

# CS 574

## Midterm Exam

### Solutions

The following exam is open book and open notes. You may feel free to use whatever additional reference material you wish, but **no electronic aids** are allowed. Please note the following instructions. There will be a ten point deduction for failure to comply with them:

- start each problem on a new sheet of paper
- write your social security number, but not your name, on each sheet of paper you turn in
- show your work whenever appropriate. There can be no partial credit unless I see how answers were arrived
- be succinct. You may lose points for facts that, while true, are not relevant to the question at hand

You have until 12:20 to finish the exam. The questions are equally weighted.

1. In a monolithic OS, a user process that wishes to read one byte from an external device requires at least two transitions across the user/kernel boundary: one transition when the process makes the system call, and a second one when the system call returns.

How many transitions across the user/kernel boundary are required in Minix V.3? Assume a really, really simple device in which reading one byte from a device only requires a single read of the device.

*The user process sends a message to the device driver; the message passing mechanism requires two transitions across the boundary. The device driver needs to read from the device; this has to be performed by the kernel on the driver's behalf, so this requires another pair of transitions. Finally, the driver must send a message to the user process; this takes another two messages. So the total is six. This actually assumes the user process can send a message directly to the device driver – from my reading of the paper there may need to be some more messages between the user process, the device driver, and the file system server. If that's accurate, we need another two messages for a total of ten transitions!*

*Notice that any answer with an odd number of transitions never gets back to the user process!*

Points	Description
5	Only zero or one transition(s)
10	Two, just like monolithic OS
15	Proper count of messages; no count of transitions per message
20	Four transitions – looks like driver does IO directly
5	Long discussion of Minix; never gets around to answering the question

2. Current file systems tend to be weak in handling small files. In particular, many little configuration files tend to be on systems; these files are frequently much smaller than the filesystem block size, so there's quite a bit of wasted space. Also, many directories are much smaller than the block size, leading to a similar waste of space.

Consider the possibility of a "multiple block size" file system. This would be a Unix-like file system, only instead of just having i-nodes and data blocks it would have i-nodes and several sizes of data blocks. For the sake of argument, we'll assume two block sizes: a 256 byte "small block" and a 4K "large block."

Discuss the idea. How would the system know whether a given data block pointer in an i-node pointed to a small block or a large block? Should the small blocks be all by themselves, or should they be interspersed with the large blocks? How should their used/free status be managed? Can you think of any other issues to discuss in their regard?

*This actually seems feasible to me. We could have some of the large blocks subdivided into 16 small blocks. A file with a size below some threshold could use small blocks; above that threshold, it could use large blocks. Keeping small blocks near i-nodes would help reduce access times; in a file system with several sets of i-nodes at different locations on disk we'd end up with them interspersed with large blocks, just as i-nodes are. We could use a bitmap, like the i-node and large block bitmaps, to maintain used/free status.*

3. Would start-time fair queuing be an appropriate low-level CPU and/or disk scheduler for an exokernel operating system?

*When we talked about exokernels, it was pretty apparent that they'd glossed over the thorny questions of actually allocating resources, particularly resources like disk and CPU bandwidth. SFQ has the nice feature in this sort of environment of simply divvying up the bandwidth according to some sort of priority scheme; as such, it seems like a really good match.*

4. The Linux "slab allocator" is a highly efficient memory allocator used within the Linux kernel. Since I don't expect you to be familiar with the Linux kernel, I'll take just a moment to describe the slab allocator.

The way it works is that a large number of objects, all the same size, are allocated in a block (called a "slab") located on a page boundary. This makes it possible to manage the objects within the slab by just setting and clearing "used" bits; this is a lot faster and easier than using free lists or buddy trees or something. Whenever all the objects in a slab have been allocated, a new slab is created so it's still possible to allocate new objects.

An odd thing about the objects in the slab is that they don't start at the very beginning of the slab. Instead, a random amount of space is inserted at the start of the slab, before the first object. Here's a picture of an allocated slab:



The shaded area at the top is the random offset before the first object in the slab, the next four rectangles are the objects within the slab (a real slab would have many more than this), and the little bit of space at the bottom is wasted.

What would you expect this random offset to do to the cache hit rate?

*It improves it by reducing the probability that searches through objects in different slabs will repeatedly access the same cache lines (so it reduces the number of conflict misses).*