CS 140 Lab Report 7

Dale Sealtiel T. Flores
2023-11373
THX/WXY

1. Illustrate how *not* calling `__sync_synchronize` in `acquire` may result in incorrect behavior with a walkthrough of a possible sequence of instructions. Cite all references used

Answer: Using the comments as a guide, `__sync_synchronize()` acts as a fence instruction like in RISC-V. It tells the compiler and CPU to not reorder loads or stores across the barrier (Cox et al., 2022).

An example is also given in the book, suppose that we have `push`

```c
l = malloc(sizeof *l);
l->data = data;
acquire(&listlock);
l->next = list;
list = l;
release(&listlock);
```

Suppose that you are running a multi CPU system (assume that `__sync_synchronize` is *not* called in `acquire`), where in the system reorders the instructions for CPU 1 (for optimization) in such a way that `line 4` ran after `line 6`.

Reordered `push` for CPU 1

```c
l = malloc(sizeof *l);
l->data = data;
acquire(&listlock);
list = l;
release(&listlock);
l->next = list;
```

Table 1: Problem encountered when not calling `__sync_synchronize`

CPU 1	CPU 2	Notes
`acquire(&listlock)`	`idle`	
`list = l`	`idle`	
`release(&listlock)`	`idle`	CPU 2 will acquire the lock
`idle`	`acquire(&listlock)`	
`idle`	`l->next = list`	`list` here is the newly initialized `l` in CPU 1
`idle`	`release(&listlock)`	CPU 1 will run its `line 6`
`l->next = list`	`idle`	`l->next` will not point to the wrong list

This would result to a wrong output for CPU 2 since the `l->next` will be pointing to the newly initialized `l` in CPU 1 which means that the old list that we want to push to in CPU 1 will be gone. since

2. Determine whether sleep locks disable interrupts while in a critical section between `acquiresleep` and `releasesleep`. If so, explain why it is necessary. Otherwise, explain why this is not done.

Answer: In xv6, sleep locks **do not** disable interrupts. Since you cannot yield while holding a spinlock, xv6 must find a way to implement a lock that allows a process to hold the lock while also allowing yield and interrupts. That is why xv6 implemented `sleeplock`.
Sleeplocks has a `locked` field that is protected by a spinloc,. When `acquiresleep` is called, it puts the process to sleep and yields the CPU and releases the spinlock.

3. Explain why failing to call `wakeup` in `releasesleep` is problematic.

Answer: Failing to call `wakeup` in `releasesleep` will result to processes that are always sleeping. Since in `acquiresleep`, it calls `sleep` which puts the processes to sleep, if `wakeup` is not called in `releasesleep`, the sleeping process will never be woken up and thus stay permanently asleep. (Cox et al., 2022)

4. Given that xv6 has a scheduler for each CPU, show which parts of the xv6 code ensure that it is impossible for a process to be scheduled on more than one CPU at a time.

Answer: If we look at `proc.c` where the implementation of `scheduler(void)` lies, we can see that `acquire(&p->lock)` is called before checking if the process is `RUNNABLE`. Having locks ensures that only one CPU can access the process at a time. If a process has already acquired the lock, other CPUs that want to access the process will have to wait until `release(&p->lock)` has been called before the process can be scheduled again.

5. For the following items, refer to the spin lock removal of activity in Section 2.4
   (a) Provide screenshots of failing `usertests` runs along with the QEMU command used when it was encountered, then provide a detailed illustration of how the `struct run` pointed to by `freelist` becomes problematic in relation to this.

Answer: In my 10 runs, I encountered 5 failing `usertests` runs.

Table 2: Failed `usertests` runs

QEMU command	Failing usertests run
`$ usertests -q` `usertests starting`	`FAILED -- lost some free pages 32432 (out of 32434)`
`$ usertests -q` `usertests starting`	`FAILED -- lost some free pages 32432 (out of 32433)` `$`
`$ usertests -q` `usertests starting`	`FAILED -- lost some free pages 32431 (out of 32432)`
`$ usertests -q` `usertests starting`	`FAILED -- lost some free pages 32427 (out of 32431)`
`$ usertests -q` `usertests starting`	`FAILED -- lost some free pages 32426 (out of 32427)`

Commenting out `acquire(&kmem.lock)` and `release(&kmem.lock)` in `kalloc` can cause multiple CPUs to access the same `struct run *r` if both CPUs do `r = kmem.freelist` at the same time. It will lead to both CPUs having the same `r` value and thus accessing the same page.
Suppose that we have 2 CPUs doing `kalloc` at the same time. This scenario could happen:

Table 3: Scenario when 2 CPUs call `kalloc` at the same time

CPU 1	CPU 2	Notes
`r = kmem.freelist`	`r = kmem.freelist`	`r` will be point to the same `struct run`
`kmem.freelist = r->next`	`kmem.freelist = r->next`	`kmem.freelist` will be updated to the same value

This will lead to both CPUs accessing the same page which will lead to overwriting of data and missing pages.

(b) Determine whether the `kalloc` lock is necessary if there is only a single CPU, and explain why or why not

Answer: Since there is only a single CPU, calling the lock in `kalloc` is not necessary. This is because there will be no other CPU that will be accessing `freelist` at the same time. Therefore, it is not necessary to have a `kalloc` lock.

6. For the following items, refer to the console write synchronization exercise in Section 2.5.2
   (a) Show all relevant changes made (with corresponding filenames) via code screenshots or snippets for the exercise in Section 2.5.2, and briefly describe what each change does. Ensure that your changes are properly committed and pushed to your Github Classroom repository.

Answer:

```
+  28 struct sleeplock sleeplock;
```

Figure 1: define a sleep lock in `kernel/console.c`

We define a sleep lock in `kernel/console.c` to be used in `consolewrite`.

```
+ 188   initsleeplock(&sleeplock, "conssleeplock");
```

Figure 2: initialize the sleep lock in `consolewrite` in `kernel/console.c`

We initialize the sleep lock in `consoleinit` using `initsleeplock(&sleeplock, "conssleeplock")`. This will initialize the sleep lock in console so that it can be used in `consolewrite`.

```
+  64   acquiresleep(&sleeplock);
   65   for(i = 0; i < n; i++){
   66     char c;
   67     if(either_copyin(&c, user_src, src+i, 1) == -1)
   68       break;
   69     uartputc(c);
   70   }
+  71   releasesleep(&sleeplock);
```

Figure 3: `acquiresleep` and `releasesleep` in `consolewrite` in `kernel/console.c`

We wrap the critical section in `consolewrite` with `acquiresleep(&sleeplock)` and `releasesleep(&sleeplock)` in order to ensure that their console writes are finished without allowing other processes to interleave their console writes.

(b) Explain why the use of spin lock instead of a sleep lock for the exercise in is potentially problematic. Explicitly show relevant code blocks especially from `kernel/uart.c`.

Answer: Since in `consolewrite`, it calls `uartputc`. We will look at the code of `uartputc`.

```
void
uartputc(int c)
{
  if(panicking == 0)
    acquire(&uart_tx_lock);

  if(panicked){
    for(;;)
      ;
  }
  while(uart_tx_w == uart_tx_r + UART_TX_BUF_SIZE){
    // buffer is full.
    // wait for uartstart() to open up space in the buffer.
    sleep(&uart_tx_r, &uart_tx_lock);
  }
  uart_tx_buf[uart_tx_w % UART_TX_BUF_SIZE] = c;
  uart_tx_w += 1;
  uartstart();
  if(panicking == 0)
    release(&uart_tx_lock);
}
```

Figure 4: `uartputc` in `kernel/uart.c`

We can observe here that `uartputc` calls `uartstart`. So let's lookat the code of `uartstart`.

```
void
uartstart()
{
  while(1){
    if(uart_tx_w == uart_tx_r){
      // transmit buffer is empty.
      return;
    }

    if((ReadReg(LSR) & LSR_TX_IDLE) == 0){
      // the UART transmit holding register is full,
      // so we cannot give it another byte.
      // it will interrupt when it's ready for a new byte.
      return;
    }

    int c = uart_tx_buf[uart_tx_r % UART_TX_BUF_SIZE];
    uart_tx_r += 1;

    // maybe uartputc() is waiting for space in the buffer.
    wakeup(&uart_tx_r);

    WriteReg(THR, c);
  }
}
```

Figure 5: `uartstart` in `kernel/uart.c`

We can see here that `uartstart` interrupts when it's ready for a new byte. According to the xv6 book (Cox et al., 2022), spin locks do not allow interrupts while holding the lock. This means that if it will interrupt, it will cause a problem since the implementation of spin locks in xv6 is more conservative. When a CPU calls `acquire`, it will always disable interrupts on that CPU.

Bibliography

Cox, R., Kaashoek, F., & Morris, R. (2022,). *xv6: a simple, Unix-like teaching operating system*. https://pdos.csail.mit.edu/6.828/2023/xv6/book-riscv-rev3.pdf