Extensible files

Pintos currently cannot extend the size of files because the Pintos file system allocates each file as a single contiguous set of blocks. Your task is to modify the Pintos file system to support extending files. Your design should provide fast random accesses to the file, so you should avoid using a design based on File Allocation Tables (FAT). One possibility is to use an indexed inode structure with direct, indirect, and doubly-indirect pointers, similar to Unix FFS. The maximum file size you need to support is 8 MiB (2^23 bytes). You must also add support for a new system call `inumber(int fd)` which returns the unique inode number of file associated with a particular file descriptor. Make sure that you gracefully handle cases where the operating system runs out of memory or disk space by leaving the file system in a consistent state (especially with regard to inode extension) and without leaking disk space or memory.

Implementation details

The basic file system allocates files as a single extent, making it vulnerable to external fragmentation: it is possible that an `n`-block file cannot be allocated even though `n` blocks are free. **Eliminate this problem by modifying the on-disk inode structure.** In practice, this probably means using an index structure with direct, indirect, and doubly indirect blocks. You are welcome to choose a different scheme as long as you explain the rationale for it in your design documentation, and as long as it does not suffer from external fragmentation (as does the extent-based file system we provide).

You can assume that the file system partition will not be larger than 8 MiB. You must support files as large as the partition (minus metadata). Each inode is stored in one disk sector, limiting the number of block pointers that it can contain. Supporting 8 MiB files will require you to implement doubly-indirect blocks.

An extent-based file can only grow if it is followed by empty space, but indexed inodes make file growth possible whenever free space is available. **Implement file growth.** In the basic file system, the file size is specified when the file is created. In most modern file systems, a file is initially created with size 0 and is then expanded every time a write is made off the end of the file. Your file system must allow this.

There should be no predetermined limit on the size of a file, except that a file cannot exceed the size of the file system (minus metadata). This also applies to the root directory file, which should now be allowed to expand beyond its initial limit of 16 files.

User programs are allowed to seek beyond the current end-of-file (EOF). The seek itself does not extend the file. Writing at a position past EOF extends the file to the position being written, and any gap between the previous EOF and the start of the `write()` must be filled with zeros. A `read()` starting from a position past EOF returns no bytes.

Writing far beyond EOF can cause many blocks to be entirely zero. Some file systems allocate and write real data blocks for these implicitly zeroed blocks. Other file systems do not allocate these blocks at all until they are explicitly written. The latter file systems are said to support “sparse files." You may adopt either allocation strategy in your file system.

You are already familiar with handling memory exhaustion in C, by checking for a NULL return value from malloc. In this project, you will also need to handle disk space exhaustion. When your file system is unable to allocate new disk blocks, you must have a strategy to abort the current operation and rollback to a previous good state.
