SOLUTIONS TO CHAPTER 4 PROBLEMS

1. You can go up and down the tree as often as you want using "..". Some of the many paths are

```
/etc/passwd
/./etc/passwd
/./../etc/passwd
/./../../etc/passwd
/etc/../etc/passwd
/etc/../etc/../etc/passwd
/etc/../etc/../etc/../etc/passwd
/etc/../etc/../etc/../etc/../etc/passwd
```

2. The Windows way is to use the file extension. Each extension corresponds to a file type and to some program that handles that type. Another way is to remember which program created the file and run that program. The Macintosh works this way.

3. These systems loaded the program directly in memory and began executing at word 0, which was the magic number. To avoid trying to execute the header as code, the magic number was a BRANCH instruction with a target address just above the header. In this way it was possible to read the binary file directly into the new process' address space and run it at 0, without even knowing how big the header was.

4. To start with, if there were no open, on every read it would be necessary to specify the name of the file to be opened. The system would then have to fetch the i-node for it, although that could be cached. One issue that quickly arises is when to flush the i-node back to disk. It could time out, however. It would be a bit clumsy, but it might work.

5. No. If you want to read the file again, just randomly access byte 0.

6. Yes. The rename call does not change the creation time or the time of last modification, but creating a new file causes it to get the current time as both the creation time and the time of last modification. Also, if the disk is nearly full, the copy might fail.

7. The mapped portion of the file must start at a page boundary and be an integral number of pages in length. Each mapped page uses the file itself as backing store. Unmapped memory uses a scratch file or partition as backing store.

8. Use file names such as */usr/ast/file*. While it looks like a hierarchical path name, it is really just a single name containing embedded slashes.

9. One way is to add an extra parameter to the read system call that tells what address to read from. In effect, every read then has a potential for doing a seek within the file. The disadvantages of this scheme are (1) an extra parameter in every read call, and (2) requiring the user to keep track of where the file pointer is.

10. The dotdot component moves the search to */usr*, so *../ast* puts it in */usr/ast*. Thus *../ast/x* is the same as */usr/ast/x*.

11. Since the wasted storage is *between* the allocation units (files), not inside them, this is external fragmentation. It is precisely analogous to the external fragmentation of main memory that occurs with a swapping system or a system using pure segmentation.

12. If a data block gets corrupted in a contiguous allocation system, then only this block is affected; the remainder of the file's blocks can be read. In the case of linked allocation, the corrupted block cannot be read; also, location data about all blocks starting from this corrupted block is lost. In case of indexed allocation, only the corrupted data block is affected.

13. It takes 9 msec to start the transfer. To read 2^{13} bytes at a transfer rate of 80 MB/sec requires 0.0977 msec, for a total of 9.0977 msec. Writing it back takes another 9.0977 msec. Thus, copying a file takes 18.1954 msec. To compact half of a 16-GB disk would involve copying 8 GB of storage, which is 2^{20} files. At 18.1954 msec per file, this takes 19,079.25 sec, which is 5.3 hours. Clearly, compacting the disk after every file removal is not a great idea.

14. If done right, yes. While compacting, each file should be organized so that all of its blocks are consecutive, for fast access. Windows has a program that defragments and reorganizes the disk. Users are encouraged to run it periodically to improve system performance. But given how long it takes, running once a month might be a good frequency.

15. A digital still camera records some number of photographs in sequence on a nonvolatile storage medium (e.g., flash memory). When the camera is reset, the medium is emptied. Thereafter, pictures are recorded one at a time in sequence until the medium is full, at which time they are uploaded to a hard disk. For this application, a contiguous file system inside the camera (e.g., on the picture storage medium) is ideal.

16. The indirect block can hold 128 disk addresses. Together with the 10 direct disk addresses, the maximum file has 138 blocks. Since each block is 1 KB, the largest file is 138 KB.

17. For random access, table/indexed and contiguous will be both appropriate, while linked allocation is not as it typically requires multiple disk reads for a given record.

18. Since the file size changes a lot, contiguous allocation will be inefficient requiring reallocation of disk space as the file grows in size and compaction of free blocks as the file shrinks in size. Both linked and table/indexed allocation will be efficient; between the two, table/indexed allocation will be more efficient for random-access scenarios.

19. There must be a way to signal that the address-block pointers hold data, rather than pointers. If there is a bit left over somewhere among the attributes, it can be used. This leaves all nine pointers for data. If the pointers are k bytes each, the stored file could be up to $9k$ bytes long. If no bit is left over among the attributes, the first disk address can hold an invalid address to mark the following bytes as data rather than pointers. In that case, the maximum file is $8k$ bytes.

20. Elinor is right. Having two copies of the i-node in the table at the same time is a disaster, unless both are read only. The worst case is when both are being updated simultaneously. When the i-nodes are written back to the disk, whichever one gets written last will erase the changes made by the other one, and disk blocks will be lost.

21. Hard links do not require any extra disk space, just a counter in the i-node to keep track of how many there are. Symbolic links need space to store the name of the file pointed to. Symbolic links can point to files on other machines, even over the Internet. Hard links are restricted to pointing to files within their own partition.

22. A single i-node is pointed to by all directory entries of hard links for a given file. In the case of soft-links, a new i-node is created for the soft link and this inode essentially points to the original file being linked.

23. The number of blocks on the disk = 4 TB / 4 KB = $2^{30.}$ Thus, each block address can be 32 bits (4 bytes), the nearest power of 2. Thus, each block can store 4 KB / 4 = 1024 addresses.

24. The bitmap requires B bits. The free list requires DF bits. The free list requires fewer bits if $DF < B$. Alternatively, the free list is shorter if $F/B < 1/D$, where F/B is the fraction of blocks free. For 16-bit disk addresses, the free list is shorter if 6% or less of the disk is free.

25. The beginning of the bitmap looks like:

(a) After writing file *B*: 1111 1111 1111 0000
(b) After deleting file *A*: 1000 0001 1111 0000
(c) After writing file *C*: 1111 1111 1111 1100
(d) After deleting file *B*: 1111 1110 0000 1100

26. It is not a serious problem at all. Repair is straightforward; it just takes time. The recovery algorithm is to make a list of all the blocks in all the files and take the complement as the new free list. In UNIX this can be done by scanning all the i-nodes. In the FAT file system, the problem cannot occur because there is no free list. But even if there were, all that would have to be done to recover it is to scan the FAT looking for free entries.

27. Ollie's thesis may not be backed up as reliably as he might wish. A backup program may pass over a file that is currently open for writing, as the state of the data in such a file may be indeterminate.

28. They must keep track of the time of the last dump in a file on disk. At every dump, an entry is appended to this file. At dump time, the file is read and the time of the last entry noted. Any file changed since that time is dumped.

29. In (a) and (b), 21 would not be marked. In (c), there would be no change. In (d), 21 would not be marked.

30. Many UNIX files are short. If the entire file fits in the same block as the i-node, only one disk access would be needed to read the file, instead of two, as is presently the case. Even for longer files there would be a gain, since one fewer disk accesses would be needed.

31. It should not happen, but due to a bug somewhere it could happen. It means that some block occurs in two files and also twice in the free list. The first step in repairing the error is to remove both copies from the free list. Next a free block has to be acquired and the contents of the sick block copied there. Finally, the occurrence of the block in one of the files should be changed to refer to the newly acquired copy of the block. At this point the system is once again consistent.

32. The time needed is $h + 40 \times (1 - h)$. The plot is just a straight line.

33. In this case, it is better to use a write-through cache since it writes data to the hard drive while also updating the cache. This will ensure that the updated file is always on the external hard drive even if the user accidentally removes the hard drive before disk sync is completed.

34. The block read-ahead technique reads blocks sequentially, ahead of their use, in order to improve performance. In this application, the records will likely not be accessed sequentially since the user can input any student ID at a given instant. Thus, the read-ahead technique will not be very useful in this scenario.

35. The blocks allotted to f1 are: 22, 19, 15, 17, 21.
The blocks allotted to f2 are: 16, 23, 14, 18, 20.

36. At 15,000 rpm, the disk takes 4 msec to go around once. The average access time (in msec) to read k bytes is then $6 + 2 + (k/1,048,576) \times 4$. For blocks of 1 KB, 2 KB, and 4 KB, the access times are about 6.0039 msec, 6.0078 msec, and 6.0156 msec, respectively (hardly any different). These give rates of about 170.556 KB/sec, 340.890 KB/sec, and 680.896 KB/sec, respectively.

37. If all files were 1 KB, then each 4-KB block would contain one file and 3 KB of wasted space. Trying to put two files in a block is not allowed because the unit used to keep track of data is the block, not the semiblock. This leads to 75% wasted space. In practice, every file system has large files as well as many small ones, and these files use the disk much more efficiently. For example, a 32,769-byte file would use 9 disk blocks for storage, given a space efficiency of 32,769/36,864, which is about 89%.

38. The indirect block can hold 1024 addresses. Added to the 10 direct addresses, there are 1034 addresses in all. Since each one points to a 4-KB disk block, the largest file is 4,235,264 bytes

39. It constrains the sum of all the file lengths to being no larger than the disk. This is not a very serious constraint. If the files were collectively larger than the disk, there would be no place to store all of them on the disk.

40. The i-node holds 10 pointers. The single indirect block holds 1024 pointers. The double indirect block is good for 1024^2 pointers. The triple indirect block is good for 1024^3 pointers. Adding these up, we get a maximum file size of 1,074,791,434 blocks, which is about 16.06 GB.

41. The following disk reads are needed:

directory for /
i-node for */usr*
directory for */usr*
i-node for */usr/ast*
directory for */usr/ast*
i-node for */usr/ast/courses*
directory for */usr/ast/courses*
i-node for */usr/ast/courses/os*
directory for */usr/ast/courses/os*
i-node for */usr/ast/courses/os/handout.t*

In total, 10 disk reads are required.

42. Some pros are as follows. First, no disk space is wasted on unused i-nodes. Second, it is not possible to run out of i-nodes. Third, less disk movement is needed since the i-node and the initial data can be read in one operation. Some cons are as follows. First, directory entries will now need a 32-bit disk address instead of a 16-bit i-node number. Second, an entire disk will be used even for

files which contain no data (empty files, device files). Third, file-system integrity checks will be slower because of the need to read an entire block for each i-node and because i-nodes will be scattered all over the disk. Fourth, files whose size has been carefully designed to fit the block size will no longer fit the block size due to the i-node, messing up performance.