Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.097 Operating System Engineering: Fall 2002

Quiz II

All problems are open-ended questions. In order to receive credit you must answer the question as precisely as possible. You have 80 minutes to answer this quiz.

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.

1 (xx/20)	2 (xx/20)	3 (xx/20)	4 (xx/20)	5 (xx/20)	Total (xx/100)

Name: Alyssa P. Hacker

I Fast mutual exclusion

A common way to implement concurrent inserts is using an atomic `TSL` (Test-Set-and-Lock) instruction:

```
List *list = NULL;
int insert_lock = 0;

insert(int data) {

  /* acquire the lock: */
  while(TSL(&insert_lock) != 0)
    ;

  /* critical section: */
  List *l = new List;
  l->data = data;
  l->next = list;
  list = l;

  /* release the lock: */
  insert_lock = 0;
}
```

Many processors support an atomic `cmpxchg` instruction. This instruction has the following semantics:

```
int cmpxchg(addr, v1, v2) {
  int ret = 0;
  // stop all memory activity and ignore interrupts
  if (*addr == v1) {
    *addr = v2;
    ret = 1;
  }
  // resume other memory activity and take interrupts
  return ret;
}
```

1. [5 points]: Give an implementation of `insert` using the `cmpxchg` instruction.

```
insert (int data) {
  List *l = new List;
  l->data = data;
  do {
    l->next = list;
  } while (!cmpxchg (&list, l->next, l));
}
```

2. [5 points]: Why is the implementation of `insert` with `cmpxchg` preferable over the one with `TSL`?

The `cmpxchg` version is wait free. There is no chance for a deadlock because there are no locks.

3. [5 points]: Give an implementation of `cmpxchg` using a restartable atomic sequence.

```
int cmpxchg_RAS(addr, v1, v2) {
  int ret = 0;
  BEGIN RAS
  if (*addr == v1) {
    *addr = v2;
  END RAS
    ret = 1;
  }
  return ret;
}
```

4. [5 points]: Can you extend the `insert()` routine to insert nodes into a doubly linked list using `cmpxchg` and without using locks? If so, give the code, otherwise explain why not.

```
struct List {
   List *next;
   List *prev;
   int data;
};
```

You can not just use `cmpxchg` to insert into a doubly linked list because the insert into list operations:

```
insert(int data) {
  // Prepare new element
  List *l = new List;
  l->data = data;
  l->prev = NULL;

  // Insert into list
  l->next = list;
  list->prev = l;
  list = l;
}
```

require two externaly visible memory writes. Even if `cmpxchg` is used for the first externally visible write, the process may be descheduled/interrupted before the second write. The first write will still be globally visible and would lead to an inconsistent list if another insert occurred.

II Soft updates

The pseudocode for the flusher described in the soft-updates paper is as follows (b is a block):

```
flushblock (b)
{
  lock b;
  for all dependencies that b depends on
    "remove" that dependency by undoing the change to b
    mark the dependency as "unrolled"
  write b
}

write_completed (b) {
  remove dependencies that depend on b.
  reapply "unrolled" dependencies that b depended on
  unlock b
}
```

Ben proposes another implementation. Instead of unrolling dependencies, Ben's implementation removes the dependencies by first flushing the block b' that b depends on and the blocks that b' depends on:

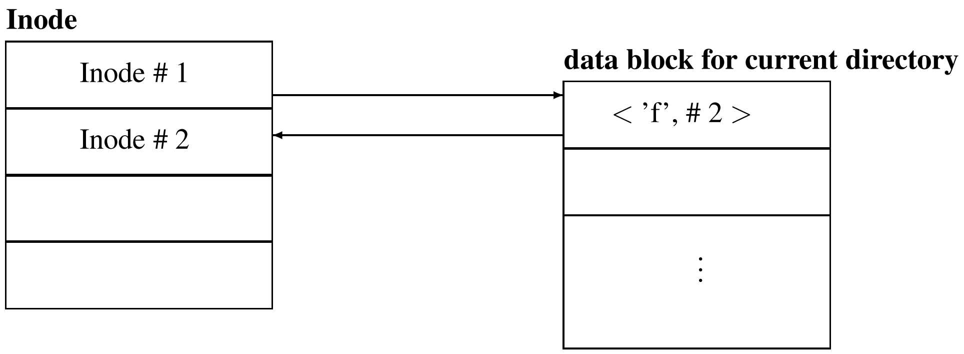
```
flushblock (b)
{
  lock b;
  for each block b′ that b depends on
      flushblock (b′);
  write b
}

write_completed (b) {
  unlock b
}
```

5. [5 points]: Draw a set of inode and data blocks and their dependencies that show that Ben's code is wrong. Why is Ben's code wrong?

Dependencies exist in the filesystem at a finer granularity than blocks. For example, multiple directory entries exist in a single block. As such, a circular dependency can exist between two blocks. However, since the flusher can only flush entire blocks, the presence of a circular dependency will result in an infinite loop in Ben's code.

One possible picture is taken from the paper (Figure 1). An alternative is given below:



6. [5 points]: List a sequence of system calls that can lead to the drawing that you provided as an answer to the previous question.

Assume a process executes the following calls in a single empty directory d:

```
creat('f'); // directory data block depends on file inode block
            // directory inode block depends on directory data block
```

Suppose f and d have inodes on the same block.

Ben proposes yet another implementation. This implementation removes the dependencies by first writing (instead of flushing) the block that b is depending on:

```
flushblock (b)
{
  lock b;
  for each block b' that b depends on
      write (b');
  write b
}

write_completed (b) {
  unlock b
}
```

7. [5 points]: Give a timeline of system calls and failures that results in incorrect behavior with this last implementation.

Any sequence of events where there is a circular dependency will result in such a case. Imagine:

```
user:   creat('a') // as above
kernel: flush (b)  // b is inode block of 'a' and cwd
          write b' // b' is data block of cwd
       CRASH
         write b   // never happens!
```

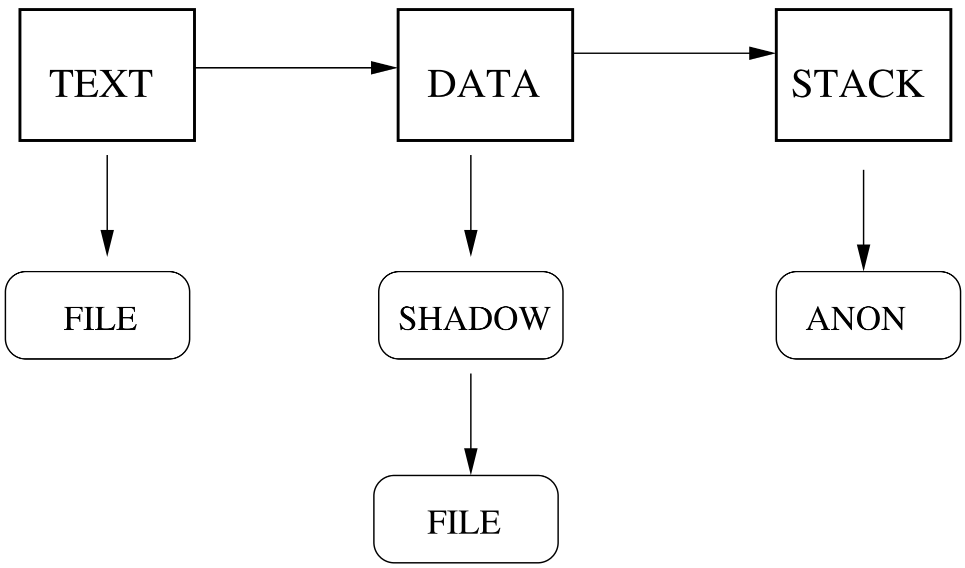
The block b' with its data that depends on b will be incorrectly written on disk.

III Mach VM system

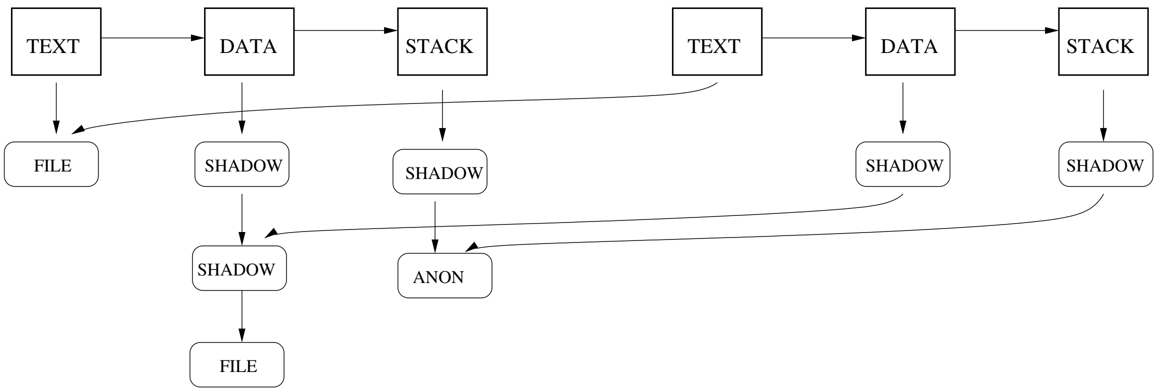
Assume a UNIX process that is running in an address space with 3 objects: one object for the read-only text of the program "a.out", one read-write, private data object (initialized from "a.out"), and one object for a read/write stack.

8. [10 points]: Draw a picture that shows the **vm_map_entries** for this process, and to which objects they are pointing. Give a brief justification for your picture.

(Hint: include the shadow objects)



9. [10 points]: Now "a.out" forks. Draw a picture that shows the **vm_map_entries** for both processes, and to which objects they are pointing. Give a brief justification for your picture.



IV Virtualizing traps and interrupts in the x86

In a hosted virtual machine monitor, a *host operating system* serves runs the VMM as an application (process) with full privileges (e.g. ring 0 on the x86). The VMM then runs the *guest operating system* inside a virtual machine. In order to do so successfully, the VMM must employ a variety of techniques to prevent the guest from discerning that it is in a virtual machine.

Ben is working on a hosted x86 virtualizer but he can't figure out how to virtualize interrupts and traps. Can you come up with a solution? **(Hint: read all questions in this section first to better understand the issues involved.)**

10. [5 points]: What IDT should the CPU use when it is executing instructions of the guest operating system? Is it the IDT of the host, the guest, or something else constructed by the VMM? Explain.

Here is one possible answer. We assume that there is no special support in the kernel besides allowing the VMM to load code that will run at ring 0. (Another possibility might be to have an exo-kernel style host operating system that allows user programs to hook into receiving notification about generic system traps.)

While the guest operating system is running, the VMM will have installed a special IDT installed that is neither the host or the guest's. This is necessary to allow the VMM to handle CPU virtualization related exceptions correctly.

11. [5 points]: What state should the VMM maintain in order to virtualize interrupts and traps? When is this state updated?

In addition to the general CPU state that must be maintained for regular CPU virtualization (e.g. general purpose registers), the VMM must maintain state about both the host operating system IDT, which it will save when it switches out the IDT before starting the guest OS, and the guest OS's desired IDT, which it will save when the guest runs `lidt`. In the case of the guest, the VMM should probably only keep a pointer to where the guest is keeping to IDT in order to avoid having to trap when the guest writes to the page where the IDT is stored.

12. [5 points]: Interrupts can be generated by hardware or by instructions that are being executed by the guest. How should the VMM handle hardware interrupts? When does the guest operating system receive an interrupt? **(Hint: The VMM relies on the host for actual device driver support.)**

Under this scheme, the VMM should immediately forward all hardware interrupts back to the host operating system. The trap frame will allow the VMM to quickly save the guest operating system's basic state, and then the other CPU state should be saved as well. Then the VMM should place the trap on the host operating system's TSS and jump the host OS's trap handler.

The guest operating system will only receive an interrupt when the VMM decides there is some sort of interrupt that the guest should get. For example, the VMM may choose to forward clock interrupts on to the guest operating system. For device I/O, the VMM will create interrupts for the guest when it receives notification from the host operating system that there is data to be provided to the guest.

Note! Calls to `outb` from the guest will result in traps and these will be emulated by the VMM with read/write calls to the host OS. The VMM should not attempt to determine if a particular interrupt is "for" the guest. This would only apply if a device were completely dedicated to the guest OS and the host OS was not allowed to use it at all.

13. [5 points]: How should the VMM handle traps that are generated by the guest (e.g. intentionally, by a system call or indirectly, as a result of the CPU virtualization)?

Errors like divide by zero or break point interrupts should be directly forwarded back into the guest operating system, using knowledge of the guest's IDT.

CPU virtualization traps and system calls will appear as GPFs since the VMM IDT is configured not to allow the `int` instruction to generate any interrupts from ring 3 or execute any privilegd instructions. The error code and instruction will allow the VMM to decide which of these two cases occurred. If the problem is the result of CPU virtualization (e.g. `lidt` executed in guest), the VMM must emulate the instruction and then continue. Otherwise, the VMM should simply forward the trap on to the guest.

End of Quiz II—-Thanks and enjoy a well-deserved break!