DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.033 Spring, 1999 Handout 15: March 5, 1999

6.033 Computer System Engineering: Spring, 1999

Quiz 1

All problems on this exam are multiple-choice questions. In order to receive credit you must fill in the blank(s) or mark the correct answer or answers for each question. You have 50 minutes to answer this exam.

Write your name on this cover sheet AND at the bottom of each page of this booklet.

Some questions may be much harder than others. Read them all through first and attack them in the order that allows you to make the most progress. If you find a question ambiguous, be sure to write down any assumptions you make. Be neat. If we can't understand your answer, we can't give you credit!

THIS IS AN OPEN BOOK, OPEN NOTES QUIZ.

Circle your recitation section number:

10:00	1. Sussman/Rahul	11. Rudolph/Forbess	
11:00	2. Sussman/Doering	4. Seshan/Boyapati	6. Rudolph/Rahul
12:00	5. Balakrishnan/Fu		
1:00	3. Seshan/Forbess	7. Saltzer/Hu	8. Balakrishnan/Shlyakhter
	10. Teller/Boyapati		
2:00	9. Saltzer/Shlyakhter	12. Teller/Doering	

Do not write in the boxes below

1-6 (XX/36)	7-11 (XX/46)	12-13 (XX/18)	Total (XX/100)

Name: ______________________________

I. Any of this sound familiar?

1. [6 points]: The term “computer system” refers to
(circle letter to left of best answer)

A. A complex system that consists of a large number of elements and interconnections.

B. A system that consists of multiple computers linked by a network.

C. Any automated system whose purpose it is to store and process information.

D. None of the above.

2. [6 points]: Hierarchy reduces complexity because
(circle letter to left of ALL THAT APPLY)

A. It cuts down on the number of interconnections between elements.

B. It assembles a number of smaller elements into one large element.

C. It enforces a structure on the interconnections between elements.

D. All of the above.

3. [6 points]: The X-window system illustrates the use of
(circle letter to left of best answer)

A. Client/server organization.

B. Remote procedure calls.

C. Virtual memory.

D. Isochronous communications.

4. [6 points]: In UNIX, directories are implemented as
(circle letter to left of best answer)

A. Files.

B. Processes.

C. Pipes.

D. None of the above.

Name: ____

5. [6 points]: Which of the following races are caught by Eraser?

(circle letter to left of best answer)

A. *Some* of the possible dynamic races that can occur in a *particular* program execution.

B. *All* the possible dynamic races that can occur in *any* program execution, except for data races during initialization.

C. *All* the possible dynamic races that can occur in *any* program execution after initialization of a variable.

D. *All* the possible dynamic races that can occur in *any* program execution, except for read-shared data.

6. [6 points]: Which of these statements are true of Birrell and Nelson's remote procedure call system?

(circle letter to left of ALL THAT APPLY)

A. An application program need not know which machine the remote procedure *callee* is running on.

B. It provides stubs for marshalling and unmarshalling of arguments and return values.

C. It hides *all* server failures from client applications.

D. It hides *some* network failures from client applications.

II. BOOZE: Ben's OOZE

Ben Bitdiddle writes a large number of object-oriented programs. After reading the OOZE paper, he is inspired to redesign his page-based virtual memory system (PAGE) into an object memory system. PAGE is a page-based virtual memory system like the one described in the lecture notes. BOOZE is Ben's object-based virtual memory system. Of course, he can run his programs on either system.

Each BOOZE object has a unique ID, called a UID. A UID has three fields: (1) a disk address for the disk block that contains the object; (2) an offset within that disk block where the object starts; and (3) the size of the object.

```
struct uid {
   int blocknr;      // disk address for disk block
   int offset;       // offset within block "blocknr"
   int size;         // size of object
}
```

Applications running on BOOZE and PAGE have similar structure. The only difference is that on PAGE, objects are referred to by their virtual address, while on BOOZE they are referred to by UIDs.

The two levels of memory in BOOZE and PAGE are main memory and a disk. The disk is a linear array of fixed-size blocks of 4Kbyte. A disk block is addressed by its block number. In *both* systems, the transfer unit between the disk and main memory is a 4Kbyte block. Objects don't cross disk block boundaries, are smaller than 4Kbyte, and cannot change size. The page size in PAGE is equal to the disk-block size; therefore, when an application refers to an object, PAGE will bring in all objects on the same page.

BOOZE keeps an object map in main memory. The object map contains entries that map a UID to the memory address of the corresponding object.

```
struct mapentry {
   struct uid UID;
   int addr;
}
```

On all references to an object, BOOZE translates a UID to an address in main memory. BOOZE uses the following procedure for translation:

```
ObjectToAddress(UID) returns address {
   addr = isPresent(UID);             // is UID present in object map?
   if (addr >= 0) return addr;        // UID is present, return addr
   addr = findFreeSpace(UID.size); // allocate space to hold object
   readObject(addr, UID);   // read object from disk & store at addr
   enterIntoMap(UID, addr);           // enter UID in object map
   return addr;                   // return memory address for object
}
```

Name: ______________________________

isPresent looks up UID in the object map; if present, it returns the address of the corresponding object; otherwise, it returns -1. **findFreeSpace** allocates free space for the object; it might *evict* (replace) another object to make space available for this one. **readObject** reads the *page* that contains the object, and then copies the *object* to the allocated address.

7. [9 points]: What does **addr** in the **mapentry** data structure denote?
(circle letter to left of best answer)

A. The memory address at which the object map is located.

B. The disk address at which to find a given object.

C. The memory address at which to find a given object that is *currently* resident in memory.

D. The memory address at which a given non-resident object *would have to be loaded*, when an access is made to it.

8. [9 points]: In what ways is BOOZE better than PAGE?
(circle letter to left of best answer)

A. Applications running on BOOZE generally use less main memory, because BOOZE stores only objects that are referenced.

B. Applications running on BOOZE generally run faster, because UIDs are smaller than virtual addresses.

C. Applications running on BOOZE generally run faster, because BOOZE transfers objects from disk to main memory instead of complete pages.

D. Applications running on BOOZE generally run faster, because typical applications will exhibit better locality of reference.

When **findFreeSpace** cannot find enough space to hold the object, it needs to write one or more objects back to the disk to create free space. **findFreeSpace** uses **writeObject** to write an object to the disk.

Ben is figuring out how to implement **writeObject**. He is considering the following options:

```
1. writeObject(addr, UID) {
       write(addr, UID.blocknr, 4Kbyte)
   }
2. writeObject(addr, UID) {
       read(buffer, UID.blocknr, 4Kbyte);
       copy(addr, buffer + UID.offset, UID.size);
       write(buffer, UID.blocknr, 4KByte);
   }
```

read(mem_addr, disk_addr, 4Kbyte) and **write(mem_addr, disk_addr, 4Kbyte)**

read and write a 4Kbyte page from/to the disk. `copy(destination, source, size)` copies `size` bytes from a source address to a destination address in main memory.

9. [9 points]: Which implementation should Ben use?

(circle letter to left of best answer)

A. Implementation 2, since implementation 1 is incorrect.

B. Implementation 1, since it is more efficient than Implementation 2.

C. Implementation 1, since it is easier to understand.

D. Implementation 2, since it will result in better locality of reference.

Ben now turns his attention to optimizing the performance of BOOZE. In particular, he wants to reduce the number of writes to the disk.

10. [9 points]: Which of the following techniques will reduce the number of writes without losing correctness?

(circle letter to left of best answer)

A. Prefetching objects on a read.

B. Delaying writes to disk until the application finishes its computation.

C. Writing to disk only objects that have been modified.

D. Delaying a write of an object to disk until it is accessed again.

Ben decides that he wants even better performance, so he decides to modify `findFreeSpace.` When `findFreeSpace` has to evict an object, it now tries not to write an object modified in the last 30 seconds (in the belief that it may be accessed soon). Ben does this by setting the `dirty` flag when the object is modified. Every 30 seconds, BOOZE calls a procedure `write-behind` that walks through the object map and writes out all objects that are dirty. After an object has been written, `write-behind` clears its `dirty` flag. When `findFreeSpace` needs to evict an object to make space for another, clean objects are the *only* candidates for replacement.

When running his applications on the latest version of BOOZE, Ben observes once in a while that BOOZE runs out of physical memory when calling `ObjectToAddress` for a new object.

11. [10 points]: Which of these strategies avoids the above problem?

(circle letter to left of ALL THAT APPLY)

A. When `findFreeSpace` cannot find any clean objects, it calls `write-behind` and try finding clean objects again.

B. BOOZE could call `write-behind` after 1 second instead of 30 seconds.

C. When `findFreeSpace` cannot find any clean pages, it picks *one* dirty object, writes the page containing the object to the disk, clears the `dirty` flag, and then uses that page for the new object.

D. All of the above strategies.

III. Toastac-25

Louis P. Hacker-Reasoner (no relation to others with similar names!) bought a used Therac-25 for $14.99 at a yard sale. After some slight modifications, he has hooked it up to the Ethernet as a computer-controllable turbo-toaster, which can toast one slice in under 2 milliseconds. (Louis likes his toast black.) He decides to use RPC to control the Toastac-25. Each toasting request starts a new thread on the server, which cooks the toast and returns an acknowledgement (or perhaps a helpful error code, such as "Malfunction 54"). Here's what the server thread looks like:

```
server thread:
   acquire(message_buffer_lock);
   decode message;
   acquire(accelerator_buffer_lock);
   release(message_buffer_lock);
   cook_toast();
   acquire(message_buffer_lock);
   put acknowledgement in message buffer;
   send message;
   release(accelerator_buffer_lock);
   release(message_buffer_lock);
   exit;
```

12. [9 points]: To his surprise, the toaster stops cooking toast the first time it is heavily used! What has gone wrong?

(circle letter to left of best answer)

A. Two server threads might deadlock, because one has **`message_buffer_lock`** and wants **`accelerator_buffer_lock`**, while the other has **`accelerator_buffer_lock`** and wants **`message_buffer_lock`**.

B. Two server threads might deadlock, because one has **`accelerator_buffer_lock`** and **`message_buffer_lock`.**

C. Toastac-25 deadlocks, because **`cook toast`** is not an atomic operation.

D. Insufficient locking allows inappropriate interleaving of server threads.

Once Louis fixes the multithreaded server, the Toastac gets more use than ever. However, when the Toastac has many simultaneous requests (i.e., there are many threads), he notices the system performance degrades badly—much more than he expected. Performance analysis shows that competition for locks is not the problem.

Name: ______________________________

13. [9 points]: What could be going wrong?

(circle letter to left of best answer)

A. The Toastac system spends all its time context switching between threads.

B. The Toastac system spends all its time waiting for requests to arrive.

C. The Toastac gets hot, and therefore cooking a toast takes longer.

D. The Toastac systems spends all its time releasing locks.

END OF QUIZ 1

(have a good weekend!)