COL331 Operating Systems

Homework - 4

Tanay Asija
2016EE10040

Assignment: Spin Locking

1. sti() and cli() in iderw

The fn call chain seen from the %eip values printed and correlating them with kernel.asm is as follows:-

Starting from the top of the stack:-

trapret → iinit → readsb → bread → iderw → idestart → trap → yield → sched → panic

The reason for the kernel panicking is that when it tries to switch the stacks by calling sched it is holding two locks at that time → ptable.lock as well as idelock. Since we had enabled interrupts after acquiring the idelock it was possible to receive interrupts, which is what happened, as can be seen from the call chain after idestart. The call to trap after idestart indicates a timer interrupt occurred which is now trying to yield the CPU holding idelock. "sched" has the check that all calls to it must hold only "ptable.lock" and no other lock. It ensures this by checking the per-CPU "ncli" value, which is now 2 owing to the calls to "idelock" and "ptable.lock". As a result it panics and thereby calls the "panic" function. The reason it does this is because this thread is going to switch out while holding "idelock" effectively stalling all other CPU's and even the current one when it tries to run another thread which might try to acquire "idelock". This will effectively deadlock the system entirely since all CPU's will be ~~stuck~~ stuck in acquire and the thread which had to call release has been switched out.

Furthermore it will never get to run again since all calls to acquire disable interrupts. This means that the system will persist in deadlock forever.

2. ftable.lock is used by filealloc(), filedup(), and fileclose()
idelock is used by iderw() and ideintr().

filealloc, filedup, fileclose are used by system calls only and not by interrupt handlers. On the other hand idelock is used by an interrupt handler, namely the IDE device or the disk interrupt handler. However it is necessary that both locks are released before the thread context switches out to prevent the kernel panicking as in "part 1" earlier.

The reason why the kernel does not panic this time is that the size of the critical section is smaller in ~~ft~~ case of "ftable.lock" as compared to the critical section size in case of "idelock". When idelock is held by ~~idestart~~ "iderw", it calls "idestart", which then calls "idewait" that waits for the disk to become ready in a busy or polling faishon. This results in the critical section size being large in time for idelock. On the other hand the critical section protected by ftable.lock in filealloc is smaller in time as it just loops over an array of fixed size (100 in this case) until it finds an unused entry. As a result the probability of an interrupt (timer or otherwise) arriving in between the critical section ~~of~~ protected by idelock is larger as compared to the critical section protected by "ftable.lock".

3. The invariant that the programmer wishes to maintain is that upon acquiring the lock the call chain setup in lk→pcs will remain preserved until the lock is released, and similarly for lk→cpu. Shifting lk→cpu and lk→pcs outside the release() exposes them to race conditions since the lock has been released and now both the thread that released the lock and the one that acquires it next are writing to the lk→cpu and lk→pcs field simultaneously, ~~[illegible]~~ and if the thread that released the lock overwrites these fields written by the thread that acquired them, the invariant is violated. So the programmer must include those fields within the critical section and use the lock to protect its own fields.

4. The first implementation is incorrect. Let us assume that two threads T1 & T2 wish to acquire the lock. Say T1 was running and acquired the lock. It gets switched out and T2 gets to run. Now T2 will issue the di command making it impossible for any other thread to run, and it will then be stuck in a while loop deadlocking the system as the thread which had to call unlock ie T1 will never get to run again as interrupts have been disabled.

The second implementation is correct since the thread that acquires the lock atomically reads the value of L, sets it to "1" and reenables interrupts "0" or "locked" if it finds it as "1" and reenables interrupts. Thread that try to acquire the lock after it has been acquired by another thread ensure that interrupts are reenabled after every atomic check so that the thread that was holding the lock gets a chance to release it.

Assignment : Sleep and Wakeup

pcqwrite and pcqread sleep on the same channel q. However since pcqwrite if it finds the queue to be empty will produce and wake up all the consumer threads, and similarly pcqread if it find the queue to be full will read the element and call wakeup on all the producer threads, it is not possible that two producers and consumer are sleeping at the same time, so they can sleep on the same channel.

Also since all producers and consumers recheck the condition once they come out of sleep, there will not be any race conditions and hence the code is correct

When the producer calls wakeup(q), all threads that are
sleeping on the channel "q" are woken up by changing their
state from SLEEPING to RUNNABLE. It then releases the lock,
allowing the other producers and consumers to acquire it.
The consumers that were waken up will try to acquire the
lock, say one of them gets it and consumes the queue. The
remaining ones will recheck the condition if they acquire the
lock and go back to sleep until a producer calls wakeup.

Assuming that no other thread goes to sleep on q or calls
wakeup(q), no other code can call wakeup on a consumer
thread, Even if it does the consumer thread will recheck the
condition and if it is false, go back to sleep.

Assignment : XV6 file system

On the version of XV6 I built, the superblock showed the following:-

sb: size 1000, nblocks 941, ninodes 200, nlog 30

logstart 2, inodestart 32, bmapstart 58

1. echo > a

This command results in the writes to the following sectors or blocks (xv6 block size is 1 sector) — 34, 34, 59 in the same order. The printed output is

(i) log-write 34
(ii) log-write 34
(iii) log-write 59

As a new file is created, one must examine the code of create() in sysfile.c and adding print statement to log-write in the functions where it is called.

→ write (i) is made by ialloc() which allocates an unused inode for the new file. (since inodes are between blocks ~~34~~ 32 - 57)

→ write (ii) is made by iupdate() to update allocated inode's nlink field (since the same block is written)

→ write (iii) is made by writei() to create directory entry for the file in the directory where it was created, (since block 59 is written to which holds the content for the current directory, This is because on xv6 the bitmap block is itself just one block, so content blocks start from block no. 59)

Q. echo x > a

The printed output is :-

(i) log_write 58

(ii) log-write 567

(iii) log-write 567

(iv) log-write 34

(v) log-write 567

(vi) log-write 34

→ write (i) is made by balloc() which allocates a new block by reading the bitmap block which is blockno 58 and setting the first unused bit to "used". balloc is called by bmap

→ write (ii) is made by bzero() which zeroes out the newly allocated block which is blockno 567. bzero is called by balloc

→ write (iii) is made by writei() to write the first byte of echo's output ie "x". It then calls iupdate to update the sz of the inode

→ write (iv) is made by iupdate to update the inode's "sz" field in blockno. 34

→ write (v) is made also by writei() but this time to write the newline character. It again calls iupdate as "sz" has changed

→ write (vi) is made by iupdate to update the sz of the allocated inode in "part 1" which is in blockno. 34

3. rm a

The printed output is –

(i) log-write 59

(ii) log-write 34

(iii) log-write 58

(iv) log-write 34

(v) log-write 34

→ write (i) is to delete the directory entry record in the directory where "a" was created. The blockno. (59) is the same as the one which was written to in "part 1" when file "a" was created to write the new directory entry.

→ write (ii) is made by iupdate() to reduce the nlink count of the inode that represented "a"

→ write (iii) is made by bfree to mark the blocks allocated for file "a" as free or unused in blockno. 58 which is the bitmap block. bfree is called by itrunc(). This frees up the 567th block entry in bitmap block.

→ write (iv) is made again by iupdate to update the "sz" field of the inode. It is also called within itrunc()

→ write (v) is made again by iupdate() to update the ~~valid~~ "type" field of the inode. It is however called within iput() after the call to itrunc().

icache being a write through cache calls iupdate every time any field of the inode is altered.

Assignment : ZCAV

1. Disk on a physical laptop

I executed the zcav program on a Virtual machine with a guest OS of Linux and host OS of Windows. The disk image has an actual size of 13.80 GB and a virtual size of 16.86 GB. The hypervisor used was Oracle VirtualBox.

The Laptop details are : Dell Vostro 3446, Windows 8.1 (does not support Windows Subsystem for Linux)

The disk characteristics are :-

Model: Toshiba MQ01ABF050

Track-Track Seek : 2ms

Max seek time : 22ms

Rotation speed : 5400 rpm

Avg Latency : 5.56ms

Size: ~~4TB~~ 465.76 GB

It has constant sectors/Track of 63. As a result even before running the zcav I expected the number of zones to be 1.

The disk specs claim a transfer rate of 6Gb/s or 750MB/s.

Result from ZCAV:-

There are no visible zones observed, implying there is a single zone for the 16 GB of memory read. The maximum bandwidth observed is ~ 175 MB/s. The results are seen in "zcav-desktop-disk"

Since the no. sectors/track is a constant, as reported by the disk manufacturer, the no. of zones expected even for a complete 500GB read of the disk would have yielded a single zone. As a result it is difficult to comment on the mapping between sector no. and physical disk layout.

2. USB Drive

Product : SanDisk Cruzer Force USB Flash Drive

Model : SDCZ71-032G-B35

Size : 32GB

Interface : USB 2.0

Type : Flash Drive

Speed : 480 Mb/s or 60MB/s
(as per 2.0 standards)

The program was again run on a VM which was able to access the externally connected USB drive.

Results from zcav -

A single zone is visible owing to the constant speed at all positions. The maximum BW observed is ~17 MB/s

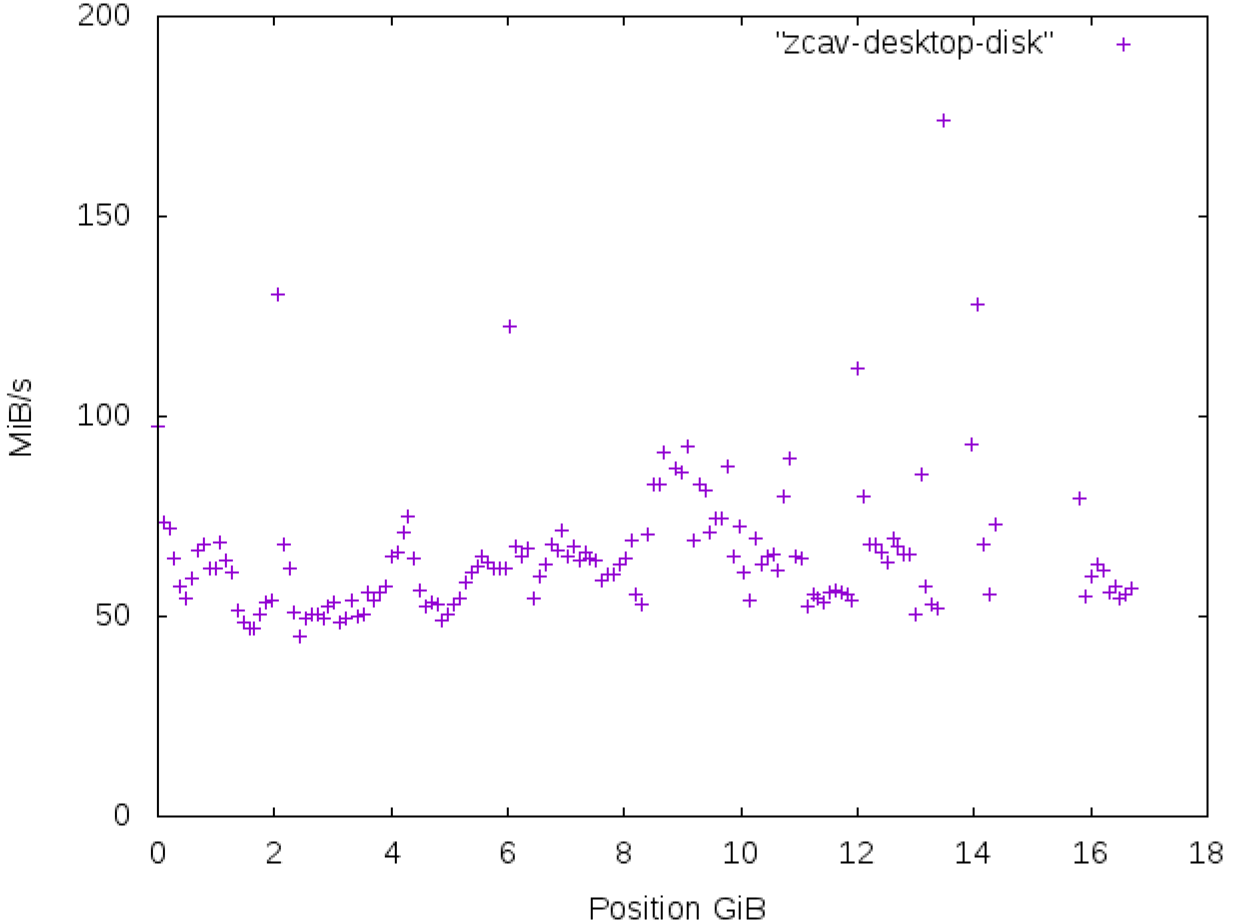
The results are seen in the plot "zcav-usb-sandisk"

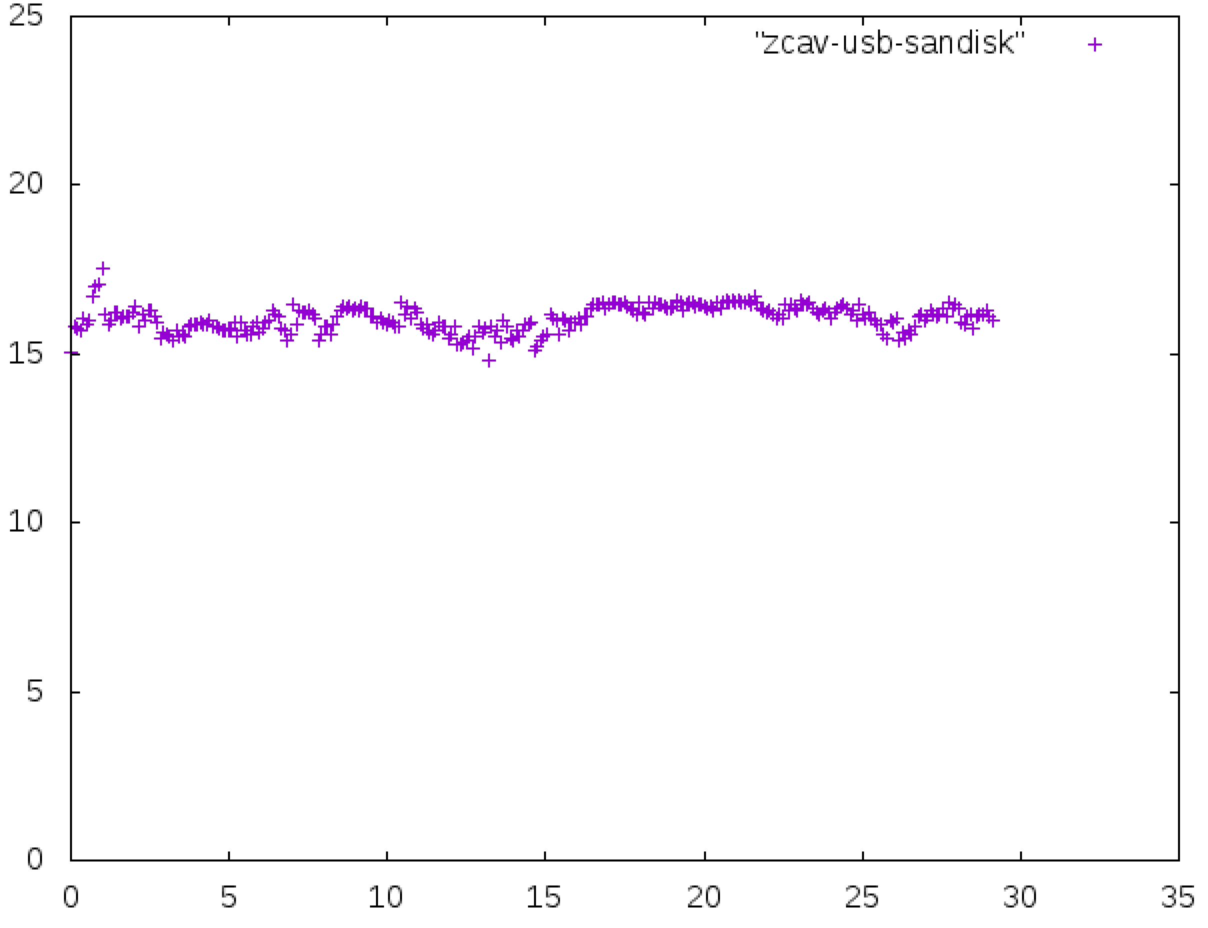
→ Comparison of 1. and 2.

Since "1.0" was ran on a HDD which is having moving parts it was expected that the platter would be divided into zones. However since the sectors/track is a constant only one zone is observed.

"2.0" was ran on a USB flash drive with no moving part, the speed was expected to be constant across all positions which it is. The speed is less than the claimed speed by the manufacturers in both cases.

Item	Value
Description	Disk drive
Manufacturer	(Standard disk drives)
Model	TOSHIBA MQ01ABF050
Bytes/Sector	512
Media Loaded	Yes
Media Type	Fixed hard disk
Partitions	6
SCSI Bus	0
SCSI Logical Unit	0
SCSI Port	0
SCSI Target ID	0
Sectors/Track	63
Size	465.76 GB (500,105,249,280 bytes)
Total Cylinders	60,801
Total Sectors	976,768,065
Total Tracks	15,504,255
Tracks/Cylinder	255


"zcav-desktop-disk"
MiB/s
Position GiB
0
50
100
150
200
2
4
6
8
10
12
14
16
18



'zcav-usb-sandisk'
25
20
15
10
5
0
0
5
10
15
20
25
30
35