OS Homework -4

2016CS10324
R Jayanth Reddy

1. Spin-locking

Turn in 1:

The code in sched() is written assuming only one lock is held while calling it and entering it. But here we have another lock (ide lock),because interrupts were enabled while holding this lock by iderw(), when timer interrupt occurs , ptable lock is also held and so another lock is held ,while entering sched(). The main problem is created due to the wrong cpu->ncli number. (This is why we actually use pushcli, popcli disabling interrupts) (Tracing eip list we get sched(), yeild() are called first)

Turn in 2:

The file_table_lock may have been released prior to ptable.lock being acquired, so now only one lock is enabled while entering sched(). This may be due to very less time required for file allocation(order less than timer interrupt period
) rather than accessing drive , so getting a timer interrupt is less probable.

Turn in 3:

release() clears lk->pcs[0] and lk->cpu before clearing lk->locked because it leads to inconsistency in state of lock if its cleared after clearing lk->locked. If lk->locked is cleared first and one thread tries to acquire the lock then lk->pcs[0] and lk->cpu fields wouldn't have expected values. If the process does something reading these values(before them being cleared) then we can run into trouble.

First implementation of the uniprocessor lock is correct and second one is incorrect.

```
lock(L) {
  int acquired = 0;
  while (!acquired) {
     //   this process can be preempted here just when L becomes one
    cli();// before this preemption is enabled
    if (L == 1) {
        acquired = 1;
```

```
      L = 0;
    }
    sti(); // after this preemption is enabled
   }
}

unlock(L) {
  L = 1;
}
```

In the above situation the second process acquiring lock after preemption exits by making L =0 , now when the first process is again run it becomes like deadlock as unlock() is already called and this process can never make L = 1 again.

2. sleep and wakeup

Yes the code is correct as we have used while loop (instead of conditionals) , so that wakeup doesn't wake all who are sleeping in the channel and also sleep and wakeup are in locked regions. This ensures multiple consumers and producers can operate on a single queue.

3. Xv6 file system

there is no member sector in struct buf (b) , so I am using b->blockno

Turn in 1:

Printed output is
log_write 34 --> ialloc() called (from create in sysdile.c) to allocate an inode
log_write 34 --> iupdate(), updates the size and status values of inode
log_write 59 --> writei() , add a directory-entry record to the parent directory's data blocks

First in sysfile.c sys_open() and then create() is called. The first two writes happen in the inode region of the disk by creating an inode and updating it. Next dirlink() is called from create() to add new file record to parents directory (Note: here in dirlink() function log_write() is not called) , next writei() is called from this dirlink() and it writes to parent data block region.

Turn in 2:

Printed output is
log_write 58   --> balloc() , to allocate a block and it writes to block bitmap region
log_write 571 --> bzero(), it zeros out block contents and writes to data block region
log_write 571 --> writei(), it writes x to the block in data block region

log_write 34 --> iupdate(), updates a's inode values like size and address and writes in inode region
log_write 571 --> writei()
log_write 34 --> iupdate()

First sys_write() is called from sysfile.c then filewrite() (from file.c) and then writei() (from fs.c) is called. Now from writei(), bmap() (from fs.c) is called first and from it balloc() is called , and within balloc() , bzero() is called. After returning back to writei() from these calls iupdate() is called. * At last we have two sets of writei() and iupdate() because echo calls write , two times (1st -- x , 2nd -- newline).

Turn in 3:

Printed output is
log_write 59 --> writei(),it write zero to a's record in parent directory data blocks(datablock region)
log_write 34 --> iupdate() (from sys_unlink())
log_write 58 --> bfree(),it frees block bitmap to mark data blocks free ,written in block bitmap region
log_write 34 --> iupdate() (from itrunc())
log_write 34 --> iupdate() (from iput())

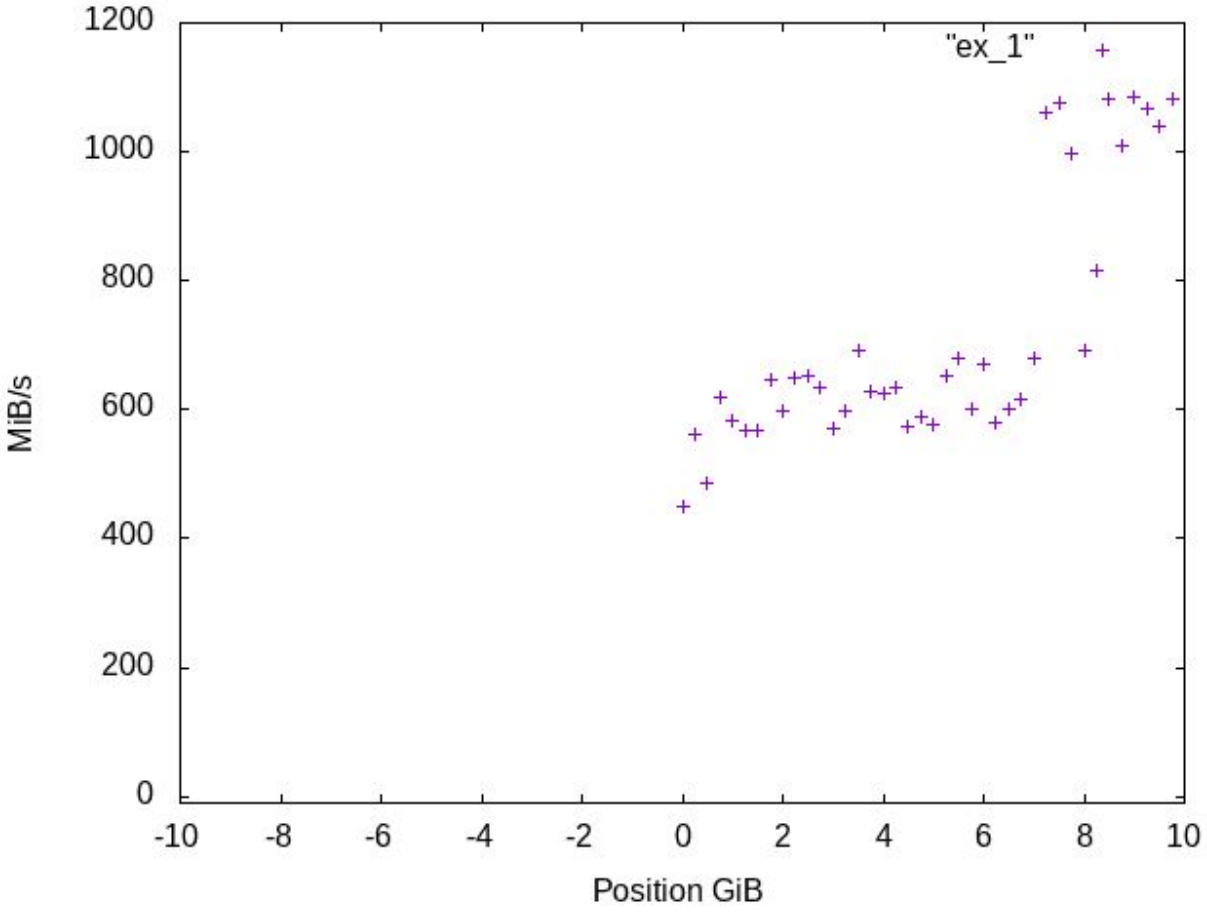
First sys_unlink() (from sysfile.c) is called , then writei() (from fs.c) is called , then iupdate (from fs.c) is called then iunlockput (from fs.c) is called. Now from this iput() , from iput() itrunc() are called. Now from itrunc(), bfree() and iupdate() are called and then we return back to iput() , from which iupdate() is called at last. Here we can observe that three iupdates are called from different functions and all are written to the inode region.
Note :- iupdate() is used to copy a modified in−memory inode to disk , so any change in any field of inode would require iupdate().
iupdate() from sys_unlink() :- ensures decrement in link count (i.e nlink field)
iupdate() from itrunc() :- it discards all the contents of inode and makes its size 0
update() from iput() :- it finally frees the inode from disk

4. ZCAV

Used ubuntu in the virtual box installed on macos.
Specifications of virtual machine
RAM -2GB , 1cpu core allotted, hard disk - 10GB , Processor: 2.7 GHz Intel Core i5
(Here system uses SSD)

I had used /dev/sda which is hard disk memory for obtaining disk parameters using zcav.
The disk specifications (I used ioping command in linux)

--- /dev/sda (block device 10 GiB) ioping statistics ---
15 requests completed in 15.4 ms, 60 KiB read, 971 iops, 3.80 MiB/s
The plot obtained using zcav is



It appears like we have very huge data transfer speed and less seek latency, this may be due to the nature of the disk which is SSD, unlike magnetic disks . Here there are no moving parts and uses flash memory, so the behaviour is different.