Writeup 2

CS 444

October 29, 2018

Anton Synytsia, Eytan Brodsky, David Jansen

Contents

1 C-Look Elevator Scheduler

C-Look elevator reads/writes from the lowest request address to the highest request address without moving back or forth, at any time in between, except for after dispatching the highest requested address, where the elevator resets to the next, lowest, requested address [1]. This allows for the disk head to read/write efficiently, in an expected direction; otherwise, changing directions too much will have performance implications and can physically deteriorate the hard-drive.

In order to implement C-Look, we had to start with an existing scheduler as a template template:

```
cd /scratch/fall2018/group1/linux-yocto-3.19/block/
cp noop-iosched.c clook-iosched.c
```

We then renamed all functions and identifiers C-Look, and modified two functions:

Request Adder Instead of appending to the end, like NO-OP did, we insert at a sorted location within the doubly-linked list of requests. Added requests are stored and sorted to the right of head:

`HEAD` $\rightarrow 1 \rightarrow 2 \rightarrow 7 \rightarrow 33 \rightarrow 34 \rightarrow \ldots$

Our sort criteria is blk_rq_pos.

In order to insert at the right location, we do the following:

1) Start at `HEAD` and traverse forward until:
   - Next points to `HEAD`, meaning at last request node.
   - Next points to a request node, whose block position is greater than the block position of the request node being added.
2) Append request node to currently, iterated node, via list_add.

```
static void clook_add_request(struct request_queue *q, struct request *rq)
{
    struct clook_data *nd = q->elevator->elevator_data;

    struct list_head *head = &nd->queue;
    struct list_head *item = &rq->queuelist;

    // Find node to append to
    struct list_head *node = head;
    struct list_head *next_node = head->next;

    while (next_node != head) { // while node is not last
        // get request at next node
        struct request *nrq = list_entry(next_node, struct request, queuelist);

        // if request at next node is greater than rq, break out
        if (blk_rq_pos(nrq) > blk_rq_pos(rq)) break;
```

```
            // increment
            node = next_node;
            next_node = node->next;
        }

        // Append request item to node
        list_add(item, node);
    }
```

Request Dispatcher Our modified version of the dispatcher prints out a note in case the request access order is inconsistent, that is if next request is less than previous:

This consistency verification also required a global variable, which we defined at the top:

```
sector_t g_last_seg = 0;
```

The following is a section of code for

2 Compiling Qemu with C-Look

2.1 Setup

In order to compile Qemu with our scheduler, we had to edit two files within `/scratch/fall2018/group1/linux-yocto-3.19/block/`:

Makefile Within the Makefile, we had to add another configuration:

```
obj-$(CONFIG_IOSCHED_CLOOK) += clook-iosched.o
```

Kconfig.iosched Within this file, we had to add the following:

```
config IOSCHED_CLOOK
    bool
    default y
    ---help---
      CS444 Group1 Clook scheduler.
```

2.2 Compiling

To recompile the scheduler, we ran the following commands:

```
cd /scratch/fall2018/group1/linux-yocto-3.19/
make clean
cp /scratch/files/config-3.19.2-yocto-standard .config
# press y to overwrite
make -j4 all > log_res.txt
```

3 Starting Qemu

In order to ease the process of running qemu, as well as, not having to run it in debug environment, we created a `qemu-run.sh` with custom commands:

```
source ./environment-setup-i586-poky-linux

#qmu-system-i386 -gdb tcp::5501 -S -nographic -kernel bzImage-qemux86.bin -drive
↪  file=core-image-lsb-sdk-qemux86.ext4,if=virtio -enable-kvm -net none -usb -localtime --no-reboot
↪  --append "root=/dev/vda rw console=ttyS0 evelator=clook"

qemu-system-i386 -gdb tcp::5501 -nographic -kernel
↪  /scratch/fall2018/group1/linux-yocto-3.19/arch/x86/boot/bzImage -drive
↪  file=core-image-lsb-sdk-qemux86.ext4 -enable-kvm -net none -usb -localtime --no-reboot --append
↪  "root=/dev/hda rw console=ttyS0 debug elevator=clook"
```

To run, execute the following commands:

```
# Set path
cd /scratch/fall2018/group1/
# (Optional) Allow execute permission
chmod +x qemu-run.sh
# Run
./qemu-run.sh
```

4 Testing

1) Create test script.

```
import os
import mmap
import struct
import random

mem = os.open("./big_file", os.O_RDWR)
addr = mmap.mmap(mem, 0x40000000, offset=random.randint(1, 256)*mmap.ALLOCATIONGRANULARITY)

base = 100
i = struct.unpack('I', addr[base:base+4])[0]
print(i)
```

2) cat /dev/urandom >big_file (create a large file)
3) Run the python script on the file. If there are no errors, then the schedule is working.

5 Concurrency 2

1) Main Point of the Assignment: The purpose of this concurrency assignment was to get us to start working with parallel programming. Not only did this assignment get us to work more with threads that each share a resource in the form of a mutex, it also forced us to think about how programs work in parallel to one another.
2) Our Approach to the problem: Our initial approach to the assignment was to think through how we wanted to implement the shared resource, we eventually ended up choosing a mutex. Next we decided to use rdrand again in order to generate random numbers for both the thinking and eating periods. If rdrand wasn't supported we made sure to use mt19937. In order to get each philosopher to eat we ran a while loop where each philosopher could be doing one of three things, eating, thinking, or waiting to get access to two forks. Our code for this process is shown below. This code is ran after a philosopher has completed their initial thinking period.

   This process begins with a philosopher grabbing and locking the mutex. The philosopher with the mutex will then check to see if two forks are available. If there are not two forks the philosopher will unlock the mutex and restart the process. If there ends up being 2 or more forks available the philosopher will grab the forks and unlock the mutex. This philosopher will now be assigned a random number between 2 and 9 seconds so they can eat. Once they are done eating they will grab and lock the mutex once again and return the two forks. When the forks are returned they will unlock the mutex and sleep (thinking period) again for a randomly assigned time of 1-20 seconds before repeating the process all over again.
3) How we ensured the solution was correct: We ensured our solution was correct by printing statements to the command line. Every time a philosopher grabbed and returned the forks we were able to see it outputted to the command line. By doing this we were able to see that each philosopher was able to grab the forks, eat, and return the forks. After running the process multiple times we were able to ensure that our program was running correctly and each philosopher was able to eat.
4) What we learned: From this assignment we learned how to get multiple threads to work in parallel with one another in order to access a shared resource. We also improved our ability to think in parallel by discussing as a group how to approach the problem and working together on the solution.

6 Version Control Log

acronym	**meaning**
V	`version`
tag	`git tag`
MF	Number of `modified files`.
AL	Number of `added lines`.
DL	Number of `deleted lines`.

V	**tag**	**date**	**commit message**	**MF**	**AL**	**DL**
1		2018-10-08	Initial commit	1	1	0
2		2018-10-08	Create .gitignore	1	6	0

Please continue at the next page

V	tag	date	commit message	MF	AL	DL
3		2018-10-08	Update state	1	0	1
4		2018-10-08	Setup readme	1	5	0
5		2018-10-08	Update README.md	1	2	0
6		2018-10-08	Uploaded concurrency	2	225	0
7		2018-10-08	changed permissions	4	0	0
8		2018-10-09	Setup writeup TeX	2	126	0
9		2018-10-09	Added pygments	1	98	0
10		2018-10-09	included pygements to preamble	1	8	6
11		2018-10-09	First sucessful compile of writeup1	6	2008	2
12		2018-10-09	Updated gitignore	5	3	2005
13		2018-10-09	Setup Writeup	1	10	20
14		2018-10-09	Added git attributes to enforce EOL	1	7	0
15		2018-10-09	EOL deal	2	1	2
16		2018-10-09	Trying to fix TEX compiling	2	79	1
17		2018-10-09	Added sections to writeup	3	8	81
18		2018-10-09	Added the writeup for Command Logs and for the Qemu flags	11	97	10
19		2018-10-10	Work on concurrency writeup	8	30	56
20		2018-10-10	More work on Concurrency writeup	2	12	7
21		2018-10-10	Made Concurrency use circular queue	1	41	34
22		2018-10-10	Fixed concurrency print type	1	9	6
23		2018-10-10	Renamed Assignment1 folder to Concurency and finished my writeup	7	251	244
24		2018-10-11	Disabled BIB command in makefile	2	3	3
25		2018-10-11	Added compile instructions	1	26	3
26		2018-10-11	README adjustments	1	3	3
27		2018-10-13	corrected name	1	1	1
28		2018-10-13	explain -localtime and -append root=/dev/vda rw console=ttyS0 debug	1	2	2
29		2018-10-13	add work log.	1	6	0
30		2018-10-13	Added the version control table	1	31	1
31		2018-10-13	Changed up how the lists were made	1	48	31
32		2018-10-15	Refactored rdrand to use X86 intrinsic and created a general rand function	1	88	69
33		2018-10-15	Updated concurrency writeup to reflect the refactored code.	1	7	10

Please continue at the next page

V	tag	date	commit message	MF	AL	DL
34		2018-10-15	Updated work log	5	477	77
35		2018-10-15	Removed thirdparty	1	0	362
36		2018-10-22	Added concurrency2	2	219	0
37		2018-10-23	Setup writeup2 and update comments in concurrency	5	233	4
38		2018-10-27	Unignored Linux Yocto with our sched	7	2	1
39		2018-10-27	Unignoring yocto	1	0	2
40		2018-10-27	yocto...	1	1	0
41		2018-10-27	Added clook scheduler	4	582	2
42		2018-10-27	Setup Writeup2	11	9310	501
43		2018-10-27	Updated makefiles	3	8	9
44		2018-10-27	Making refs work	1	1	1
45		2018-10-27	Added Eyton's scheduler tester	2	16	0
46		2018-10-27	Added David's makefile	2	27	1
47		2018-10-28	Fixed compile errors in CLOOK and modifed Kconfig for inclusion	4	84	6
48		2018-10-28	Added command for starting VM	1	5	0
49		2018-10-29	Made clook print if unexpected access order	2	8	0
50		2018-10-29	Clook Scheduler Writeup	11	165	8
51		2018-10-29	Fixes to README	1	7	7

References

[1] “Disk scheduling,” https://web.archive.org/web/20080606005055/http://www.dcs.ed.ac.uk/home/stg/pub/D/disk.html.