OS Homework-3 Salman Zafar

Ikhlas Ahmed Khan

27096

1. **Why do we need push_up and pop_up on top of intr_on and intr_off?**

   Ans) We need push_up in acquire to disable all interrupts before entering the while loop inside acquire, which handles the locking. Similarly, we need pop_up after releasing the lock in the release function to re-enable interrupts. This is done to prevent any timer interrupt from occurring between the lock acquisition and its release. If a timer interrupt were to occur, it would itself acquire &tickslock to increment the ticks counter. If another user process acquires a lock during this stage, it would cause a deadlock. To prevent this, push_up and pop_up are used to disable and enable interrupts, respectively.

2. **What is the difference between locks and semaphores usage-wise?**
   Ans) Locks are mainly useful in scenarios where the expected time a thread will spend waiting for the lock to be released is short. For example, if a process acquires a lock, it continuously checks in a while loop to see if the lock becomes available. For short critical sections where a lock is held briefly, this approach is efficient. However, if a process has to wait for a longer period, this busy waiting can be inefficient and wasteful.
   In contrast, semaphores are designed to avoid busy waiting in situations where the expected waiting time may be prolonged. In such cases, xv6 uses sleep/wakeup calls with semaphores, putting the waiting process into a sleep state. This allows the respective process to pause without consuming CPU resources, improving efficiency and resource management.

LOCK	**SEMAPHORE**
locks can be used only for mutual exclusion.	Semaphores can be used either for mutual exclusion or as a counting semaphore.
A lock is a low-level synchronization mechanism.	A semaphore is a signaling mechanism.

locks allow only one process at any given time to access the critical section.	Semaphores allow more than one process at any given time to access the critical section.
locks can be wasteful if they are hold for a long time duration.	In semaphore there is no resource wastage of process time and resources.
Only one thread is allowed at a time to acquire the lock and proceed with a critical section.	One or several threads are allowed to access the critical section.
locks are very efficient because they are blocked only for a short period of time.	Semaphores are held for a longer period of time. To access its control structure it uses spin lock.
In locks, a process waiting for lock will keep the processor busy by continuously polling the lock.	In semaphore, a process is waiting for a semaphore to go into sleep to be woken up at any time and then try for the lock again.
locks are valid for only one process.	Semaphores can be used to synchronize between different processes.
In locks, a process waiting for lock will instantly get access to a critical region as the process will poll continuously for the lock.	In semaphore, a process waiting for a lock might not get into the critical region as soon as the lock is free because the process would have gone to sleep and when it is woken up it will enter into the critical section.
It is a busy wait process.	It is a sleep wakeup process.
locks can have only two values – LOCKED and UNLOCKED	In semaphore, mutex will have value 1 or 0, but if used as counting semaphore it can have different values.

In uniprocessor system locks are not very useful because they will keep the processor busy every time while polling for the lock , thus disabling any other process from running.	In a uniprocessor system semaphores are convenient because they don't keep the processor busy while waiting for the lock.
In locks it is recommended to disable the interrupts while holding the locks.	Semaphore can be locked with interrupt enabled.
Thread cannot sleep while waiting for the lock when it fails to get the lock, but it continues a loop of trying to get locked.	Thread goes to sleep waiting for the lock when it fails to get the lock.

3. **How many times in the code so far locks and semaphores have occurred?**
Ans) In xv6, locks are acquired and released in various key areas, gaving occurrences in 14 different files. These can be found in the following files: (note in brackets i have mentioned that how many times locks have occurred in the following codes)

   1. sleeplock.c (3 times)

kernel/sleeplock.c

```
void
acquiresleep(struct sleeplock *lk)
{
  acquire(&lk->lk);

  release(&lk->lk);

void
releasesleep(struct sleeplock *lk)
{
  acquire(&lk->lk);

  release(&lk->lk);

  acquire(&lk->lk);

  release(&lk->lk);
```

2. bio.c(7)

kernel/bio.c

```
63    acquire(&bcache.lock);
68        b->refcnt++;
69        release(&bcache.lock);
70        acquiresleep(&b->lock);
83        release(&bcache.lock);
84        acquiresleep(&b->lock);
114 // Release a locked buffer.
122   releasesleep(&b->lock);
123
124   acquire(&bcache.lock);
125   b->refcnt--;
136   release(&bcache.lock);
141   acquire(&bcache.lock);
143   release(&bcache.lock);
148   acquire(&bcache.lock);
150   release(&bcache.lock);
```

3. log.c(4)

kernel/log.c

```
acquire(&log.lock);
    release(&log.lock);
acquire(&log.lock);
release(&log.lock);
  commit();
  acquire(&log.lock);
  log.committing = 0;
  wakeup(&log);
  release(&log.lock);
}
acquire(&log.lock);
release(&log.lock);
```

4. pipe.c(5)

kernel/pipe.c

```
61   acquire(&pi->lock);
70     release(&pi->lock);
73     release(&pi->lock);
81
82   acquire(&pi->lock);
83   while(i < n){
84     if(pi->readopen == 0 || killed(pr)){
85       release(&pi->lock);
86       return -1;
100  release(&pi->lock);
112  acquire(&pi->lock);
115      release(&pi->lock);
128  release(&pi->lock);
```

5. file.c(4)

kernel/file.c

```c
  acquire(&ftable.lock);
      release(&ftable.lock);
  release(&ftable.lock);
{
  acquire(&ftable.lock);
  if(f->ref < 1)
  f->ref++;
  release(&ftable.lock);
  return f;
  acquire(&ftable.lock);
    release(&ftable.lock);
  release(&ftable.lock);
```

6. proc.c (16 locks)

```
26  // must be acquired before any p->lock.
97    acquire(&pid_lock);
100   release(&pid_lock);
115     acquire(&p->lock);
119       release(&p->lock);
131     release(&p->lock);
139     release(&p->lock);
254   release(&p->lock);
294     release(&np->lock);
315   release(&np->lock);
317   acquire(&wait_lock);
319   release(&wait_lock);
321   acquire(&np->lock);
323   release(&np->lock);
368   acquire(&wait_lock);
376   acquire(&p->lock);
381   release(&wait_lock);
397   acquire(&wait_lock);
405         acquire(&pp->lock);
413             release(&pp->lock);
556   // (wakeup locks p->lock),
557   // so it's okay to release lk.
558

558
559   acquire(&p->lock);  //DOC: sleeplock1
560   release(lk);
```

7. kalloc.c (2 locks)

kernel/kalloc.c

```c

  acquire(&kmem.lock);
  r->next = kmem.freelist;
  kmem.freelist = r;
  release(&kmem.lock);
}
  acquire(&kmem.lock);
  release(&kmem.lock);
```

8. sysproc.c (3)

kernel/sysproc.c

```c
  acquire(&tickslock);
      release(&tickslock);
  release(&tickslock);
{
  uint xticks;

  acquire(&tickslock);
  xticks = ticks;
  release(&tickslock);
  return xticks;
```

9. console.c(3)

kernel/console.c

```c
target = n;
acquire(&cons.lock);
while(n > 0){
    if(killed(myproc())){
      release(&cons.lock);
      return -1;
release(&cons.lock);
acquire(&cons.lock);
release(&cons.lock);
```

10. fs.c(7)

kernel/fs.c

```
251   acquire(&itable.lock);
258       release(&itable.lock);
274   release(&itable.lock);
284   acquire(&itable.lock);
286   release(&itable.lock);
301   acquiresleep(&ip->lock);
326   releasesleep(&ip->lock);
339   acquire(&itable.lock);
345     // so this acquiresleep() won't block (or deadlock).
346     acquiresleep(&ip->lock);
347
348     release(&itable.lock);
349
355     releasesleep(&ip->lock);
356
357     acquire(&itable.lock);
358   }
361   release(&itable.lock);
```

11. uart.c(2 locks)

kernel/uart.c

```
  acquire(&uart_tx_lock);
  release(&uart_tx_lock);
  }

  // send buffered characters.
  acquire(&uart_tx_lock);
  uartstart();
  release(&uart_tx_lock);
}
```

12. printf.c(1 lock)

kernel/printf.c

```
  if(locking)
    acquire(&pr.lock);

  if(locking)
    release(&pr.lock);

```

13. trap.c(1 lock)

kernel/trap.c

```c
clockintr()
{
  if(cpuid() == 0){
    acquire(&tickslock);
    ticks++;
    wakeup(&ticks);
    release(&tickslock);
```

14. virtio_disk.c (2 lock)

kernel/virtio_disk.c

```c
  acquire(&disk.vdisk_lock);


  release(&disk.vdisk_lock);
}
{
  acquire(&disk.vdisk_lock);


  release(&disk.vdisk_lock);
```

You can view all these instances of `acquire` in the following search results: [Lock occurrences in xv6 source]
https://github.com/search?q=repo%3Amit-pdos%2Fxv6-riscv+acquire+release&type=code

For semaphores, `sleep` and `wakeup` functions appear

proc.c(2 times baqi tou sleep and wakeup are only defined here)

kernel/proc.c

```
// helps ensure that wakeups of wait()ing
      wakeup(initproc);
  // Parent might be sleeping in wait().
  wakeup(p->parent);
    sleep(p, &wait_lock);  //DOC: wait-sleep
    // regular process (e.g., because it calls sleep), and thus cannot
// Atomically release lock and sleep on chan.
// Reacquires lock when awakened.
void
sleep(void *chan, struct spinlock *lk)
{
  struct proc *p = myproc();

  // guaranteed that we won't miss any wakeup
  // (wakeup locks p->lock),
  acquire(&p->lock);  //DOC: sleeplock1
  // Go to sleep.
  p->state = SLEEPING;
// Wake up all processes sleeping on chan.
wakeup(void *chan)
      if(p->state == SLEEPING && p->chan == chan) {
      if(p->state == SLEEPING){
        // Wake process from sleep().
  [SLEEPING]  "sleep ",
```

Sleeplock.c (1 semaphore found)

kernel/sleeplock.c

```c
// Sleeping locks
#include "sleeplock.h"
initsleeplock(struct sleeplock *lk, char *name)
  initlock(&lk->lk, "sleep lock");
acquiresleep(struct sleeplock *lk)
    sleep(lk, &lk->lk);
releasesleep(struct sleeplock *lk)
  wakeup(lk);
}

int
holdingsleep(struct sleeplock *lk)
{
  int r;

```

Pipe.c(4 times)

kernel/pipe.c

```c
#include "sleeplock.h"
    wakeup(&pi->nread);
    wakeup(&pi->nwrite);
      return -1;
    }
    if(pi->nwrite == pi->nread + PIPESIZE){ //DOC: pipewrite-full
      wakeup(&pi->nread);
      sleep(&pi->nwrite, &pi->lock);
    } else {
      char ch;
  wakeup(&pi->nread);
    sleep(&pi->nread, &pi->lock); //DOC: piperead-sleep
  wakeup(&pi->nwrite);  //DOC: piperead-wakeup
```

Show less

Log.c(2 times)

kernel/log.c

```
6    #include "sleeplock.h"
22   // sleeps until the last outstanding end_op() commits.
132        sleep(&log, &log.lock);
135        sleep(&log, &log.lock);
162      wakeup(&log);
167      // call commit w/o holding locks, since not allowed
168      // to sleep with locks.
169      commit();
171      log.committing = 0;
172      wakeup(&log);
173      release(&log.lock);
```

Show less

Virtio_disk.c(2 times)

kernel/virtio_disk.c

```
14   #include "sleeplock.h"
181    wakeup(&disk.free[0]);
232      sleep(&disk.free[0], &disk.vdisk_lock);
284    while(b->disk == 1) {
285      sleep(b, &disk.vdisk_lock);
286    }
320      b->disk = 0;   // disk is done with buf
321      wakeup(b);
322
```

Console.c(1 time)

kernel/console.c

```c
#include "sleeplock.h"

      }
      sleep(&cons.r, &cons.lock);
    }

        cons.w = cons.e;
        wakeup(&cons.r);
      }
```

Uart.c(1 time)

kernel/uart.c

```c
    // wait for uartstart() to open up space in the buffer.
    sleep(&uart_tx_r, &uart_tx_lock);
  }

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

```


You can explore the occurrences of `sleep` and `wakeup` in the following link:
https://github.com/search?q=repo%3Amit-pdos%2Fxv6-riscv+sleep+wakeup&type=code

4. How can we use locks and semaphores to ensure a process 2 running before a process 1?

Ans)

PROCESS 1

Acquire(&s→lock) released in sleep
sleep (s, &s→lock)
(PROCESS 1 CODE HERE AFTER IT WAKES UP)
Release(&s→lock)

PROCESS 2

Acquire (Acquire(&s→lock)
(PROCESS 2 CODE HERE)
wakeup(s)
Release (Acquire(&s→lock)

THE ABOVE CODE ENSURE PROCESS 2 RUNS BEFORE PROCESS 1…

Explanation:

Process 1 (Parent):

- It acquires a lock and then goes to sleep, meaning it must be woken up by another process to continue execution. The sleep function releases that respective lock, allowing Process 2 (the child) to acquire it and perform its tasks. After process 2 wakes up process 1 it continues its work and then releases the lock.

Process 2 (Child):

- After being scheduled, it acquires the lock, performs its work, and calls wakeup() to wake up Process 1 and release its lock. Once Process 1 is awake, it continues its execution and releases its lock and completes execution.

This coordination ensures that Process 2 completes its work before Process 1.