OS4-FILES: FILEDESC-REDIRECTION-BUFFERING (91-120)

os4-fd:

91 [T / F] 1KB is 1000 bytes.

92 [T / F] The first byte of a file is always called "file offset 0".

93 [T / F] Every process has its own file descriptor (FD) table.

94 [T / F] Threads of the same process share the same file descriptor table.

95 [T / F] When you call "close(stdout)", it will return an error because the "screen" cannot be closed.

96 [T / F] Every open() creates a new FD structure pointed by an unused FD entry with the smallest number.

97 [T / F] Opening the same file two times in the same process will only use one FD entry in the FD table.

98 [T / F] Every FD structure has an "offset" (current position) information, identifying the next starting offset to be read when the user calls read().

99 [T / F] Inside an FD, there is a separate offset information for read and write (*e.g.*, readOffset and writeOffset).

100 [T / F] File descriptors only describe "open status", but not the static file information (file metadata).

101 [T / F] "File" is a hardware concept (*e.g.*, the disk knows for every byte/block on the disk, which file owns that area and who owns the data).

102 [T / F] In a simplest definition, the OS divides the disk into several areas (the metadata and the actual data location).

103 [T / F] Two processes are not allowed to open the same existing file.

104 [T / F] read(N bytes), when successfully read N bytes, will move the current position (offset) by N (offset = offset + N).

os4-redirection:

105 [T / F] Now I understand that by separating fork() and execve(), the shell can manipulate the I/O redirection before the new program is executed.

106 [T / F] dup2(7,13), assuming both entries are valid, will make the OS copy the pointer in FD entry #13 to overwrite the FD entry #7.

107 [T / F] After a successful dup2(A,B) call, there will be an FD structure that is pointed by both entries A and B. The OS will increment the reference count of that FD structure by one.

108 [T / F] newFd=dup(A) is similar to dup2(), but instead of you providing the destination FD entry, the OS will copy FD entry A (*i.e.*, the pointer! NOT the FD structure) to a new FD entry number X and return X as the new FD.

109 [T / F] After forking, the child process shares the same FD table as its parent.

110 [T / F] sleep(N) means sleep for N seconds. This is just a lazy way this professor "removes" non-determinism, but it's not the correct way.

111 [T / F] Opening a file in fd=open(...,OAPPEND) mode will cause the write(fd,...) system call to write the new content to the end of the file (defined by the file size at the moment of the write).

112 [T / F] When a process P opens a file F is opened in OAPPEND mode, there is no way that P can modify the pre-existing content of file F within the same process.

113 [T / F] I promise I'll play the "fun with file descriptors" game multiple times, including the ones with APPEND and lseek(). I promise that in this game, I will *draw a diagram/sketch* of the scenario; the FD tables, their pointers to FD structures, the current offsets, the file content, etc. just like what we did in class. Based on my sketch, I will try to answer the question.

os4-buffering:

114 [T / F] When calling read() and write(), always check the return values for shortcounts and errors for correctness.

115 [T / F] Buffering is a prevalent concept, it's "everywhere," not just in C libraries but also in many applications like Microsoft Office and browsers, and disks, OS, etc.

116 [T / F] The concept of buffering or caching is basically to reduce the number of expensive calls (just like having a refrigerator instead of going to grocery every few hours).

117 [T / F] This is again why you should use libraries (C/Java/etc. libraries) because libraries have more user-level optimizations on top of the existing syscalls.

118 [T / F] With buffering, a larger buffer (but not unnecessarily too big) in general will result in a better performance.

119 [T / F] ... 119

120 [T / F] ... 120