1999 Operating System functions

p4 q7
SMH
p11 q7

`mbufs`).

2. Cacheing doesn't exist: data has a temporal aspect, and holding old versions of packets is not useful.
3. Allocation is sporadic: can't prefetch, or choose size, or whatever. Driven by the (external) network on receive.
4. Locality of reference doesn't exist (sim. to 2).
5. No "write-merge" benefit in delaying transmission.
6. May wish to treat a logical unit (e.g. packet) as a set of smaller logical units (e.g. fragments, headers, etc) ⇒ need to be able to chain things together.

2 Question 2

The following are three ways which a file-system may use to determine which disk blocks make up a given file.

a) Chaining in a map.

b) Tables of pointers.

c) Extent lists.

Briefly describe how each scheme works. [2 marks each]

Describe the ~~Give two~~ benefits and ~~two~~ drawbacks of using scheme (c). [4 marks]

~~One can protect access to files by using *access control lists* or *capability lists*. Compare and contrast these two approaches.~~ [6 marks]

You are part of a team designing a distributed filing system which replicates files for performance and fault-tolerance reasons. It is required that rights to a given file can be revoked within T milliseconds ($T \geq 0$). Describe how you would achieve this, commenting on how the value of T would influence your decision. [4 marks]

2.1 Answers to Question 2

2.1.1 Brief descriptions

Two marks each for chaining in a map, table of pointers, and extent lists.

Chaining in a Map — in this scheme we keep an array of e.g. 4-byte entries for every e.g. 4K block on the disk (partition). Each entry contains a "pointer' (usually an

index) to the next block in the chain, or else contains an "end-of-chain" marker. Directory entries map a name to the index of the first block in the file.
Typically it will not be possible to keep the entire array resident in memory; only a certain part or parts will be cached.

Table of Pointers — in this scheme we keep a array or table *per file*. The n^{th} entry in the array holds the logical block number of the n^{th} block in the file. Since the table size scales linearly with the number of blocks in the file, we typically link together a number of fixed size tables to handle large files. The resulting structure is typically a tree (although could be a list).

Extent Lists — similarly to above, keep an array or table per file. In this case, however, each entry points to an *extent* on the disk: a range of k contiguous blocks. The values of k may be different for each entry. As with the table of pointers scheme, we usually chain together fixed size tables in a tree (e.g. balanced two-level tree).

2.1.2 Pros and Cons of Extent Lists

One mark for each benefit, and one for each drawback.

Some advantages are:

- Contiguous blocks on disk (an extent) are generally faster to read or write[1]. Particularly good for continuous media data.
- Space required for meta-data is smaller since don't require an entry for every block.
- Can make smallest extent very small (e.g. h/w block size) to reduce internal fragmentation (particularly good for small files), but still not suffer poor performance/metadata overhead with large files.

Some disadvantages are:

- Translation of arbitrary offset within a file requires more than quantize, scale & index. Need to *search* the (perhaps chained) list of extents.
- If get fragmentation, may not be possible to allocate extents larger than 1 block (in which case lose all the advantages above). This is particularly bad if we have chosen a very small unit block size (to avoid internal frag as shown above).
- To avoid above (or to avoid "can't extend" if no of extents limited), may have to try to defragment the disk; takes a long time.
- If want to get advantages from extents, need to have more complex buffer cache (and/or memory mapped infrastructure).

[1]Of course one can get head switches or cylinder switches between "logically" adjacent blocks, but we normally don't get to see that detail.

There are other ways of stating the above. There are also probably further significant comparisons to be made, which should also score marks.

2.1.4 Distributed Revocation

or a sensible alternative

5 ~~3~~ *marks will be given for either of the below solutions;* ~~1 additional mark will go for (even briefly) mentioning the alternate scheme~~. Two main possibilities present themselves: the first is to use a pure capability augmented by *timeouts*. Each capability is only valid for a certain number of milliseconds (perhaps hundreds of thousands though). Once it has expired, it can be renewed so long as the permissions it grants have not been revoked. The timeout value could be encoded into the capability as e.g. an epoch value.

For example, the epoch value, a (unique?) object id, plus the set of rights an perhaps some other values could form the input to a secure hash function. For validation, the *current* epoch value could be used; if the hash fails to yield the correct bit string, it will be rejected.

One problem with this scheme is that if T is small, then capabilities will time out quickly. Hence a large amount of time may be spent in renegotiation. Furthermore, this scheme is not useful is T is 0.

An alternative scheme is to use *password capabilities* (sometime called lock-key capabilities). In this scheme the capability itself grants access to an entry in an ACL (perhaps held elsewhere). To revoke a given capability, this ACL is updated. The entries in the ACL could be a single bit denoting the validity of the associated capability, or could contain some maximum level of permissions or, even, the canonical copy of the permissions.

This allows one to revoke relatively easily, but has the disadvantage that copies of the ACL must be kept up-to-date, or else a centralised ACL must be used. If there are no copies of the object, then this situation provides excellent revocation support for any value of T. Otherwise the critical delay will be the amount of time to update the relevant ACLs and/or to access the central ACL.