Chapter 1

1.1 Explain how the notion of interrupt facilitated the implementation of time sharing.

1.5 Explain the difference between an interrupt and a trap.
Chapter 3

3.5 Many systems use the clock interrupt handler to determine if the current thread’s time slice is over and, if so, force the thread to yield the processor to another thread. If the system’s architecture uses the current thread’s kernel stack to handle interrupts then the current thread’s context was saved on its kernel stack when the interrupt occurred and, even though the system is running in the interrupt context, the interrupt handler might simply call switch to switch to a different thread.

a. On architectures that have separate interrupt stacks, such as the DEC VAX-11, it does not wait for an interrupt handler to call switch directly. Explain why not.
b. Explain what might be done in such an architecture for an interrupt handler to force a thread to call switch before normal execution resumes in the thread.
4.2 Ideally, data coming from the communications network should be deposited by the network hardware directly into the physical memory locations used by network applications, and outgoing data from an application should go directly into the physical memory locations from which network hardware will retrieve it for transmission. In other words, incoming and outgoing network data should never be copied.

a. Explain why such a zero-copy approach is not possible with the socket-buffer approach described in Section 4.1.2.2.

b. Describe how the socket-buffer approach might be adapted to make possible the zero-copy approach.

4.8 Assume that in a particular computer architecture there is a special register, accessible only in privileged mode, pointing to an interrupt vector containing the addresses of the routines that are to handle each of the various types of interrupts known to the architecture. For example, if an unmasked interrupt of type i occurs, then the current processor context is pushed onto the current kernel stack and control transfers to the routine whose address is in the ith component of the vector pointed to by the register.

We would like to virtualize the architecture. Suppose a (real) interrupt occurs while a virtual machine is running.

a. In which stack is the processor's context saved?

b. Suppose the VMM decides that the interrupt should be handled by a virtual machine. Explain how it makes this happen?
5.4

The following code is an alternative to the implementation of mutexes given in Section 5.1.2. Does it work? Explain why or why not.

```c
kmutex_lock(mutex_t *mut) {
    if (mut->locked) {
        enqueue(mut->wait_queue, CurrentThread);
        thread_switch();
    }
    mut->locked = 1;
}
kmutex_unlock(mutex_t *mut) {
    mut->locked = 0;
    if (!queue_empty(mut->wait_queue))
        enqueue(RunQueue, dequeue(mut->wait_queue));
}
```

5.8

We have a new architecture for interrupt handling. There are $n$ possible sources of interrupts. A bit vector is used to mask them: if bit $i$ is 1, then interrupt source $i$ is masked. The operating system employs $n$ threads to handle interrupts, one per interrupt source. When interrupt $i$ occurs, thread $i$ handles it and interrupt source $i$ is automatically masked. When the thread completes handling of the interrupt, interrupt source $i$ is unmasked. Thus if interrupt source $i$ attempts to send an interrupt while a previous interrupt from $i$ is being handled, the new interrupt is masked until the handling of the previous one is completed. In other words, each interrupt thread handles one interrupt at a time.

Threads are scheduled using a simple priority-based scheduler. It maintains a list of runnable threads (the exact data structure is not important for this problem). There’s a global variable `CurrentThread` that refers to the currently running thread.
11. An operating system has a simple round-robin scheduler used in conjunction with time slicing: when a thread’s time slice is over, it goes to the end of the run queue and the next thread runs. The run queue is implemented as a singly linked list of threads, with pointers to the first and last threads in the queue. Assume for parts a and b that we have a uniprocessor system.

a. The system has a mix of long-running compute threads that rarely block and interactive threads that spend most of their time blocked, waiting for keyboard input, then have very brief bursts of using the processor. Assuming we want the system to have good interactive response, explain what is wrong with the scheduler.

b. How might the scheduler be improved to provide good interactive response? (Hint: a simple improvement is sufficient.)

c. We add three more processors to our system and add the appropriate synchronization (spin locks) to our scheduler data structures. Describe the performance problems that will arise.

d. Describe what might be done to alleviate these performance problems, yet still have reasonable parallelism.

14. When an interrupt occurs, on which stack should the registers of the interrupted thread be saved? Explain. (Hint: there are two possibilities: the stack of the interrupted thread and the stack of the interrupt-handling thread.)

b. After the registers are saved, what further actions are necessary so that the interrupt-handling thread can be handled by the scheduler? (Hint: consider the scheduler’s data structures.)

c. Recall that Windows employs DPCs (deferred procedure calls) so that interrupt handlers may have work done when there is no other interrupt handling to be done. How could this be done in the new architecture? (Hint: it’s easily handled.)

d. If there are multiple threads at the same priority, we’d like their execution to be time-sliced — each runs for a certain period of time, then yields to the next. In Windows, this is done by the clock interrupt handler’s requesting a DPC, which forces the current thread to yield to the processor. Explain how such time-slicing can be done on the new architecture.

14. In hierarchical stride scheduling, whenever a new thread joins a group, the total number of tickets held by the group increases and thus so does that group’s collective share of processor time. A better approach might be to give each group a fixed number of tickets to be readjust each member thread’s bribe whenever a new thread joins the group. Describe how we might modify hierarchical stride scheduling so that each group’s share of processor time remains constant despite the addition or deletion of group members, and that such addition and deletion is done in constant time (not counting the time to update the balanced tree).
Chapter 6 File Systems

6.2 Explain why the typical transfer rate achieved on S5FS is far less than the maximum transfer rate of the disk.

6.4 Assume we’re using the Rhinopias disk drive.
   a. Suppose we are using S5FS and we’ve doubled the block size from 512 bytes to 1024 bytes. Is the maximum expected transfer rate roughly doubled? Explain.
   b. We’ve done the same with FFS. Is the maximum expected transfer rate roughly doubled? Explain. Assume that we’re using two-way block interleaving: when we allocate successive blocks on a track, one block is skipped. Thus, in the best case, every other block of a track is allocated to a file.
   c. Why should we care about the block size in FFS?

6.5 Explain what is meant by innocuous inconsistency. How does it differ from non-innocuous or real inconsistency?

6.6 The layout of disk blocks is an important factor in file-system performance. FFS uses the technique of block interleaving to reduce rotational delays. This technique provides a large improvement over the use of consecutive blocks. However, many of today’s most advanced file systems, such as WAFL and ZFS, do not use block interleaving.
   a. Explain how block interleaving helps FFS reduce the time required to read from disk.
   b. Explain how WAFL’s design provides fast access without the need for block interleaving.

6.10 One early application of shadow paging in a file system was in an FFS-like system. All operations between open and close of a file were treated as part of a single transaction. Thus if the system crashed before the file was closed, the effect was as if nothing had happened to the file since the open. This was done by making a copy of the file’s inode when it was opened, and applying all changes to copies of blocks being modified. When the file was closed, the reference to its inode in the directory it appeared in was replaced with a reference to its inode in the directory it appeared in was replaced with a reference to the new, modified inode. Thus in a single disk write, all modifications took effect at once.

There are a few problems with this approach. It doesn’t handle files that appear in multiple directories. It also doesn’t handle concurrent access to the file by multiple processes. Let’s ignore both these problems. There’s a performance problem: the original blocks of the file were allocated on disk so that the file could be accessed sequentially very quickly. But when we make copies of the disk blocks being modified, these copies aren’t necessarily going to be near the original copies. Thus writing these blocks might incur long seek delays. For this reason shadow paging wasn’t used in file systems until relatively recently. How might these problems be fixed?
6.11 NTFS does a combination of both redo and undo journaling. Why is redo journaling also done?

6.17 RAID level 4, in handling a one-block write request, must write to both a data disk and the check disk. However, it need not read data blocks from the other data disks in order to recompute the check block. Explain why not.

6.18 We have an operating system whose only file system is S5FS. Suppose we decide to employ a RAID level 4 or RAID level 5 disk system. Can S5FS take advantage of all the features of these systems. Explain. Assume that the block size of S5FS is equal to the disk sector size.