QUESTIONS

1. [1] Why does a different version of stat(), lstat(), exist that treats symbolic links differently? Why isn't a different version needed for hard links?

Answer: For most directory-tree-traversing programs, including findbin, it's important to use lstat(), as follows.

For the most part, if you attempt to access a symbolic link, the kernel *follows* this symbolic link automatically, giving you instead the file that the symlink points to. If this weren't the case, then symlinks wouldn't mean what they do mean. A symlink is a stand-in for the pointed-to file.

But you can't have the kernel *always* following symlinks, only almost-always. For example, an ls −R, or find, would get very confused by symlinks if it called stat() rather than lstat(). In particular, if a symlink points to a parent directory, then to opendir that symlink and continue traversing from there will result in infinite recursion.

So when symlinks were introduced, a dozen or so programs needed to be modified to be able to continue to work in their presence. These days, many more programs need to be aware of symlinks. Anything which traverses a directory tree needs to treat symlinks-which-point-to-a-directory differently from directories. Programs such as "ls" need to collect information on the symlink, rather than the pointed-to file.

The way to do this is to call the special call "lstat()", which is like stat() so long as its parameter is not a symlink. If its parameter *is* a symlink, it *does not follow* the symlink, but rather, reports information about the symlink itself.

Thus for example, "ls −l" calls lstat(), not stat(). There is an option '−L' to make it follow the symlinks, but otherwise it doesn't.

For more examples: "test −f" calls stat(), but "test −L" (check whether the file is a symlink) needs to call lstat().

box has no reason to call stat() or lstat(), but if it did, it would call stat, not lstat, because we *do* want it to follow symlinks, in the normal way.

1. [1] How can you modify 3000pc so the producer stops producing once it fills the queue?

Answer: call either wait\_for\_consumer(shared \*s)or set the the

s->prod\_waiting = 1;

1. [1] Under what circumstances is fill\_rand\_buffer() called in 3000random?

Answer: fill random buffer

1. [1] If the two signal handling functions in 3000pc were replaced by one function, would there be any significant loss of functionality? Briefly explain.
2. [2] How could you modify 3000test.c so it can report on whether two device files are equal without actually accessing the underlying devices? Specify the changes you would make to 3000test.c rather than doing this from scratch.
3. [2] Does the MAP\_SHARED flag on line 60 of 3000test.c (inside the call to mmap) make a significant difference in its execution? Specifically, what happens when you remove it or changed it to MAP\_PRIVATE? Why?
4. [2] When a file is mmap'd into memory, when is its contents loaded from disk? How can you verify this using 3000test?
5. [2] What is one way you can modify 3000pc so the consumer consumes exactly as fast as the producer produces (i.e., the producer and the consumer move in lock step?) Your modification should not involve sleeping (by either the producer or the consumer).
6. [2] Which is faster, /dev/urandom or /dev/random? What evidence do you have for this difference based on code that you ran (3000random or other programs)?
7. [2] How does the behavior of 3000pc change if you delete lines 149 and 152 (the if statement in wakeup\_consumer())? Why?
8. [2] What happens if you delete line 231 (the call to wakeup\_producer()) in 3000pc? Why?

Ans: Producer will be gone to infinite Waiting time and won’t be free

Because wakeup\_producer() set the waiting time of producer to zero means producer is ready for produce.

1. [2] How does the behavior of the program change if you change QUEUESIZE to 8? What about 128?