Instructions:

- Print your name in the space provided below.
- This examination is closed book and closed notes, aside from the permitted one-page formula sheet.
- No calculators or other computing devices may be used. The use of any such device will be interpreted as an indication that you are finished with the test and your test form will be collected immediately.
- Answer each question in the space provided. If you need to continue an answer onto the back of a page, clearly indicate that and label the continuation with the question number.
- If you want partial credit, justify your answers, even when justification is not explicitly required.
- There are 8 questions, some with multiple parts, priced as marked. The maximum score is 100.
- When you have completed the test, sign the pledge at the bottom of this page, sign your fact sheet, and turn in the test and fact sheet.
- Note that failing to return this test, and discussing its content with a student who has not taken it are violations of the Honor Code.

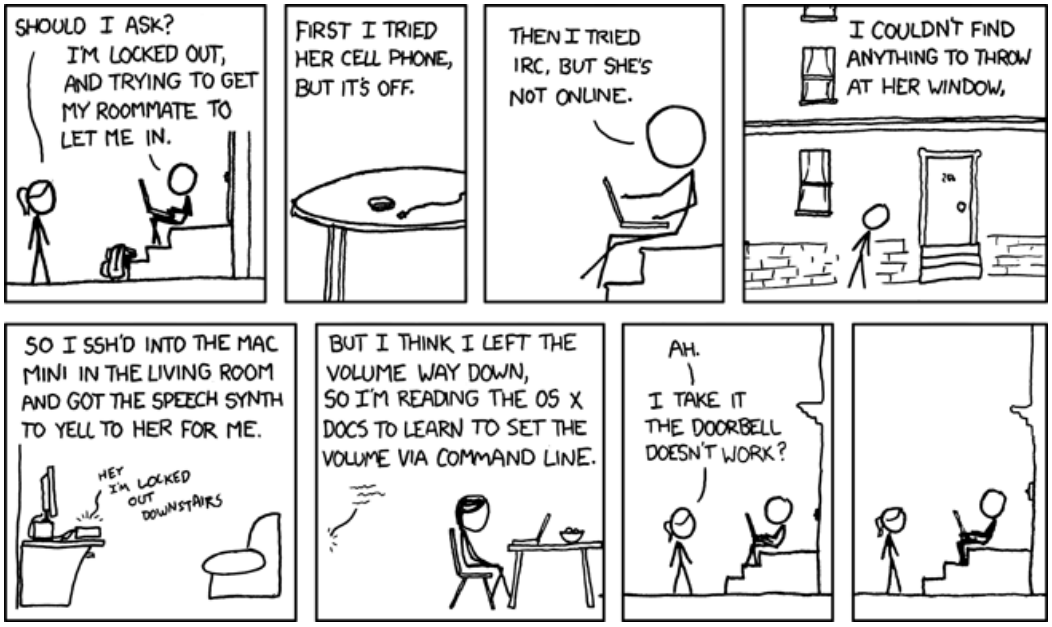
Do not start the test until instructed to do so!

Name Solution

printed

Pledge: On my honor, I have neither given nor received unauthorized aid on this examination.

signed


SHOULD I ASK?
I'M LOCKED OUT, AND TRYING TO GET MY ROOMMATE TO LET ME IN.
FIRST I TRIED HER CELL PHONE, BUT IT'S OFF.
THEN I TRIED IRC, BUT SHE'S NOT ONLINE.
I COULDN'T FIND ANYTHING TO THROW AT HER WINDOW,
SO I SSH'D INTO THE MAC MINI IN THE LIVING ROOM AND GOT THE SPEECH SYNTH TO YELL TO HER FOR ME.
HEY I'M LOCKED OUT DOWNSTAIRS
BUT I THINK I LEFT THE VOLUME WAY DOWN, SO I'M READING THE OS X DOCS TO LEARN TO SET THE VOLUME VIA COMMAND LINE.
AH.
I TAKE IT THE DOORBELL DOESN'T WORK?

xkcd.com

I. Processes 20 points

1. [13 points] Suppose the following C code is executed up to the comment `HERE`. The executables `progA`, `progB`, `progC` and `progD` are installed in directories in the system path.

At that point in execution, how many processes (created as a result of running this program) are running, and what code are they executing?

```
int main() {                          // Proc1 is running this code.

   int index = 0;

   int pid = fork();                  // Creates Child1, running this code.

   if ( pid == 0 && index == 0 ) {    // Child1 enters here.
      index++;                        // Child1 increments its version of index.
      pid = fork();                   // Creates Gchild1, running this code.
      execve("progA", NULL, NULL);    // Child1 and Gchild1 both run progA.
   }
   if ( pid != 0 && index != 0 ) {    // Nobody enters here.
      pid = fork();
      execve("progB", NULL, NULL);
   }
   if ( pid != 0 && index == 0 ) {    // Proc1 enters here.
      pid = fork();                   // Creates Child2, running this code.
      if ( pid == 0 ) {               // Proc1 enters here, runs progC.
         execve("progC", NULL, NULL);
      }                               // Child2 falls to here, runs progD.
      execve("progD", NULL, NULL);
   }
   // HERE
   . . .
}
```

So, we have four processes:

Proc1, running progD.
Child1, running progA.
Gchild1, also running progA.
Child2, running progC.

Remember:

- a fork() call creates a new process running the same program (from the point of the return from fork())
- an exec() call loads a new program into the calling process, and does not return.
- a fork'd child process has its own memory space; hence its own copies of variables

2. [7 points] Circle the appropriate classification for each of the following statements:

A `fork`'d child process automatically shares local variables with its parent process. **A fork'd child gets a copy of the parent's address space.**	true (false)
A call to `fork()` creates a child process and triggers a context switch to the child process. **Calling fork() does not cause a context switch to the child, although we expect the child will eventually be scheduled to run.**	true (false)
A call to `fork()` creates a child process that is running the same program as its parent. **Straight from the notes.**	(true) false
System calls always result in a mode switch. **System call... calling kernel code... mode switch; exception would be the case where system code (already running in kernel mode) calls other system code.**	(true) false
Mode switches occur only when a running process makes a system call. **For example, a mode switch may occur because the kernel takes an action, like a context switch.**	true (false)
When a process sends a signal, the sending process is not necessarily blocked until the signal is delivered. **Sending a signal would not, usually, cause the sender to block at all.**	(true) false
A signal handler, written as part of a user process, executes in kernel mode. **That would be very dangerous... and hence is not done.**	true (false)

II. Multi-threading 30 points

3. Consider the following declarations and code:

```
#define SZ 100

int A[SZ];
int B[SZ];

void incSegment(int* List, int Start, int Stop) {

   for (int Idx = Start; Idx <= Stop; Idx++) {
      List[Idx]++;
   }
}

int addSegment(int* List, int Start, int Stop) {

   int Sum = List[Start];
   for (int Idx = Start + 1; Idx <= Stop; Idx++) {
      Sum += List[Idx];
   }
   return Sum;
}
```

a) [8 points] Suppose two threads are created and execute the function calls shown below. For each case, determine whether the execution of the two threads involves a race condition.

Thread 1	Thread 2	Race condition?
`addSegment(A, 20, 30)`	`addSegment(A, 25, 35)`	No Neither modifies A[].
`incSegment(A, 50, 75)`	`incSegment(B, 50, 75)`	No Modifying different arrays.
`incSegment(B, 0, 20)`	`incSegment(B, 21, 40)`	No Modifying different ranges.
`addSegment(A, 20, 30)`	`incSegment(A, 25, 35)`	Yes Thread 2 modifies array cells 25:30, which are read by Thread 1.

b) [6 points] Suppose the two functions given earlier were modified as shown below, with a suitably-initialized global `pthread_mutex_t` variable named `lock`:

```
void incSegment(int* List, int Start, int Stop) {

   pthread_mutex_lock( &lock );
   for (int Idx = Start; Idx <= Stop; Idx++) {
      List[Idx]++;
   }
   pthread_mutex_unlock( &lock );
}

int addSegment(int* List, int Start, int Stop) {

   pthread_mutex_lock( &lock );
   int Sum = List[Start];
   for (int Idx = Start + 1; Idx <= Stop; Idx++) {
      Sum += List[Idx];
   }
   pthread_mutex_unlock( &lock );
   return Sum;
}
```

Would this change eliminate all of the race conditions, if any, that you identified in part a)? Explain.

A thread running either function cannot reach the point that it accesses a potentially-shared array unless it first holds the lock.

Note: a race condition occurs if the interleaved execution of two or more threads can yield different results, depending on how the threads are interleaved. In the code shown above, there is no race condition; whichever thread acquires the lock first will report logically correct results, or make logically correct modifications, given the state of the array when the lock is acquired.

c) [4 points] Suppose, for the purpose of this question, that the change shown in part b) was made, and that it would eliminate all possible race conditions. Would the change be acceptable otherwise? Explain.

The basic objection is that there is a loss of potential concurrency:

- **there could not be two threads both running addSegment() concurrently, even though that could not lead to interference;**
- **there could not be two threads operating on different segments of the same array, even though that could not lead to interference;**
- **there could not be two threads operating on different arrays, even though that could not lead to interference (we could fix that by having different locks for each array).**

4. [12 points] A multithreaded program makes use of the following data type, using objects of type `Shared` that are created by calls to the function `makeShared()`:

```
struct _Shared {
   int counter;
   int end;
   int *array;
};
typedef struct _Shared Shared;

Shared* makeShared(int end) {

   Shared *shared = malloc (sizeof(Shared));
   shared->array = calloc (end, sizeof(int));

   shared->counter = 0;
   shared->end = end;

   return shared;
}
```

The program uses the following `main()` function:

```
int main() {

   int i;
   pthread_t child[NUM_CHILDREN];

   // The following object is shared amongst all the threads:
   Shared *myShared = makeShared(100000000);

   for (i = 0; i < NUM_CHILDREN; i++) {
      pthread_create(child + i, NULL, threadFunc, (void *) myShared);
   }

   for (i = 0; i < NUM_CHILDREN; i++) {
      pthread_join(child[i], NULL);
   }

   checkOnes(myShared);  // count 1s in the array in shared
   return 0;
}
```

Each thread begins by executing the following function:

```
void* threadFunc (void *arg) {

   Shared* ourShared = (Shared*) arg;

   writeSomeOnes(ourShared);

   printf("Child done.\n");

   pthread_exit(NULL);
}
```

A solution that locked/unlocked around the call to writeSomeOnes() completely avoids concurrent execution of the threads.

That is not a valid response to the given question.

And, as you can see, each thread then calls and executes this function:

```
void writeSomeOnes(Shared *shared) {

  printf ("Starting child at counter %d\n", shared->counter);
  while ( 1 ) {
    pthread_mutex_lock( &lock );  // acquire lock on *shared
    if (shared->counter >= shared->end) { // quit if all elems have been set
      pthread_mutex_unlock( &lock );  // release lock on *shared,
                                      //   before exiting fn
      return;
    }

    shared->array[shared->counter]++;

    shared->counter++;

    pthread_mutex_unlock( &lock );  // release lock on *shared
  }

}
```

Unfortunately, if the program is executed with four threads, we get the following results:

```
CentOS > ./prog
Starting child at counter 0
Starting child at counter 443811
Starting child at counter 1019503
Starting child at counter 2575987
Child done.
Child done.
Child done.
Child done.
54907032 errors.
```

Whether you locked/unlocked around the calls to printf() was not vital.

Acquiring the lock before entering the loop eliminated concurrency.

Failing to unlock inside the if results in deadlock unless there's only one thread.

Your task is to fix this behavior, so that each element of the shared array will be set to 1 (and so no errors would be reported).

You may use pthread mutex variables, condition variables, or semaphores, as you like; these should be declared at file scope, and you should write the declarations/initializations of those in the space below.

```
pthread_mutex_t lock = initially unlocked;
```

You may add appropriate synchronization code in `threadFunc()`, or in `writeSomeOnes()`, or in both, as you see fit.

III. Linking and Loading 30 points

5. Consider the short C program, consisting of two source files, shown below:

```
// a.c
A1  extern int A;
A2  int B = 10;
A3  int C;

A4  int main() {

A5      int D = 0;
A6      C = A;
A7      D = f();
A8      return 0;
    }
```

```
// b.c
B1  int A = 15;
B2  int C = 35;

B3  int f() {

B4      return (A + C);
    }
```

The given C code compiles, links, and executes with no runtime errors.

a) [4 points] Where is the variable `A`, referred to in line ***A6***, defined? (Just state a line number.)

B1

Consider lines ***A1*** and ***B1***, both of which involve the identifier `A`. Why is your answer above correct?

The extern statement in *A1* does not define a variable; it just indicates there should be a definition of that variable in some other file.

b) [4 points] Where is the variable `C`, referred to in line ***A6***, defined? (Just state a line number.)

B2

Consider lines ***A3*** and ***B2***, both of which involve the identifier `C`. Why is your answer above correct?

Line *A3* declares a weak symbol *C* (no initialization); line *B2* defines a strong symbol that overrides the weak symbol.

c) [2 points] Where is the variable `A`, referred to in line ***B4***, defined? (Just state a line number.)

B1

d) [2 points] Where is the variable `C`, referred to in line ***B4***, defined? (Just state a line number.)

B2

6. When the two C files given above are compiled with the switches `-c -m32 -O0`, we get two object files, `a.o` and `b.o`. Running `objdump` shows that `gcc` produced the following `.text` and `.data` segments from `a.c` and `b.c` (some irrelevant details have been omitted). Relocation records are highlighted in gray.

```
a.o:
.text
00000000 <main>:
   0:  55                      push   %ebp
   1:  89 e5                   mov    %esp,%ebp
   3:  83 e4 f0                and    $0xfffffff0,%esp
   6:  83 ec 10                sub    $0x10,%esp
   9:  c7 44 24 0c 00 00 00    movl   $0x0,0xc(%esp)
  10:  00
  11:  a1 00 00 00 00          mov    0x0,%eax
                                 12: R_386_32    A
  16:  a3 00 00 00 00          mov    %eax,0x0
                                 17: R_386_32    C
  1b:  e8 fc ff ff ff          call   1c <main+0x1c>
                                 1c: R_386_PC32  f
  20:  89 44 24 0c             mov    %eax,0xc(%esp)
  24:  b8 00 00 00 00          mov    $0x0,%eax
  29:  c9                      leave
  2a:  c3                      ret

.data
00000000 <B>:
   0:  0a 00 00 00
```

```
b.o:
.text
00000000 <f>:
   0:  55                      push   %ebp
   1:  89 e5                   mov    %esp,%ebp
   3:  8b 15 00 00 00 00       mov    0x0,%edx
                                 5: R_386_32   A
   9:  a1 00 00 00 00          mov    0x0,%eax
                                 a: R_386_32   C
   e:  01 d0                   add    %edx,%eax
  10:  5d                      pop    %ebp
  11:  c3                      ret

.data
00000000 <A>:
   0:  0f 00 00 00
00000004 <C>:
   4:  23 00 00 00
```

a) [2 points] The code in the file `a.c` refers to four variables: `A`, `B`, `C` and `D`. Why is only one of those variables shown in the `.data` segment for `a.o`?

A is defined in a different file.
C is weak, and would not be shown in the .data segment even if there were no strong C.
D is a local automatic and is stored on the stack.

b) [4 points] In the disassembly of `a.o`, we find the following lines:

```
. . .
 9:  c7 44 24 0c 00 00 00   movl   $0x0,0xc(%esp)
10:  00
11:  a1 00 00 00 00        mov    0x0,%eax
                             12: R_386_32    A
16:  a3 00 00 00 00        mov    %eax,0x0
                             17: R_386_32    C
. . .
```

The `movl` instruction is initializing the variable `D`, and the `mov` instructions are copying the variable `A` into the variable `C`. Why are there a relocation records for the two `mov` instruction, but no relocation record for the `movl` instruction?

D is a local automatic, so its address is not resolved by the linker; A and C have static duration, and their addresses are resolved by the linker.

c) [2 points] In the disassembly of `a.o`, we find the following line, which is a call to the function `f()`:

```
. . .
1b:  e8 fc ff ff ff        call   1c <main+0x1c>
                             1c: R_386_PC32  f
. . .
```

The relocation tag here is `R_386_PC32`, whereas the tags for the code in part b) were `R_386_32`. Why is there a difference? (Simply saying that this is a function call is not an acceptable answer.)

The parameter to the call instruction is a PC-relative offset to the address of the called procedure.

Addresses of variables are absolute, not PC-relative.

7. When the two object files are linked to form an executable elf file, the linker produces the following `.text` and `.data` segments (some irrelevant details have been omitted, as have some relevant details).

```
.text
080483f0 <main>:
 . . .
 80483f9:      c7 44 24 0c 00 00 00    movl   $0x0,0xc(%esp)
 8048400:      00
 8048401:      a1 ** ** ** **          mov    *********,%eax
 8048406:      a3 ** ** ** **          mov    %eax,*********
 804840b:      e8 0c 00 00 00          call   804841c <f>
 8048410:      89 44 24 0c             mov    %eax,0xc(%esp)
 . . .

0804841c <f>:
 804841c:      55                      push   %ebp
 . . .
 804841f:      8b 15 ** ** ** **       mov    *********,%edx
 8048425:      a1 ** ** ** **          mov    *********,%eax
 . . .

.data
0804a018 <B>:
 804a018:      0a 00 00 00
0804a01c <A>:
 804a01c:      0f 00 00 00
0804a020 <C>:
 804a020:      23 00 00 00
```

a) [4 points] When the `call` machine instruction in `main()` was linked, the parameter to `call` became `0x0000000c` (remember, little endian byte-ordering is used). Why is that value correct?

When the call instruction is being executed, the PC will store the address of the next instruction (0x0804810). The address of f is 0x0804841c. The difference of the two addresses is 0x0000000c.

b) [6 points] The following two instructions required relocation by the linker:

```
 804841f:      8b 15 ** ** ** **       mov    *********,%edx
 8048425:      a1 ** ** ** **          mov    *********,%eax
```

Write the completed machine code for the two instructions, as the linker would have after it performed the necessary relocations. You might want to refer back to the object dump for `b.o`.

The values are the addresses of the variables A and C, respectively:

```
804841f:      8b 15 1c a0 04 08
8048425:      a1 20 a0 04 08
```

IV. Explicit Memory Management 20 points

8. Consider the use of an explicit free list.

a) [6 points] When using the implicit list approach, it's necessary to use a boundary tag (footer) in each block in the list, because a block has no other way to determine the length of the previous block. But, if an explicit list is used, each free block has pointers to (near) the beginning of the preceding and succeeding free blocks. So, why is a boundary tag still necessary when using an explicit list?

When a block B is deallocated, it will not store pointers to any other blocks. There is, therefore, no efficient way to determine whether the block immediately preceding B is free unless that block contains a boundary tag.

We could, of course, traverse the free list looking for a block that immediately precedes B, but that would be very inefficient.

b) [6 points] When an explicit list is used, the memory manager must store a list node (two pointers) in each block in the list. How much impact does that have on the amount of memory that's available to store client data? Explain.

As mentioned above, allocated blocks will not store free list nodes, so they have no impact at all on how much space is available for client data.

The only quibble is that this does impose a minimum size on a free block, since there must be enough room for the pointers; that could, in principle, result in more "padding" in certain allocations and reduce the total amount of memory available to clients.

c) [8 points] Discuss the pros/cons of using deferred coalescing.

If we defer coalescing when a block is freed:

- we reduce the cost of performing a free (obviously)
- we may save cost if that freed block is subsequently chosen to satisfy an allocation request

If we do not defer coalescing when a block is freed:

- we may have to coalesce on the fly when looking for a block to satisfy a subsequent allocation request, making mallocs more costly
- we will have more elements in the free block list, possibly making searches more costly