Operating System Lab Week 10
Assignment-10

Name :- Shishu **RegNo:- 2020CA089**

Objective

In this week's exercise you will be implementing support for double indirection in inode structure. By default, xv6 inode has pointers to 12 direct block pointers and one indirect pointer, this pointer points to a block, which can hold 128 direct pointers.
Hence, currently xv6 can support 128 + 12 blocks to be referenced through an inode. Assuming each block corresponds to a sector, an inode can only refer to 140 sectors (approx 70 kB).

Your job is to make changes to the inode structure so that it can support one double indirection pointer. This pointer would point to a block of indirect pointers, i.e. 128 blocks of indirect pointers, each of which refers to a block of direct pointers this referencing 128*128 sectors, by a single double indirect pointer.

File: Makefile

```
.........................................

# QEMU's gdb stub command line changed in 0.11
QEMUGDB = $(shell if $(QEMU) -help | grep -q '^-gdb'; \
      then echo "-gdb tcp::$(GDBPORT)"; \ else echo "-s
      -p $(GDBPORT)"; fi)
ifndef CPUS
CPUS := 1
endif
QEMUOPTS = -drive file=fs.img,index=1,media=disk,format=raw -drive
                       file=xv6.img,index=0,media=disk,format=raw -
smp $(CPUS) - -snapshotm 512
$(QEMUEXTRA)
........................................
```

File: mkfs.c

```
.............................................
#define NINODES 200

// Disk layout: // [ boot block | sb block | log | inode blocks |
free bit map | data blocks ]

int nbitmap = FSSIZE/(BSIZE*8) + 1; int ninodeblocks =
NINODES / IPB + 1; int nlog = LOGSIZE; int nmeta;// Number of
                    meta blocks (boot, sb, nlog, inode,
 int     nblocks    =bitmap)
 20985;                        // Number of data blocks
int nlog = LOGSIZE;

int ninodes = 200;

int size = 21029;

int fsfd;

.......................................
```

File: param.h

```
#define NPROC        64 // maximum number of processes
#define KSTACKSIZE 4096 // size of per-process kernel stack
#define NCPU          8 // maximum number of CPUs
#define NOFILE       16 // open files per process
#define NFILE       100 // open files per system
#define NINODE       50 // maximum number of active i-nodes
#define NDEV         10 // maximum major device number
#define ROOTDEV       1 // device number of file system root disk
#define MAXARG       32 // max exec arguments
#define MAXOPBLOCKS  10 // max # of blocks any FS op writes #define
LOGSIZE      (MAXOPBLOCKS*3) // max data blocks in on-disk log
#define NBUF                 (MAXOPBLOCKS*3) // size of disk block
cache                     20000
#define FSSIZE// size of file system in blocks
```

File: fs.c

```
static uint bmap(struct inode
*ip, uint bn) { uint addr, *a;
struct buf *bp;

  if(bn < NDIRECT){ if((addr = ip->addrs[bn])
    == 0) ip->addrs[bn] = addr = balloc(ip-
    >dev);
    return addr;
  } bn -=
  NDIRECT;

  if(bn < NINDIRECT){
    // Load indirect block, allocating if necessary.
    if((addr = ip->addrs[NDIRECT]) == 0) ip-
    >addrs[NDIRECT] = addr = balloc(ip->dev); bp =
    bread(ip->dev, addr); a = (uint*)bp->data;
    if((addr = a[bn]) == 0){ a[bn] = addr =
    balloc(ip->dev); log_write(bp);
    }
    brelse(bp);
    return
    addr;
  } bn -=

   NINDIRECT;

       if (bn < NDINDIRECT) {
   if ((addr = ip -> addrs[NDIRECT + 1]) == 0) { ip ->
     addrs[NDIRECT + 1] = addr = balloc(ip -> dev);
   } bp = bread(ip -> dev, addr); a = (uint * )
   bp -> data; uint block_index = bn /
   NINDIRECT; if ((addr = a[block_index]) == 0)
   { a[block_index] = addr = balloc(ip -> dev);
   } brelse(bp); uint offset =
   bn % NINDIRECT; struct buf *
   bp2;
   bp2 = bread(ip -> dev, addr); a =
   (uint * ) bp2 -> data; if ((addr =
   a[offset]) == 0) { a[offset] = addr =
   balloc(ip -> dev);
   }
   brelse(bp2);
   return addr;
 }

  panic("bmap: out of range");
}
```

FIle: big.c

```
#include "types.h"
#include "stat.h"
#include "user.h"
#include "fcntl.h"

int main() { char
buf[512]; int fd, i,
sectors;

  fd = open("big.file", O_CREATE | O_WRONLY); if(fd < 0){
  printf(2, "big: cannot open big.file for writing\n");
  exit();
  }

  sectors = 0;
  while(1){
    *(int*)buf = sectors; int cc =
    write(fd, buf, sizeof(buf)); if(cc
    <= 0) break;
    sectors++;
     if (sectors % 100 == 0)
           printf(2, ".");
  } printf(1, "\nwrote %d sectors\n",

  sectors);

  close(fd); fd = open("big.file", O_RDONLY); if(fd < 0){
  printf(2, "big: cannot re-open big.file for reading\n");
  exit();
  } for(i = 0; i < sectors; i++){ int cc = read(fd,
  buf, sizeof(buf)); if(cc <= 0){ printf(2, "big:
  read error at sector %d\n", i); exit();
    } if(*(int*)buf != i){ printf(2, "big: read the wrong data
    (%d) for sector %d\n",
             *(int*)buf, i);
      exit();
    } } printf(1, "done;

  ok\n");

  exit();
}
```

File: Makefile

```
UPROGS=\
     _cat\

...................................................

     _race1\
      _big\

EXTRA=\ mkfs.c ulib.c user.h cat.c echo.c forktest.c grep.c
     kill.c\ ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c
     zombie.c\ printf.c umalloc.c myls.c ps.c Nprocess.c c foo.c
     nice.c\ race.c        race1.c
 README dot-bochsrc  big.c\  *.pl toc.* runoff runoff1 runoff.list\
      .gdbinit.tmpl gdbutil\
```

- **bmap()**

  bmap() is called both when reading and writing a file. When writing, bmap() allocates new blocks as needed to hold file content, as well as allocating an indirect block if needed to hold block addresses.

  bmap() deals with two kinds of block numbers. The bn argument is a "logical block" , a block number relative to the start of the file. The block numbers in ip->addrs[], and the argument to bread(), are disk block numbers.

- **addr[ ]**
  The addrs array records the block numbers of the disk blocks holding the file's content.

- **brelse()** brelse moves the buffer to the front of the linked list , clears the B_BUSY bit, and wakes any processes sleeping on the buffer. Moving the buffer causes the list to be ordered by how recently the buffers were used i.e. the first buffer in the list is the most recently used, and the last is the least recently used.

- **bread()**
  It read disk sector and return buffer

- **balloc()** balloc allocates a new disk block, and bfree frees a block. Balloc starts by calling readsb to read the superblock from the disk (or buffer cache) into sb. balloc decides which blocks hold the data block free bitmap by calculating how many blocks are consumed by the boot sector, the superblock, and the inodes (using BBLOCK).