get a pointer to the first
   BlockInfo in the free list. */
#define FREE_LIST_HEAD …
```

Code for coalesceFreeBlock:

```
/* Coalesce 'oldBlock' with any preceeding or following free blocks. */
static void coalesceFreeBlock(BlockInfo* oldBlock) {
  BlockInfo *blockCursor;
  BlockInfo *newBlock;
  BlockInfo *freeBlock;
  // size of old block
  size_t oldSize = SIZE(oldBlock->sizeAndTags);
  // running sum to be size of final coalesced block
  size_t newSize = oldSize;

  // Coalesce with any preceding free block
  blockCursor = oldBlock;
  while ((blockCursor->sizeAndTags & TAG_PRECEDING_USED)==0) {
    // While the block preceding this one in memory (not the
    // prev. block in the free list) is free:

    // Get the size of the previous block from its boundary tag.
    size_t size = SIZE(*((size_t*)UNSCALED_POINTER_SUB(blockCursor,
WORD_SIZE)));
    // Use this size to find the block info for that block.
    freeBlock = (BlockInfo*)UNSCALED_POINTER_SUB(blockCursor, size);
    // Remove that block from free list.
    removeFreeBlock(freeBlock);

    // Count that block's size and update the current block pointer.
    newSize += size;
    blockCursor = freeBlock;
  }
  newBlock = blockCursor;

  // Coalesce with any following free block.
  // Start with the block following this one in memory
  blockCursor = (BlockInfo*)UNSCALED_POINTER_ADD(oldBlock, oldSize);
  while ((blockCursor->sizeAndTags & TAG_USED)==0) {
    // While the block is free:

    size_t size = SIZE(blockCursor->sizeAndTags);
    // Remove it from the free list.
```

```
    removeFreeBlock(blockCursor);
    // Count its size and step to the following block.
    newSize += size;
    blockCursor = (BlockInfo*)UNSCALED_POINTER_ADD(blockCursor, size);
  }

  // If the block actually grew, remove the old entry from the free
  // list and add the new entry.
  if (newSize != oldSize) {
    // Remove the original block from the free list
    removeFreeBlock(oldBlock);

    // Save the new size in the block info and in the boundary tag
    // and tag it to show the preceding block is used (otherwise, it
    // would have become part of this one!).
    newBlock->sizeAndTags = newSize | TAG_PRECEDING_USED;
    // The boundary tag of the preceding block is the word immediately
    // preceding block in memory where we left off advancing blockCursor.
    *(size_t*)UNSCALED_POINTER_SUB(blockCursor, WORD_SIZE) = newSize |
                                                   TAG_PRECEDING_USED;

    // Put the new block in the free list.
    insertFreeBlock(newBlock);
  }
  return;
}
```