Locks, Sleep & Wakeup

P3b Report

Chahak Tharani[1] and Deepti Rajagopal[1]

[1]University of Wisconsin-Madison

March 17, 2022

1 Introduction

This report contains the analysis of two locks in xv6, and 2 examples of how sleep() and wakeup() functions are used in xv6. The locks that have been analyzed are - ftable lock, and tickslock. The examples of sleep() and wakeup() that are analyzed are - exit() and wait(), sys_sleep() and wakeup() in timer interrupt.

2 Locks

Locks are used in order to ensure that race conditions do not occur. In the context of operating systems, race condition occurs when two threads (that share the same code and data) access the same shared data, and update it at the same time, resulting in faulty values. In this section, we explore two instances of Spinlock

2.1 Spinlock

Spinlock has 3 major methods:

- initlock - initializes the 'locked' and 'cpu' variable of the spinlock.
- acquire - loops until the lock is acquired. It disables the interrupts to prevent deadlock. It calls the Atomic Exchange method to continuously check the lock variable and sets it to 1 when it finds it to be 0. Sets the cpu variable.
- release - It resets the cpu variable, It writes 0 to the lock variable atomically by running an assembly instruction. It then re-enables the interrupts.

2.2 ftable Lock

ftable lock uses spinlock to serialize the allocation of a struct file in the file table. It is used in 3 places of file.c - filealloc(), filedup(), and fileclose(). All the open files in the system are kept in a global file table, the ftable. The filealloc() function allocates a file, the filedup() function creates a duplicate reference, and the fileclose() function releases a reference.

2.2.1 Critical Section Length Analysis

- In filealloc(), the ftable.lock is first acquired, and if for the given file, the f->ref is 0, then the f->ref is marked to 1, and the lock is released. The critical section here is 3 instructions long. The shared resource here is f->ref which can be modified by other processes that invoke filedup() or fileclose(). Hence the lock here is essential.
- In filedup(), the lock is first acquired, and then the f->ref is incremented (in order to create a duplicate reference), after which the lock is released. The critical section here is 2 instructions long. Like above, f->ref is the shared resource.
- In fileclose(), the lock is acquired, and if the f->ref value is greater than 0, then the number of references is decremented, and the lock is released. On the other hand, if f->ref is 0, then the file is closed

We created a user program to show how long these critical sections are, and how many times they are being invoked during the program. Figure 1 shows the trace of this program. As can be seen in Figure 1, calling the fork() method results in the filedup() method being called. Here, contention can happen for the resource f->ref, when both fork() and sys_dup() try to call the filedup() function. Or when multiple processes try to call fork() at the same time. Since f->ref is a shared resource, in order that f->ref reflects the correct value, a lock is put around it. When locking is added, we usually try to minimize the critical section in order to maximize concurrency. Here, since f->ref is the only shared resource, xv6 has put the lock just

```
$ testSleep1
(filedup): Critical section start, pid = 2
(filedup): Critical section end, pid = 2
(filedup): Critical section start, pid = 2
(filedup): Critical section end, pid = 2
(filedup): Critical section start, pid = 2
(filedup): Critical section end, pid = 2
(wait): Parent going to sleep, pid = 2
(filedup): Critical section start, pid = 3
(filedup): Critical section end, pid = 3
(filedup): Critical section start, pid = 3
(filedup): Critical section end, pid = 3
(filedup): Critical section start, pid = 3
(filedup): Critical section end, pid = 3
Parent waiting on child, pid = 3
(wait): Parent going to sleep, pid = 3
Child, pid = 4
(fileclose): Critical section start, pid = 4
(fileclose): Critical section end, pid = 4
(fileclose): Critical section start, pid = 4
(fileclose): Critical section end, pid = 4
(fileclose): Critical section start, pid = 4
(fileclose): Critical section end, pid = 4
(exit): Process exiting, pid = 4
Parent, pid = 3
(fileclose): Critical section start, pid = 3
(fileclose): Critical section end, pid = 3
(fileclose): Critical section start, pid = 3
(fileclose): Critical section end, pid = 3
(fileclose): Critical section start, pid = 3
(fileclose): Critical section end, pid = 3
(exit): Process exiting, pid = 3
```

Figure 1: Trace of critical sections of ftable.lock

around the instruction where f->ref is being updated. Likewise, multiple callers exist for fileclose(), like exit(), sys_open(), sys_close(), etc, and the lock here is needed for the same reasons as above, i.e., so that the f->ref is not incorrectly updated.

2.3 tickslock

Tickslock is a spinlock on clock ticks variable. It is initialized in the trap.c file in tvinit() method. It is used in 3 functions:

- trap() - Handles the interrupts. Tickslock is used while handling the timer interrupt. It acquires the tickslock, increments ticks and wakes up all processes sleeping on ticks. For this, it acquires the ptable lock and checks all sleeping processes if they are sleeping on ticks variable, it changes their state to RUNNABLE and finally releases the ptable and tickslock. The timer interrupt handler also increments the ticks variable. If sys_sleep holds tickslock and timer interrupt would occur, a deadlock condition can arise since interrupt handler is waiting for tickslock to be released, but sys_sleep cannot continue until timer interrupt returns. To prevent this deadlock situation, XV6 conservatively disables interrupts while acquiring any spinlock and re-enables them when the spinlock is released.

- sys_sleep() - This is the system call for sleeping for some number of clock ticks. It acquires the tickslock and stores the current ticks in a variable and compares this variable to the current ticks till the difference is less than the sleep time given. When the gap increases from the sleep time, it releases the lock. When in while condition, it calls sleep() method, which releases the tickslock and causes the process to sleep. This method acquires the ptable lock to change the state of the process to SLEEPING, sleep on ticks channel, and returns control to scheduler. Finally it releases the ptable lock and reacquires tickslock.

- sys_uptime() - This method acquires the tickslock to get the correct value of the ticks variable and then releases the tickslock.

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

int pid1, pid2, pid3;
void process1() {
    printf(1, "Starting process %d\n", getpid());
    sleep(3);
    uptime();
    printf(1, "Ending process %d\n", getpid());
    exit();
}

void process2() {
    printf(1, "Starting process %d\n", getpid());
    uptime();
    sleep(1);
    uptime();
    printf(1, "Ending process %d\n", getpid());
    exit();
}

// Test file to trace the behaviour of tickslock
int main(int argc, char* argv[]) {
    int pid0 = getpid();
       pid1 = fork();
    if (pid1 < 0) {
        printf(2, "Fork child process 1 failed\n");
        exit();
    } else if (pid1 == 0) { // child process 1
        process1();
    }

    pid2 = fork();
    if (pid2 < 0) {
        printf(2, "Fork child process 2 failed\n");
        exit();
    } else if (pid2 == 0) { // child process 2
        process2();
    }

    sleep(5);
    printf(1,"Ending process %d\n", pid0);
    exit();
}
```

Figure 2: Program for analyzing tickslock and associated sleep and wakeup methods

2.3.1 Critical Section length analysis for Tickslock:

- sys_sleep() - After acquiring the tickslock, 1 instruction is to set the ticks variable, then we check in a while loop if sleep time has exhausted, within this we also check if the process is killed, in which case the critical section ends and the lock is released. Otherwise, we call sleep, where we have 3 instruction before checking the condition if

```
1  Booting from Hard Disk..xv6...
2  cpu0: starting 0
3  sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
4  init: starting sh
5  $ trace
6  Critical section begin in sysproc.c, sys_sleep for process 3
7  Starting process 4
8  Critical section begin in sysproc.c, sys_sleep for process 4
9  Starting process 5
10
11 Critical section begin in sysproc.c, sys_uptime for process 5
12 uptime is 255
13 Critical section end in sys_uptime for process 5
14
15 Timer interrupt caused wakeup for process 3, sleep time elapsed 1 out of 5
16 Timer interrupt caused wakeup for process 4, sleep time elapsed 1 out of 3
17 Critical section begin in sysproc.c, sys_sleep for process 5
18 Timer interrupt caused wakeup for process 3, sleep time elapsed 2 out of 5
19 Timer interrupt caused wakeup for process 4, sleep time elapsed 2 out of 3
20 Timer interrupt caused wakeup for process 5, sleep time elapsed 1 out of 1
21 Sleep completed, Critical section end in sys_sleep for process 5
22
23 Critical section begin in sysproc.c, sys_uptime for process 5
24 uptime is 257
25 Critical section end in sys_uptime for process 5
26
27 Ending process 5
28 Timer interrupt caused wakeup for process 3, sleep time elapsed 3 out of 5
29 Timer interrupt caused wakeup for process 4, sleep time elapsed 3 out of 3
30 Sleep completed, Critical section end in sys_sleep for process 4
31
32 Critical section begin in sysproc.c, sys_uptime for process 4
33 uptime is 258
34 Critical section end in sys_uptime for process 4
35
36 Ending process 4
37 Timer interrupt caused wakeup for process 3, sleep time elapsed 4 out of 5
38 Timer interrupt caused wakeup for process 3, sleep time elapsed 5 out of 5
39 Sleep completed, Critical section end in sys_sleep for process 3
40 Ending process 3
```

Figure 3: Analysis for sys_sleep(), sys_uptime() and trap() calls for timer interrupt

the lock is same as ptable lock, if not equal it acquires the ptable lock and release tickslock. The critical section ends here. After waking up, it rechecks from the while loop condition. So, in total there are 8 instruction in the critical section.

- sys_uptime() - Between acquiring and releasing the tickslock, this function has only 1 instruction for setting the ticks variable.

- timer interrupt - We increment the ticks in the critical section. And then acquire ptable lock and wakeup all processes sleeping on ticks channel and then release the ptable lock. So in worst case, NPROC processes are sleeping, 2*NPROC instructions for checking the condition and resetting the state. 2 instructions for acquiring and releasing the ptable lock and 1 instruction for incrementing the ticks variable.

3 Trace Analysis for tickslock and associated sleep and wakeup calls

For tracing the behaviour of tickslock and sleep and wakeup calls in sys_sleep, sys_uptime and timer interrupt methods, we have written a test (trace.c) which creates 2 processes by fork() method and traces the above system calls. This setup runs on 1 CPU.

- Trace: Line 6: All 3 processes are runnable at the start and all are contending for the tickslock. Pid 3 acquires it and sys_sleep() system call is called for main() process (pid 3). This process then goes to sleep and releases the tickslock and process with pid 4 starts, acquires the tickslock, releases it and goes to sleep, then process 5 starts and calculates the uptime after acquiring the tickslock and then releases it.

- Observation: We observe that since acquiring the tickslock disables the timer interrupt, the running processes do not switch in between acquiring the tickslock and the process going to sleep. Similarly, we do not see any process switch when a process calculates the uptime. This disabling of interrupt prevents deadlock since timer interrupt also acquires the tickslock.

- Trace: Line 15: At this instant, process 5 and the timer interrupt are contending for tickslock. Timer interrupt acquires it and wakes the sleeping processes on the ticks channel (i.e. pid 3 and 4) , pid 3 acquires the tickslock, checks if its sleep time has exhausted, since it hasn't it releases the lock and goes to sleep. Similar thing happens for process 4. Then pid 5 acquires and goes to sleep. In the next timer interrupt all 3 processes are woken up, the sleep time of pid 5 is exhausted and it completes its critical section and releases the tickslock. It requires the tickslock to check the uptime, so the timer interrupt cannot occur. Process 5 ends and similar behaviour is followed by pid 3 and pid 4.

- Observation: We observe that all processes change their state from RUNNING to SLEEPING and go to sleep on ticks channel after sys_sleep() system call. When a timer interrupt occurs, all processes sleeping on ticks channel are woken up, made RUNNABLE from SLEEPING and they recheck their sleep condition. A timer interrupt can only occur when no process is holding the tickslock, so it can acquire it when the interrupt occurs. All processes woken up on the timer interrupt are RUNNABLE and will reaquire the ticks lock to check their sleep condition. Whichever process is chosen by the scheduler tries to acquire the tickslock, but a timer interrupt can occur before it and acquire the lock. There will be no contention for tickslock once a particular process has acquired the lock since now now the scheduler cannot switch the process because timer interrupt cannot occur in the first place.

4 Sleep & Wakeup

Sleep and Wakeup is used in operating systems, so that when a lock is acquired by a process (say, process A), the contending process (say, process B), instead of spinning (as in spinlock), puts itself to sleep (by giving up the CPU), and is awoken by process A, once it has completed execution and is ready to release its lock. This is a more resourceful way of utilizing the CPU, rather than process B just spinning indefinitely until process A gives up the lock. In this section, we explore two such instances of how sleep() and wakeup() functions are used. Before that, we detail out what sleep() and wakeup() do internally.

The sleep function marks the current process as SLEEPING and then calls sched to release the CPU. The wakeup function looks for a process sleeping on the given wait channel and marks it as RUNNABLE. Sleep acquires p->lock and releases lk, because it no longer needs to hold lk once it has p->lock. Wakeup also waits to acquire p->lock, so the wakeup will not miss the sleep. In the case that lk is the same as p->lock (for example, in wait), it means that sleep already has p->lock, and hence doesn't need to release it until the process state is marked as SLEEPING.

Wakeup loops over the process table and acquires p->lock of each process in the loop. When it finds a process is SLEEPING with a matching chan (the channel on which the process was sleeping), it changes that process's state to RUNNABLE. The next time the scheduler runs, it will see that the process is ready to be run. Wakeup will then see the sleeping process and wake it up. All processes on the same channel are awoken by a single wakeup call.

4.1 exit() & wait()

The exit() method closes all file descriptors of the process, and then acquires a lock to wake up the parent using the wakeup1() routine. It then goes ahead and marks itself as ZOMBIE and jumps into the scheduler, never to return. It is important to note that the process doesn't die immediately. Instead, the process marks itself as ZOMBIE, and the parent process would perform the cleanup in the wait() function call.

The wait() method checks if any of the child processes are in ZOMBIE state, and if yes, goes ahead and marks them as UNUSED, and releases the lock. On the other hand, if none of the children are in ZOMBIE state, it means that the exit() method has not yet been called by the children, and hence the parent has to go to sleep using the sleep() method, only to be awoken by the exit() invocation by the child. One important thing to note is that once the child is in ZOMBIE state, the parent process does the cleanup for the child process. Also, in the event that the parent exits before the child, we still need to ensure that the child process has been cleaned up. In order for this to happen, when the parent exits before the child, the init process adopts the children, and performs the cleanup for the child processes.

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

int
main(int argc, char *argv[]) {
        int rc = fork();
        if(rc == 0) {
                sleep(1000);
                printf(1, "Child, pid = %d\n", getpid());
                exit();
        } else if(rc > 0) {
                printf(1, "Parent waiting on child, pid = %d\n", getpid());
                wait();
                printf(1, "Parent, pid = %d\n", getpid());
        } else {
                printf(1, "Fork failed");
        }
        exit();
        return 0;
}
~
```

Figure 4: Program using exit() and wait()

```
$ testSleep1
(wait): Parent going to sleep, pid = 2
Parent waiting on child, pid = 3
(wait): Parent going to sleep, pid = 3
Child, pid = 4
(exit): Process exiting, pid = 4
Parent, pid = 3
(exit): Process exiting, pid = 3
$ _
```

Figure 5: Trace of exit() and wait() for parent and child process

We created a user program that has a parent process and a child process in order to analyse the behaviour of sleep and wakeup in the exit and wait functions. We also added printf statements inside the functions of exit and wait. In the program shown in Figure 2, the parent process waits for the child to exit.

Figure 3 shows the trace of the above program. As can be seen in the figure, the parent is put to sleep first upon calling the wait function. Once the child process has finished executing (pid 4), it calls the exit() function, which wakes up all sleeping processes on the same channel, hence waking up the parent (pid 3), and then the parent wakes up and prints.

It is important to note that if the child process were to start running first, then the wait() called by the parent would not result in the parent sleeping. Instead, the parent sees that the child is marked to ZOMBIE state, so the parent cleans up the child, and then can continue to execute itself.

4.2 sys_sleep() and trap()

The sleep() method called in sys_sleep() acquires the ptable lock while holding the tickslock. Holding tickslock is necessary as it ensures that no other process could start a call to wakeup(chan). Now that sleep holds ptable lock, it is safe to release tickslock: some other process may start a call to wakeup(ticks), but wakeup will wait to acquire ptable lock, and thus will wait until sleep has finished putting the process to sleep, keeping the wakeup from missing the sleep. It then releases the ticks lock and changes the state to SLEEPING to prevent busy waiting.
When a timer interrupt happens, all processes sleeping on ticks channel are woken up. Wakeup loops over the process table. It acquires the ptable lock of each process it inspects, because it may manipulate that process's state and because ptable lock ensures that sleep and wakeup do not miss each other. When wakeup finds a process in state SLEEPING on ticks channel, it changes that process's state to RUNNABLE. The next time the scheduler runs, it will see that the process is ready to be run.