Locking

Operating system

Introduction

- Race condition
- What is lock?
- Lock implementation
  - Atomic Operation
  - Deadlock
  - Interrupt handler
  - Memory ordering
- Lock usage in xv6
  - usual/unusual case
  - In uniprocessor system

Start from race condition

- Any code that accesses shared data concurrently from multiple CPUs is likely to yield incorrect results or a broken data structure.

Parts of the program where the shared resource is accessed is protected. This protected section is the **critical section** or **critical region.**

```
// Increment ref count for file f.
struct file*
filedup(struct file *f)
{
  if(f->ref < 1)
    panic("filedup");
  f->ref++;
  return f;
}
```

filedup function in file.c

Start from race condition

Thread 1	Thread 2		Integer value
			0
read value		←	0
increase value			0
write back		→	1
	read value	←	1
	increase value		1
	write back	→	2

Correct result

Thread 1	Thread 2		Integer value
			0
read value		←	0
	read value	←	0
increase value			0
	increase value		0
write back		→	1
	write back	→	1

Incorrect result

What is Lock?

- a **lock** is a synchronization mechanism for enforcing limits on access to a resource in an environment where there are many threads of execution.
- Locks ensure mutual exclusion so that only one CPU can execute the code at a time.
- So locks can be used to protect critical section and to prevent race condition.

What is Lock? (Cont'd)

Only one CPU can execute the code at a time

```
// Increment ref count for file f.
struct file*
filedup(struct file *f)
{
  acquire(&ftable.lock);
  if(f->ref < 1)
    panic("filedup");
  f->ref++;
  release(&ftable.lock);
  return f;
}
```

filedup function in file.c

Simple lock code. is it working?

No, We use shared data(lk->locked)
so it can cause race condition!

So, We should execute this
instructions(line 31, 32, 55)
atomically.

```
24 void
25 acquire(struct spinlock *lk)
26 {
27   if(holding(lk))
28     panic("acquire");
29
30   for(;;){
31     if (!lk->locked) {
32       lk->locked = 1;
33       break;
34     }
35   }
36   // Record info about lock acquisition for debugging.
37   lk->cpu = cpu;
38   getcallerpcs(&lk, lk->pcs);
39 }
```

```
46 void
47 release(struct spinlock *lk)
48 {
49   if(!holding(lk))
50     panic("release");
51
52   lk->pcs[0] = 0;
53   lk->cpu = 0;
54
55   lk->locked = 0;
56
57 }
```

Simple lock code. is it working? (Cont'd)

Not likely. Because we didn't consider some issues.

```
void
acquire(struct spinlock *lk)
{
  if(holding(lk))
    panic("acquire");

  // The xchg is atomic.
  while(xchg(&lk->locked, 1) != 0)
    ;

  // Record info about lock acquisition for debugging.
  lk->cpu = cpu;
  getcallerpcs(&lk, lk->pcs);
}
```

```
void
release(struct spinlock *lk)
{
  if(!holding(lk))
    panic("release");

  lk->pcs[0] = 0;
  lk->cpu = 0;

// equivalent to lk->locked = 0.
// This code can't use a C assignment
// since it might not be atomic.
  asm volatile("movl $0, %0" : "+m"
(lk->locked) : );
}
```

Deadlock

- Let’s say two code paths in xv6 needs locks A and B
- What happened when two CPUs run each code paths concurrently?

```
void
function1()
{
  acquire(&A.lock);
  acquire(&B.lock);
  // execute critical section code
  release(&B.lock);
  release(&A.lock);
```

I already acquired lock A and I am waiting for lock B

```
void
function2()
{
  acquire(&B.lock);
  acquire(&A.lock);
  // execute critical section code
  release(&A.lock);
  release(&B.lock);
}
```

I already acquired lock B and I am waiting for lock A

Dreadful Deadlock!

- We are stucked at the code line 28, and we cannot progress forever.
- Like this!

Deadlock prevention

- Therefore, all code paths must acquire locks in the same order.

```
void
function1()
{
  acquire(&A.lock);
  acquire(&B.lock);
  // execute critical section code
  release(&B.lock);
  release(&A.lock);
}
```

```
void
function2()
{
  acquire(&A.lock);
  acquire(&B.lock);
  // execute critical section code
  release(&B.lock);
  release(&A.lock);
}
```

- Xv6 has few lock-order chains and the longest chains are ony two deep.
- Moreover, xv6 always keep above schema to prevent deadlock.

You should consider interrupt too.

- Sometimes, interrupt handling can cause deadlock

```
void
iderw(struct buf *b)
{
  struct buf **pp;
...
  acquire(&idelock);
  // Append b to idequeue.
  // Start disk if necessary.
  // Wait for request to finish.
  while((b->flags &
(B_VALID|B_DIRTY)) != B_VALID){
    sleep(b, &idelock);
  }

  release(&idelock);
```

1. Acquire idelock

iderw function in ide.c

```
void
ideintr(void)
{
  struct buf *b;

  acquire(&idelock);
...
  // Read data if needed.
  // Wake process waiting for this
buf.
  // Start disk on next buf in
queue.
  release(&idelock);
```

2. IDE Interrupt is occurred and trap calls ideintr

3. Try to acquire idelock but …

ideintr function in ide.c

Solve it simple! Then is it working at last?

No.
Sadly, we have one more issue

- So, xv6 disable interrupt during lock handling

```
void
acquire(struct spinlock *lk)
{
  pushcli(); // disable interrupts to avoid deadlock.
  if(holding(lk))
    panic("acquire");

  // The xchg is atomic.
  while(xchg(&lk->locked, 1) != 0)
    ;

  // Record info about lock acquisition for debugging.
  lk->cpu = cpu;
  getcallerpcs(&lk, lk->pcs);
}
```

```
void
release(struct spinlock *lk)
{
  if(!holding(lk))
    panic("release");

  lk->pcs[0] = 0;
  lk->cpu = 0;

// equivalent to lk->locked = 0.
// This code can't use a C assignment
// since it might not be atomic.
  asm volatile("movl $0, %0" : "+m"
(lk->locked) : );

  popcli();
}
```

Compiler and CPU sometimes trick you

- Many compilers and processors, however, execute code out of order to achieve higher performance.

If then, the CPU or the compiler may reorder the instructions for high performance

```
1 while (f == 0);
2 print x;
```

Processor #1

\+

```
4 x = 42;
5 f = 1;
```

Processor #2

=

```
print 42
```

Expected result

A processor or a compiler may notice that instructions [line 4] and [line 5] are not dependent

```
1 while (f == 0);
2 print x;
```

Processor #1

\+

```
4 x = 42;
5 f = 1;
```

→

```
4 f = 1;
5 x = 42;
```

Processor #2

=

```
 print 42
Or
 print ???
```

Incorrect result

Fortunately, We can easily deal with it

- To tell the hardware and compiler not to perform such reorderings, xv6 uses **__sync_synchronize()** in both acquire and release.
- _sync_synchronize() is a memory barrier
  - it tells the compiler and CPU to not reorder loads or stores across the barrier

```
1 while (f == 0);
2 __sync_synchronize();
3 print x;
```

Processor #1

+

```
4 x = 42;
5 __sync_synchronize();
6 f = 1;
```

Processor #2

=

```
print 42
```

Expected result

We finally implement lock!

```c
void
acquire(struct spinlock *lk)
{
  pushcli(); // disable interrupts to avoid deadlock.
  if(holding(lk))
    panic("acquire");

  // The xchg is atomic.
  while(xchg(&lk->locked, 1) != 0)
    ;
  __sync_synchronize();

  // Record info about lock acquisition for debugging.
  lk->cpu = cpu;
  getcallerpcs(&lk, lk->pcs);
}
```

```c
void
release(struct spinlock *lk)
{
  if(!holding(lk))
    panic("release");

  lk->pcs[0] = 0;
  lk->cpu = 0;
  __sync_synchronize();

// equivalent to lk->locked = 0.
// This code can't use a C assignment
// since it might not be atomic.
  asm volatile("movl $0, %0" : "+m" (lk->locked) : );

  popcli();
}
```

Usual example of locking in xv6

```
fork(void)
{
  int i, pid;
  struct proc *np;

  // Allocate process.
  if((np = allocproc()) == 0){
    return -1;
  }
…
  acquire(&ptable.lock);

  np->state = RUNNABLE;

  release(&ptable.lock);

  return pid;
}
```

Unusual example of locking in xv6

```
scheduler(void)
{
  for(;;){
    // Enable interrupts on this processor.
    sti();
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p
      if(p->state != RUNNABLE)
        continue;
      proc = p;
      switchuvm(p);
      p->state = RUNNING;
      swtch(&cpu->scheduler, p->context);
      switchkvm();

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      proc = 0;
    }
    release(&ptable.lock);
  }
}
```

Q : Swtch changes context.
So who release the lock?

A : Next process which occupies CPU

Ptable.lock is released in …

A fork child's very first scheduling

```
void
forkret(void)
{
  static int first = 1;
  // Still holding ptable.lock from scheduler.
  release(&ptable.lock);
...
  // Return to "caller", actually trapret (see allocproc).
}
```

Not first time

```
void
yield(void)
{
  acquire(&ptable.lock);
//DOC: yieldlock
  proc->state = RUNNABLE;
  sched();
  release(&ptable.lock);
}
```

In uniprocessor system ..

```
int
wait(void)
{
...
  acquire(&ptable.lock);
  for(;;){
...
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
...
        kfree(p->kstack);
        p->kstack = 0;
...
        release(&ptable.lock);
        return pid;
      }
    }
...
```

In uniprocessor system ..

```
int
wait(void)
{
...
  acquire(&ptable.lock);
  for(;;){
...
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
...
        kfree(p->kstack);
        p->kstack = 0;
...
        release(&ptable.lock);
        return pid;
      }
    }
...
```

If the number of cpus is 1, no race condition exists?

In uniprocessor system ..

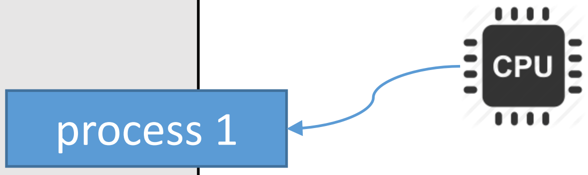
```
int
wait(void)
{
...
  acquire(&ptable.lock);
  for(;;){
...
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
...
        kfree(p->kstack);
        p->kstack = 0;
...
        release(&ptable.lock);
        return pid;
      }
    }
...
```

If the number of cpus is 1, no race condition exists?

NO

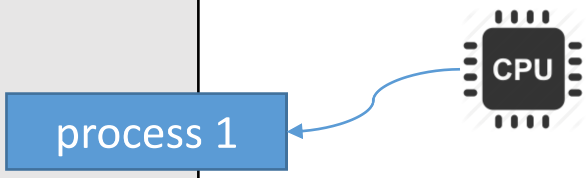
In uniprocessor system ..

```
int
wait(void)
{
...
  acquire(&ptable.lock);
  for(;;){
...
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
...
      kfree(p->kstack);
      p->kstack = 0;
...
      release(&ptable.lock);
      return pid;
    }
  }
...
```



In uniprocessor system ..

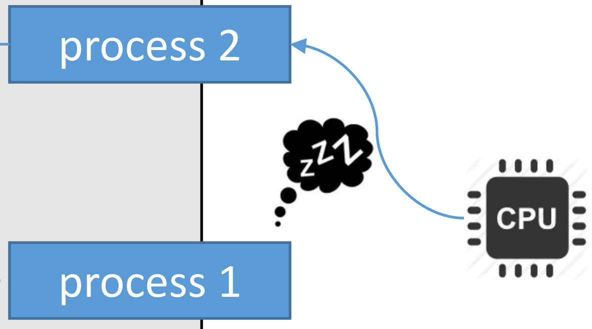
```
int
wait(void)
{
...
  acquire(&ptable.lock);
  for(;;){
...
   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
...
     kfree(p->kstack);
     p->kstack = 0;
...
     release(&ptable.lock);
     return pid;
    }
  }
...
```



Time Interrupt occurs!

In uniprocessor system ..

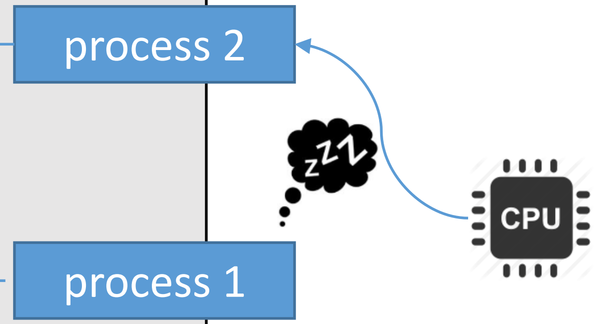
```
int
wait(void)
{
...
  acquire(&ptable.lock);
  for(;;){
...
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
...
      kfree(p->kstack);
      p->kstack = 0;
...
      release(&ptable.lock);
      return pid;
     }
    }
...
```



HYU 한양대학교 HANYANG UNIVERSITY

In uniprocessor system ..

```
274 int
275 wait(void)
276 {
...
281   acquire(&ptable.lock);
282   for(;;){
...
285     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
...
292         kfree(p->kstack);
293         p->kstack = 0;
...
300         release(&ptable.lock);
301         return pid;
302       }
303     }
...
```



Race condition occurs!

In uniprocessor system ..

```
int
wait(void)
{
...
  acquire(&ptable.lock);
  for(;;){
...
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
...
      kfree(p->kstack);
      p->kstack = 0;
...
      release(&ptable.lock);
      return pid;
    }
  }
...
```

If the number of cpus is 1,
a race condition can occur.

It must be protected!

Thank You