11.10 The open-file table is used to maintain information about files that are currently open. Should the operating system maintain a separate file table for each user or maintain just one table that contains references to files that are currently being accessed by all users? If the same file is being accessed by two different programs or users, should there be separate entities in the open-file table? Explain.

Given the scenario that multiple users are modifying a file, we need a way to know that this is the case, that way if one user deletes a file for example the operating system will know another user is still accessing it and will not remove it from the disk until the other user is done. For this reason a central open-file table is necessary otherwise we would not know how many users are currently accessing a file.

If two processes are modifying the same file, it is likely that they are currently modifying two different parts of the same file. Thus we will need two separate entries in the open-file table to keep track of what exactly each user is doing to the file.

11.14 If the operating system knew that a certain application was going to access file data in a sequential manner, how could it exploit this information to improve performance.

The operating system could employ a prefetch approach to file access. When the application requests a block of a file, the operating system knows that subsequent blocks in the file are likely to be accessed next and can thus load those subsequent blocks into memory before the application actually needs them.

If this information is known sufficiently ahead of time, it may be worth the operating systems effort to keep blocks as close together on disk as possible. This will minimize read/write times of these files as the disk head will not have to jump around the disk.

11.17 Some systems provide file sharing by maintaining a single copy of a file. Other systems maintain several copies, one for each of the users sharing the file. Discuss the relative merits of each approach.

By maintaining only one copy of the file, you risk corrupting the file if multiple writes are performed simultaneously by different users. Which could possibly lose the changes being made. This is not a problem if you maintain multiple copies as you can cross reference the differences upon completion of writes. However by taking the latter approach the multiple copies will require more disk space to store the data.

12.10 Contrast the performance of the three techniques for allocating disk blocks (contiguous, linked, and indexed) for both sequential and random file access.

---

contiguous This method is obviously very weak to external fragmentation, but since all of memory is stored in one contiguous block, sequential read is as easy as stepping through the file. Random access can also be easily calculated by a single offset from the base address of the file. No need to jump around memory following different blocks. So in this context, contiguous actually does quite well in both scenarios.

linked Linked sequential works reasonably well, as you read the file and reach the end of the block you only need follow the link and can continue reading. However random access is not so good, you cannot compute a direct memory address of where you want to go and instead must essentially read the entire file and jump around the disk to follow the links, thus linked random access will not perform particularly well.

indexed Index requires that you have your index table in memory, but assuming that is the case, it is easy to reference the index table while sequentially reading each block, once you reach the end of the block you can reference the table in memory to tell you where to go next. In this sense sequential access is similar to the behavior of linked. However the difference comes with indexed random access, with the access to the table and references to all blocks, you need only calculate which block the data you desire is in, look it up in the index table, and can jump straight to that location in memory. Thus, unlike linked you will actually perform quite well as all of your references are stored in one place and not scattered throughout memory.

12.15 Consider a file system on a disk that has both logical and physical block size of 512 bytes. Assume that the information about each file is already in memory. For each of the three allocation strategies (contiguous, linked, and indexed), answer the following questions.

(a) How is the logical-to-physical address mapping accomplished in this system? (for indexed allocation, assume that a file is always less than 512 blocks long.)
contiguous Block number is obtained by dividing the logical address by 512 then adding the starting block number of the file. Block offset is obtained by the remainder of that division.

linked We only store 511 bytes of data (extra byte is the link). Let V be the result of dividing the logical address by 512, this is how many blocks we need to traverse following their references starting at the starting block number of the file. The block we end up on is the physical block number. The offset into this block is the remainder of the logical address divided by 511 plus 1 to account for the link at the start of the block.

indexed Assuming the index table is in memory, we divide the logical address by 512 to get an index into the index table, the value at that index is the block number. Like contiguous memory, the block offset is obtained by finding the remainder of dividing the logical address by 512.

(b) If we are currently at logical block 10 (the last block accessed was block 10) and want to access logical block 4, how many physical blocks must be read from the disk.
contiguous 1. We can jump straight to the correct block using the computation above.

linked 4. We will have to jump to the first block in the file, and then start following links from there.

indexed 2. One for the index table (if not already in memory, else only 1). Then one more to jump to the correct block found by the lookup in the index table.

12.17 Fragmentation on a storage device can be eliminated by recompaction of the information. Typical disk devices do not have relocation or base registers (such as those used when memory is compacted), so how can we relocate files? Give three reasons why recompacting and relocation are often avoided.

File blocks will need to be read into memory, in order to be swapped, and then written back to their new location. Relocation registers are not necessary because most disk files are not sequentially stored. This will often be avoided first and foremost because it is incredibly expensive, disks are slow and we will have to read and write to/from every used block in the system. If we are using blocks files need not be stored sequentially, so we may not even need to perform this operation. New files can still be allocated by distributing them through available blocks (we have internal fragmentation, not external).