OSD 19CS2106S

Session – 31

Implementation of spin Locks

Sequence-number-increment example:

When multiple process shares the same file, we need to make sure that each process sees a consistent view of its data. If each process uses files that other process don't read or modify, no consistency problems will exist. Similarly, if a file is read-only, there is no consistency problem with more than one process reading its value at the same time. However, when one process can modify a file that other process can read or modify, we need to synchronize. When one process modifies a file, other process can potentially see inconsistencies when reading the file

Consider the following scenario, which comes from the Unix print spoolers (the BSD **`lpr`** command and the System V **`lp`** command). The process that adds a job to the print queue (to be printed at a later time by another process) must assign a unique sequence number to each print job. The process ID, which is unique while the process is running, cannot be used as the sequence number, because a print job can exist long enough for a given process ID to be reused. A given process can also add multiple print jobs to a queue, and each job needs a unique number. The technique used by the print spoolers is to
have a file for each printer that contains the next sequence number to be used. The file is just a single line containing the sequence number in ASCII. Each process that needs to assign a sequence number goes through three steps:

1. it reads the sequence number file,
2. it uses the number, and it increments the number
3. and writes it back.

The problem is that in the time a single process takes to execute these three steps, another process can perform the same three steps. Chaos can result, as we will see in some examples that follow.

What is needed is for a process to be able to set a lock to say that no other process can access the file until the first process is done. Program:1 does these three steps. The functions **`my-lock`** and **`my-unlock`** are called to lock the file at the beginning and unlock the file when the process is done with the sequence number. We print the name by which the program is being run **`(argv`** `[0 ]`) each time around the loop when we print the sequence number.

```
void
my_lock(int fd)
{
        return;
}

void
my_unlock(int fd)
{
        return;
}
```

If the sequence number in the file is initialized to one, and a single copy of the program is run, we get the following output:

```
[vishnu@team-osd ~]$ cc seqnumnolock.c
[vishnu@team-osd ~]$ vi seqno
[vishnu@team-osd ~]$ ./a.out
./a.out: pid = 5448, seq# = 1
./a.out: pid = 5448, seq# = 2
./a.out: pid = 5448, seq# = 3
./a.out: pid = 5448, seq# = 4
./a.out: pid = 5448, seq# = 5
./a.out: pid = 5448, seq# = 6
./a.out: pid = 5448, seq# = 7
./a.out: pid = 5448, seq# = 8
./a.out: pid = 5448, seq# = 9
./a.out: pid = 5448, seq# = 10
./a.out: pid = 5448, seq# = 11
./a.out: pid = 5448, seq# = 12
./a.out: pid = 5448, seq# = 13
./a.out: pid = 5448, seq# = 14
./a.out: pid = 5448, seq# = 15
./a.out: pid = 5448, seq# = 16
./a.out: pid = 5448, seq# = 17
```

./a.out: pid = 5448, seq# = 18
./a.out: pid = 5448, seq# = 19
./a.out: pid = 5448, seq# = 20

When the sequence number is again initialized to one, and the program is run twice in the background, we have the following output:

```
[vishnu@team-osd ~]$ vi seqno
[vishnu@team-osd ~]$ ./a.out & ./a.out &
[1] 7891
[2] 7892
[vishnu@team-osd ~]$ ./a.out: pid = 7892, seq# = 1
./a.out: pid = 7892, seq# = 2
./a.out: pid = 7892, seq# = 3
./a.out: pid = 7892, seq# = 4
./a.out: pid = 7892, seq# = 5
./a.out: pid = 7892, seq# = 6
./a.out: pid = 7892, seq# = 7
./a.out: pid = 7892, seq# = 8
./a.out: pid = 7892, seq# = 9
./a.out: pid = 7892, seq# = 10
./a.out: pid = 7892, seq# = 11
./a.out: pid = 7892, seq# = 12
./a.out: pid = 7892, seq# = 13
./a.out: pid = 7892, seq# = 14
./a.out: pid = 7891, seq# = 8
./a.out: pid = 7892, seq# = 15
./a.out: pid = 7892, seq# = 16
./a.out: pid = 7892, seq# = 17
./a.out: pid = 7891, seq# = 17
./a.out: pid = 7892, seq# = 18
./a.out: pid = 7891, seq# = 19
./a.out: pid = 7892, seq# = 19
./a.out: pid = 7892, seq# = 20
./a.out: pid = 7891, seq# = 20
./a.out: pid = 7891, seq# = 21
./a.out: pid = 7891, seq# = 22
./a.out: pid = 7891, seq# = 23
./a.out: pid = 7891, seq# = 24
./a.out: pid = 7891, seq# = 25
./a.out: pid = 7891, seq# = 26
./a.out: pid = 7891, seq# = 27
./a.out: pid = 7891, seq# = 28
./a.out: pid = 7891, seq# = 29
./a.out: pid = 7891, seq# = 30
./a.out: pid = 7891, seq# = 31
./a.out: pid = 7891, seq# = 32
./a.out: pid = 7891, seq# = 33
./a.out: pid = 7891, seq# = 34
./a.out: pid = 7891, seq# = 35
./a.out: pid = 7891, seq# = 36

[1]-  Done                    ./a.out
[2]+  Done                    ./a.out
```

This is not what we want. Each process reads, increments, and writes the sequence number file 20 times (there are exactly 40 lines of output), so the ending value of the sequence number should be 40.

What we need is some way to allow a process to prevent other processes from accessing the sequence number file while the three steps are being performed. That is, we need these three steps to be performed as an *atomic operation* with regard to other processes. Another way to look at this problem is that the lines of code between the calls to **`my-lock`** and **`my-unlock`** in program form a *critical region*

When we run two instances of the program in the background as just shown, the output is *nondeterministic.* There is no guarantee that each time we run the two programs we get the same output. This is OK if the three steps listed earlier are handled atomically with regard to other processes, generating an ending value of 40. But this is not OK if the three steps are not handled atomically, often generating an ending value less than 40, which is an error. For example, we do not care whether the first process increments the sequence number from 1 to 20, followed by the second process incrementing

it from 21 to 40, or whether each process runs just long enough to increment the sequence number by two (the first process would print 1 and 2, then the next process would print 3 and 4, and so on).
Being nondeterministic does not make it incorrect. Whether the three steps are performed atomically is what makes the program correct or incorrect. Being nondeterministic, however, usually makes debugging these types of programs harder.

Lock Files

Posix guarantees that if the **open** function is called with the **O-CREAT** (create the file if it does not already exist) and **O-EXCL** flags (exclusive open), the function returns an error if the file already exists. Furthermore, the check for the existence of the file and the creation of the file (if it does not already exist) must be ***atomic*** with regard to other processes. We can therefore use the file created with this technique as a lock. We are guaranteed that only one process at a time can create the file (i.e., obtain the lock), and to release the lock, we just **unlink** the file. Figure 9.12 shows a version of our locking functions using this technique. If the
open succeeds, we have the lock, and the **my-lock** function returns. We **close** the file because we do not need its descriptor: the lock is the existence of the file, regardless of whether the file is open or not. If **open** returns an error of **EEXIST,** then the file exists and we try the **open** again

```
/* Lock functions using open()with O_CREAT and O_EXCL flags. */
#include        <errno.h>
#include        <fcntl.h>               /* for nonblocking */
#include        <stdio.h>
#include        <stdlib.h>
#include        <string.h>
#include        <unistd.h>
#define MAXLINE 4096                    /* max text line length */
#define SEQFILE "seqno"                 /* filename */
#define LOCKFILE "seqno.lock"
void    my_lock(int), my_unlock(int);
int
main(int argc, char **argv)
{
        int     fd;
        long    i, seqno;
        pid_t   pid;
        ssize_t n;
        char    line[MAXLINE + 1];
        pid = getpid();
        fd = open(SEQFILE, O_RDWR, 0666);
        for (i = 0; i < 20; i++) {
                my_lock(fd);                    /* lock the file */
                lseek(fd, 0L, SEEK_SET);        /* rewind before read */
                n = read(fd, line, MAXLINE);
                line[n] = '\0';                 /* null terminate for sscanf */
                n = sscanf(line, "%ld\n", &seqno);
                printf("%s: pid = %ld, seq# = %ld\n", argv[0], (long) pid, seqno);
                seqno++;                        /* increment sequence number */
                snprintf(line, sizeof(line), "%ld\n", seqno);
                lseek(fd, 0L, SEEK_SET);        /* rewind before write */
                write(fd, line, strlen(line));
                my_unlock(fd);                  /* unlock the file */
        }
        exit(0);
}
void
my_lock(int fd)
{
        int     tempfd;
        while ( (tempfd = open(LOCKFILE, O_RDWR|O_CREAT|O_EXCL, 0644)) < 0) {
                if (errno != EEXIST)
                        printf("open error for lock file");
                /* someone else has the lock, loop around and try again */
        }
        close(tempfd);                  /* opened the file, we have the lock */
}
void
my_unlock(int fd)
{
        unlink(LOCKFILE);               /* release lock by removing file */
}
[vishnu@team-osd ~]$ vi seqno
```

```
[vishnu@team-osd ~]$ cc seqnum.c
[vishnu@team-osd ~]$ vi seqno
[vishnu@team-osd ~]$ ./a.out
./a.out: pid = 5140, seq# = 1
./a.out: pid = 5140, seq# = 2
./a.out: pid = 5140, seq# = 3
./a.out: pid = 5140, seq# = 4
./a.out: pid = 5140, seq# = 5
./a.out: pid = 5140, seq# = 6
./a.out: pid = 5140, seq# = 7
./a.out: pid = 5140, seq# = 8
./a.out: pid = 5140, seq# = 9
./a.out: pid = 5140, seq# = 10
./a.out: pid = 5140, seq# = 11
./a.out: pid = 5140, seq# = 12
./a.out: pid = 5140, seq# = 13
./a.out: pid = 5140, seq# = 14
./a.out: pid = 5140, seq# = 15
./a.out: pid = 5140, seq# = 16
./a.out: pid = 5140, seq# = 17
./a.out: pid = 5140, seq# = 18
./a.out: pid = 5140, seq# = 19
./a.out: pid = 5140, seq# = 20
[vishnu@team-osd ~]$ ./a.out & ./a.out &
[1] 5263
[2] 5264
[vishnu@team-osd ~]$ ./a.out: pid = 5263, seq# = 1
./a.out: pid = 5264, seq# = 2
./a.out: pid = 5263, seq# = 3
./a.out: pid = 5264, seq# = 4
./a.out: pid = 5263, seq# = 5
./a.out: pid = 5264, seq# = 6
./a.out: pid = 5263, seq# = 7
./a.out: pid = 5264, seq# = 8
./a.out: pid = 5263, seq# = 9
./a.out: pid = 5264, seq# = 10
./a.out: pid = 5263, seq# = 11
./a.out: pid = 5264, seq# = 12
./a.out: pid = 5263, seq# = 13
./a.out: pid = 5264, seq# = 14
./a.out: pid = 5263, seq# = 15
./a.out: pid = 5264, seq# = 16
./a.out: pid = 5263, seq# = 17
./a.out: pid = 5264, seq# = 18
./a.out: pid = 5263, seq# = 19
./a.out: pid = 5264, seq# = 20
./a.out: pid = 5263, seq# = 21
./a.out: pid = 5263, seq# = 22
./a.out: pid = 5264, seq# = 23
./a.out: pid = 5263, seq# = 24
./a.out: pid = 5264, seq# = 25
./a.out: pid = 5263, seq# = 26
./a.out: pid = 5264, seq# = 27
./a.out: pid = 5263, seq# = 28
./a.out: pid = 5264, seq# = 29
./a.out: pid = 5263, seq# = 30
./a.out: pid = 5264, seq# = 31
./a.out: pid = 5263, seq# = 32
./a.out: pid = 5264, seq# = 33
./a.out: pid = 5263, seq# = 34
./a.out: pid = 5264, seq# = 35
./a.out: pid = 5263, seq# = 36
./a.out: pid = 5264, seq# = 37
./a.out: pid = 5263, seq# = 38
./a.out: pid = 5264, seq# = 39
./a.out: pid = 5264, seq# = 40
```

[1]- Done ./a.out
[2]+ Done ./a.out

Each file type has its own unique data structure that records all the information in a format only the software that created it can understand. Databases store metadata in fixed tables with rigid schemas and pointers linking multiple tables together. This enables data to be structured and categorized so that it can be indexed, searched and easily manipulated using Structured Query Language (SQL) transactions. A simple SQL query is able to extract all the information about a file and its relational hierarchy to all the other files in your database.

how AUTO_INCREMENT and concurrent insert works using locks in DBMS

Terminal – 1	Terminal – 2

`$mysql -u root -p`
`MariaDB [(none)]> create database testdb;`
`MariaDB [(none)]> create user 'testuser'@localhost identified by 'password';`
`MariaDB [(none)]> grant all on testdb.* to 'testuser' identified by 'password';`
`MariaDB [(none)]> quit;`
`$mysql -u testuser -p`
 	
`MariaDB [(none)]> SHOW DATABASES;`
`MariaDB [(none)]> use testdb;`
 	
`MariaDB [testdb]> CREATE TABLE cities (id INT NOT NULL PRIMARY KEY AUTO_INCREMENT, name VARCHAR(100), pop MEDIUMINT, founded DATE, owner VARCHAR(100));`
`MariaDB [testdb]> show tables;`
`MariaDB [testdb]> describe cities;`
 	
`MariaDB [testdb]> DELIMITER //`
 	
`CREATE PROCEDURE dorepeatt(p1 INT)`
`  BEGIN`
`    SET @x = 1;`
`    REPEAT SET @x = @x + 1;`
`INSERT INTO cities (id, name, pop, founded, owner)`
`VALUES`
`(NULL, 'a', 20, "2010-01-12",'testuser'),`
`(NULL, 'b', 56, "2011-2-28",'testuser'),`
`(NULL, 'c', 3, "120314",'testuser'),`
`(NULL, 'd', 34, "13*04*21",'testuser'),`
`(NULL, 'e', 12, "2010-09-19",'testuser'),`
`(NULL, 'f', 09, "2010-12-13",'testuser');`
` UNTIL @x > p1 END REPEAT;`
`  END`
`//`
 	
`MariaDB [testdb]> DELIMITER ;`
	`$mysql -u root -p`
	`MariaDB [(none)]> create user 'vishnu'@localhost identified by 'password';`
	`MariaDB [(none)]> grant all on testdb.* to 'vishnu' identified by 'password';`
	`MariaDB [(none)]> quit;`
	
	`$mysql -u vishnu -p`
	
	`MariaDB [(none)]> SHOW DATABASES;`
	`MariaDB [(none)]> use testdb;`
	`MariaDB [testdb]>show tables;`
	`MariaDB [testdb]>describe cities;`
	
	`MariaDB [testdb]>DELIMITER //`
	
	`CREATE PROCEDURE dorepeatv(p1 INT)`
	`  BEGIN`
	`    SET @x = 1;`
	`    REPEAT SET @x = @x + 1;`
	`INSERT INTO cities (id, name, pop, founded, owner)`
	`VALUES`
	`(NULL, 'A', 120, "2020-09-30",'vishnu'),`
	`(NULL, 'B', 156, "2001-8-22",'vishnu'),`
	`(NULL, 'C', 13, "180924",'vishnu'),`
	`(NULL, 'D', 134, "17*09*13",'vishnu'),`
	`(NULL, 'E', 112, "2018-11-29",'vishnu'),`
	`(NULL, 'F', 109, "2000-02-17",'vishnu');`
	` UNTIL @x > p1 END REPEAT;`
	`  END`
	`//`
	
	`MariaDB [testdb]>DELIMITER ;`

Execute in two terminals in parallel	
`MariaDB [testdb]> CALL dorepeatt(100);`	`MariaDB [testdb]>CALL dorepeatv(100);`

```
MariaDB [testdb]> select * from cities;
+------+------+------+------------+----------+
| id   | name | pop  | founded    | owner    |
+------+------+------+------------+----------+
|    1 | a    |   20 | 2010-01-12 | testuser |
|    2 | b    |   56 | 2011-02-28 | testuser |
|    3 | c    |    3 | 2012-03-14 | testuser |
|    4 | d    |   34 | 2013-04-21 | testuser |
|    5 | e    |   12 | 2010-09-19 | testuser |
.
.
.
.

|  364 | d    |   34 | 2013-04-21 | testuser |
|  365 | e    |   12 | 2010-09-19 | testuser |
|  366 | f    |    9 | 2010-12-13 | testuser |
|  367 | A    |  120 | 2020-09-30 | vishnu   |
|  368 | B    |  156 | 2001-08-22 | vishnu   |
.
.
.


|  399 | c    |    3 | 2012-03-14 | testuser |
|  400 | d    |   34 | 2013-04-21 | testuser |
|  401 | e    |   12 | 2010-09-19 | testuser |
```

```
|  402 | f    |    9 | 2010-12-13 | testuser |
|  403 | A    |  120 | 2020-09-30 | vishnu   |
|  404 | B    |  156 | 2001-08-22 | vishnu   |
|  405 | C    |   13 | 2018-09-24 | vishnu   |
|  406 | D    |  134 | 2017-09-13 | vishnu   |
|  407 | E    |  112 | 2018-11-29 | vishnu   |
|  408 | F    |  109 | 2000-02-17 | vishnu   |
|  409 | a    |   20 | 2010-01-12 | testuser |
|  410 | b    |   56 | 2011-02-28 | testuser |

.
.
.

|  634 | d    |   34 | 2013-04-21 | testuser |
|  635 | e    |   12 | 2010-09-19 | testuser |
|  636 | f    |    9 | 2010-12-13 | testuser |

.
.
.

| 1191 | C    |   13 | 2018-09-24 | vishnu   |
| 1192 | D    |  134 | 2017-09-13 | vishnu   |
| 1193 | E    |  112 | 2018-11-29 | vishnu   |
| 1194 | F    |  109 | 2000-02-17 | vishnu   |
| 1195 | A    |  120 | 2020-09-30 | vishnu   |
| 1196 | B    |  156 | 2001-08-22 | vishnu   |
| 1197 | C    |   13 | 2018-09-24 | vishnu   |
| 1198 | D    |  134 | 2017-09-13 | vishnu   |
| 1199 | E    |  112 | 2018-11-29 | vishnu   |
| 1200 | F    |  109 | 2000-02-17 | vishnu   |
+------+------+------+------------+----------+
1200 rows in set (0.00 sec)
```