CS33 Homework Assignment 8 Solutions

Fall 2019

1. Information about files is stored in the following data structures (see Lecture 19):
   i. directory entry
   ii. file-descriptor table
   iii. file-context structure
   iv. inode

   The directory entry and the inode are stored on disk and brought into memory when necessary; the other two exist only in memory. We know that a single file might be referenced by multiple directory entries as well as be open in multiple processes, perhaps multiple times in some of these processes. Access permissions (the permission vectors of slides XX-15 and XX-16) are checked just when a file is opened, and thus not on each *read* or *write* of the file.

   a. In which of the data structures does it make sense to store the size of a file (i.e., the byte offset of the location just after the last byte of the file)?

      Answer: in the inode.

   b. In which of the data structures does it make sense to store the permission vector?

      Answer: in the inode.

   c. When a *read* or *write* system call is executed, the location in the file at which the system call will take place is taken from the file-location field of the file-context structure. One option of the *open* system call is O_APPEND, which specifies that *write*s to the file always take place at the current end of the file. Explain how this is made to work, in terms of the data structures listed above.

      Answer: the file location at which the write takes place is taken from the size field in the inode; the file-location field (and the size field of the inode) is updated with the offset of the new end of the file. Note that care must be taken if there are multiple writes to the file at once: the OS guarantees that appends to file always take place at the current end.

   d. A directory is represented just like any other file, except that its contents are interpreted specially. In which of the above data structures is it indicated that a file is a directory?

      Answer: in the inode.

2. Unix's I/O system calls, in particular *read* and *write*, were designed with the idea that the vast majority of I/O is sequential, i.e., starting at the beginning of a file and proceeding in order to the end. However, it is possible to do non-sequential (or random) I/O through the use of the *lseek*

system call. This function simply sets the file location in the file-context structure, thus affecting where the next read or write takes place.

A relatively new pair of Unix system calls is *pread* and *pwrite*, which work just like *read* and *write*, except that the file location is specified as an argument and the value in the file-context structure is ignored. This might be considered merely an optimization, allowing random I/O to be done in a single system call rather than two, but it turns out to be necessary in certain situations. What are these situations?

Answer: Consider the case in which a process inherits an open file after a fork. As explained in lecture the new process shares the file-context structure with its creator. Thus if both are doing random I/O at the same time, there will be a conflict in their use of the file-location field. One process might set the file location to, say, 16, but before it gets a chance to do a read, the other has set it 1036.

This problem is exacerbated in the case of multi-threaded processes (which we take up near the end of the semester) in which multiple threads of control can be running within a single process.