JUN-YI LAU | 49002253

PINTOS PROJECT : FILESYSTEM

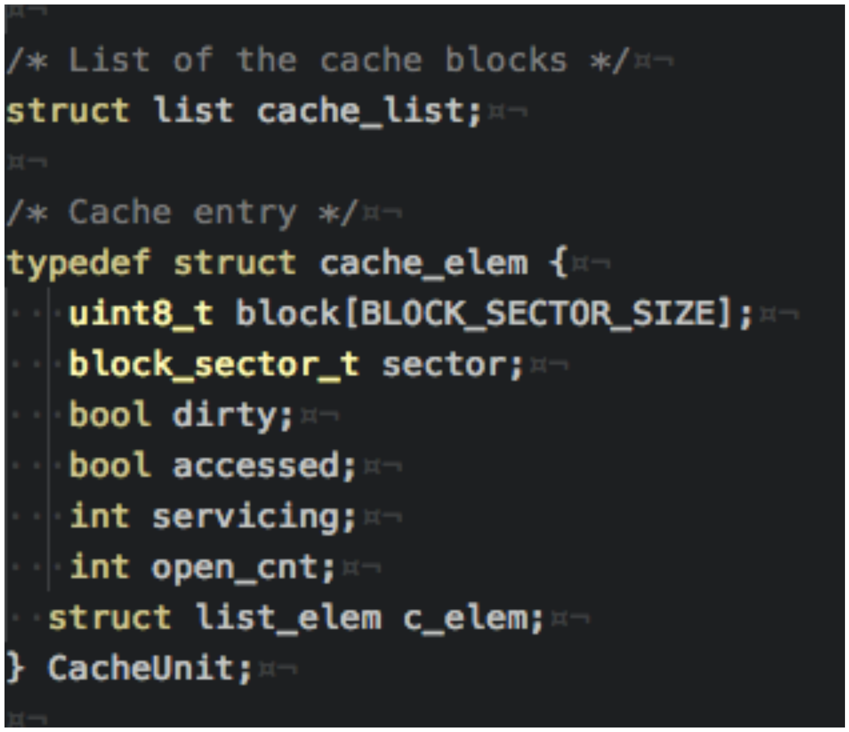
Cache Buffer

Problem:

All file accesses in Pintos are disk accesses which are slow. We want to make more disk accesses memory accesses for improved performance.

Data Structure:

Cache Buffer data structure in **cache.c**



```
/* List of the cache blocks */
struct list cache_list;

/* Cache entry */
typedef struct cache_elem {
    uint8_t block[BLOCK_SECTOR_SIZE];
    block_sector_t sector;
    bool dirty;
    bool accessed;
    int servicing;
    int open_cnt;
  struct list_elem c_elem;
} CacheUnit;
```

Solution:

The cache buffer provides the **cache_get_block ()** function. Given a block sector, it will return a **CacheUnit** containing the data contained in that block sector. If the sector is already in the buffer it will be returned. Otherwise it will be fetched from disk, put into a **CacheUnit** and added to the buffer before being returned.

If the cache is full, then eviction needs to take place to be able to hold the new **CacheUnit**. The algorithm we have used to decide which **CacheUnit** to eject from the buffer is the **Second Chance Algorithm**. We iterate through the cache buffer. If its **accessed** attribute has been set, then we will set it to **false** and continue on to the next **CacheUnit**. When we find the first **CacheUnit** that has its **accessed** attribute set to **false** then if it is dirty, then we will write its contents to its associated block sector before clearing that **CacheUnit** and filling it with the new data.

We have also implemented **write behind** but running a small segment of code that runs on a parallel thread. This parallel thread will periodically sleep and once it awakes will run a code which iterates through the cache buffer and writes any **CacheUnits** that have their **dirty** attribute set and resets it.

At this time **write ahead** has been implemented but properly integrated. The idea behind this was to have a request loader and **write ahead daemon** which waits to handle the request. The daemon would also run on a parallel thread. Each time **cache_get_block** runs, it will also call **cache_write_ahead (sector + 1)**. **Cache_write_ahead** will then places the request onto a **read_list**. The daemon which has been waiting for the list to become non-empty, will search the buffer for the sector, otherwise will fetch the block from disk and add it to the buffer. Since this is on a separate thread, this would be done in parallel to the main thread. Unfortunately, most likely due to synchronization problems the execution of Pintos would stall when **write_ahead()** was activated.

For synchronization we enforce that only one operation can be acting on the cache buffer list at any given time. There is a global cache lock which is used for this purpose.

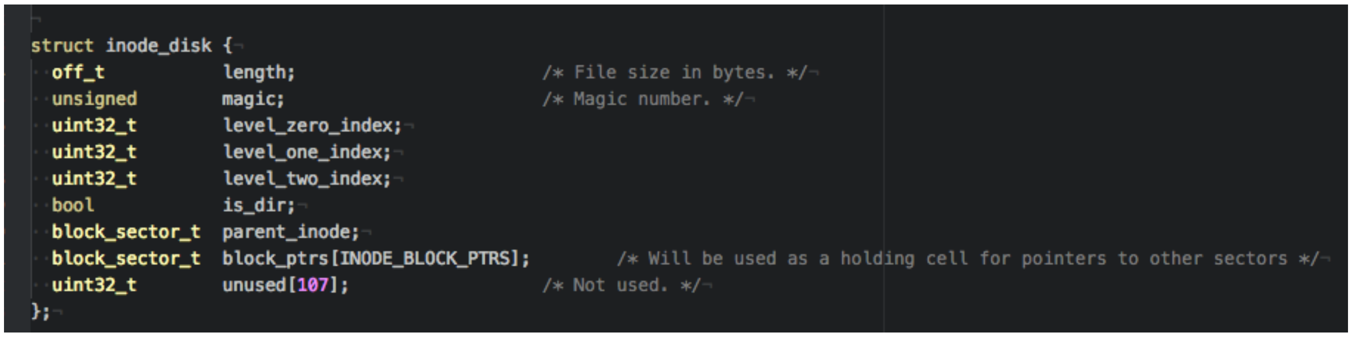
Index Extendible Files

Problem:

Files are written in sequential disk space. This means that files cannot be expanded unless there is free space below the end of the file.

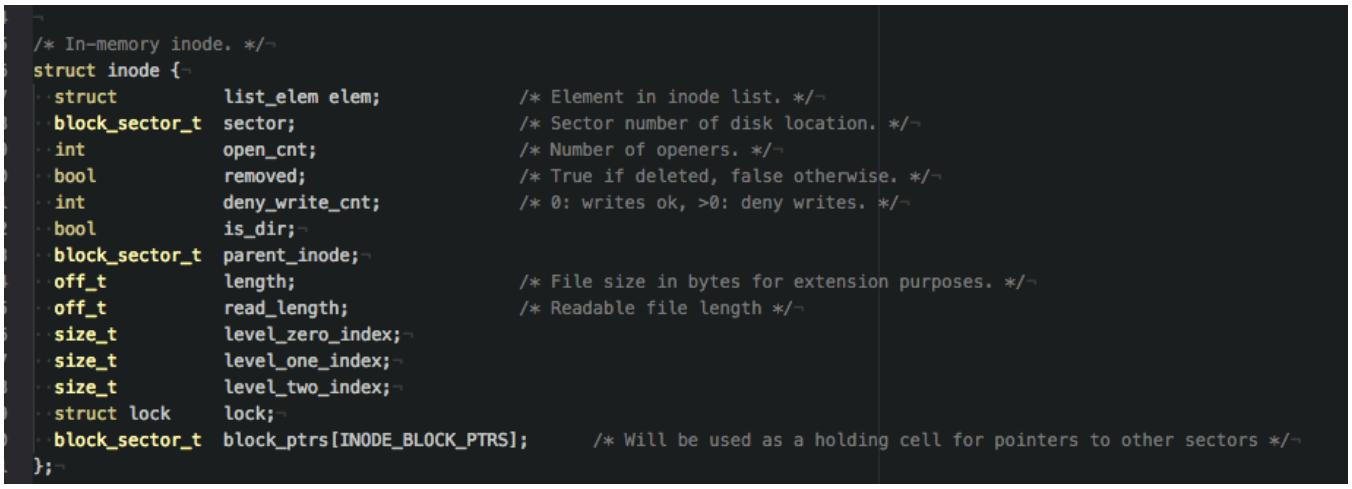
Data Structure:

The inode structure that lives in disk memory.



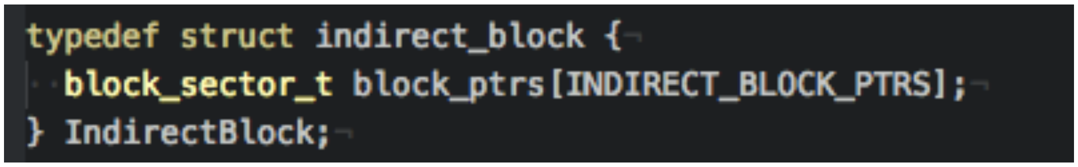
```c
struct inode_disk {
  off_t          length;                  /* File size in bytes. */
  unsigned       magic;                   /* Magic number. */
  uint32_t       level_zero_index;
  uint32_t       level_one_index;
  uint32_t       level_two_index;
  bool           is_dir;
  block_sector_t parent_inode;
  block_sector_t block_ptrs[INODE_BLOCK_PTRS];       /* Will be used as a holding cell for pointers to other sectors */
  uint32_t       unused[107];             /* Not used. */
};
```

The inode structure that exists in virtual memory.



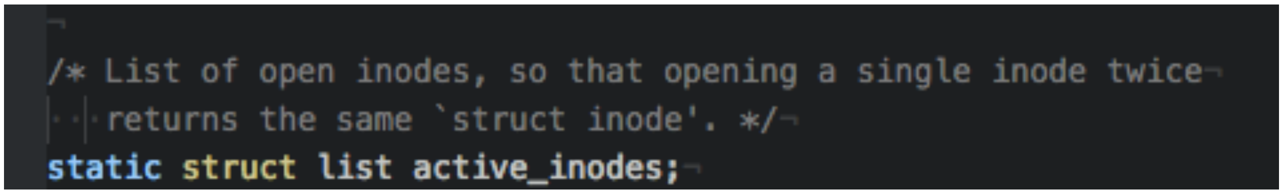
```c
/* In-memory inode. */
struct inode {
  struct         list_elem elem;            /* Element in inode list. */
  block_sector_t sector;                    /* Sector number of disk location. */
  int            open_cnt;                  /* Number of openers. */
  bool           removed;                   /* True if deleted, false otherwise. */
  int            deny_write_cnt;            /* 0: writes ok, >0: deny writes. */
  bool           is_dir;
  block_sector_t parent_inode;
  off_t          length;                    /* File size in bytes for extension purposes. */
  off_t          read_length;               /* Readable file length */
  size_t         level_zero_index;
  size_t         level_one_index;
  size_t         level_two_index;
  struct lock    lock;
  block_sector_t block_ptrs[INODE_BLOCK_PTRS];      /* Will be used as a holding cell for pointers to other sectors */
};
```

The indirect block structure that points to block sectors.



```
typedef struct indirect_block {
  block_sector_t block_ptrs[INDIRECT_BLOCK_PTRS];
} IndirectBlock;
```

The inodes in use will be kept in a global list.



```

/* List of open inodes, so that opening a single inode twice
   returns the same `struct inode'. */
static struct list active_inodes;
```

Solution:

The key components of the inodes are the array of 14 block pointers and 3 index pointers. The 14 block pointers are divided from index 0, 4 for direct block pointers, 9 for indirect block pointers and 1 for a double indirect block pointer. The 4 direct block pointers will hold pointers to 4 block sectors. The indirect block pointers will hold pointers to blocks which will be populated with addresses to up to 128 block pointers. The double indirect block pointer will point to a sector which will hold 128 pointers to indirect block pointers. The collective number of blocks encompassed by this structure multiplied by the block size of 512 bytes makes approximately 8MB, meaning a file by this structure can occupy 8MB of space.

The index pointers are used to navigate the 3 levels of block pointers. One pointer handles the 14 pointers stored directly in the inode. We will refer to this level as “level 0”. The second index pointer is used to navigate the next level of block pointers. This would include the 9 indirect block pointers, and the first level of the double indirect pointer. We will call this “level 1” The third index is used to navigate the second level of indirect blocks of the double indirect index. We will call this “level 2”.

Expansion is based upon the proposed length of the file if it was expanded. We compare this number with the current length of the file. The byte difference can then be used to figure out how many additional sectors that we need for expansion. Once this has been determined, we will find the next index in the inode which has not been assigned to yet. We determine this through the combined use of the 3 indexes that we introduced prior. From this point, we can then begin to attach all of the sectors we need to the inode sequentially incrementing the indexes as we go.

To make use of the cache buffer, each time a write or read operation is performed on an inode, we call **cache_get_block()** to fetch the data from disk or the **cache_buffer** if it is already in there.

To prevent situations where two different pieces of code are writing to the same inode at the same time, each inode has its own lock which are used each time **inode_write_at()** is invoked.