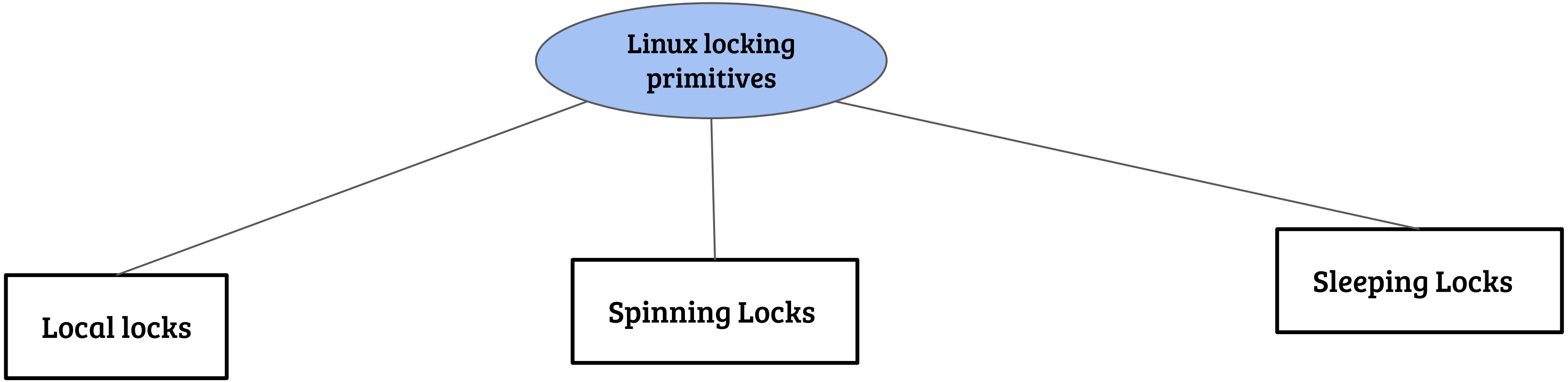
CS614: Linux Kernel Programming

Concurrency, Locks, Semaphores

Debadatta Mishra, CSE, IIT Kanpur

Linux locking overview (non preempt_RT kernel)



Linux locking primitives

Local locks

Wrapper for preemption and interrupt disabling (on local CPU)

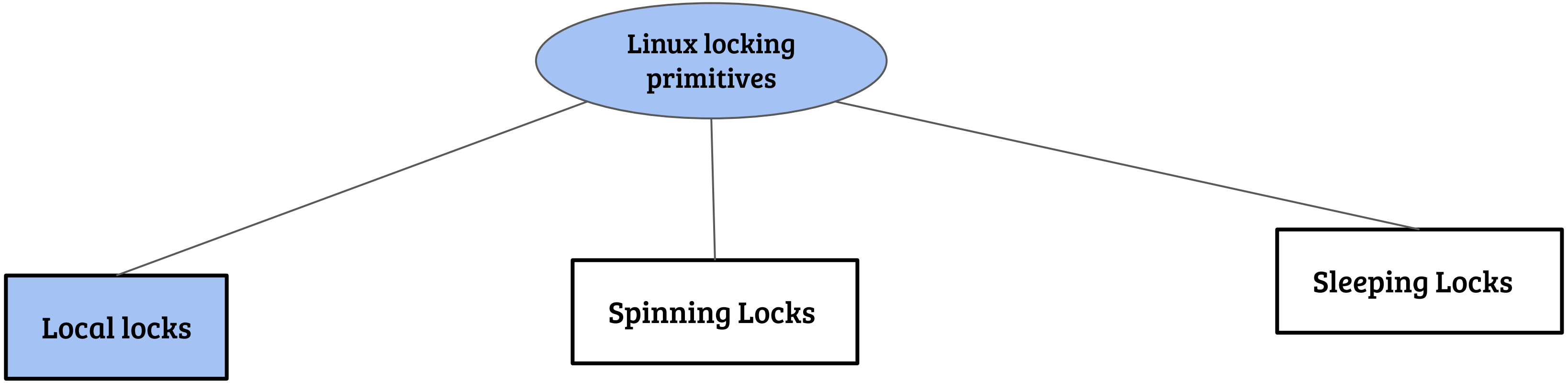
Spinning Locks

Implicitly disable preemption. Variants for further protection (irq, bh)

Sleeping Locks

Scheduling involved, preemption is expected

Linux locking overview: local locks



Linux locking primitives

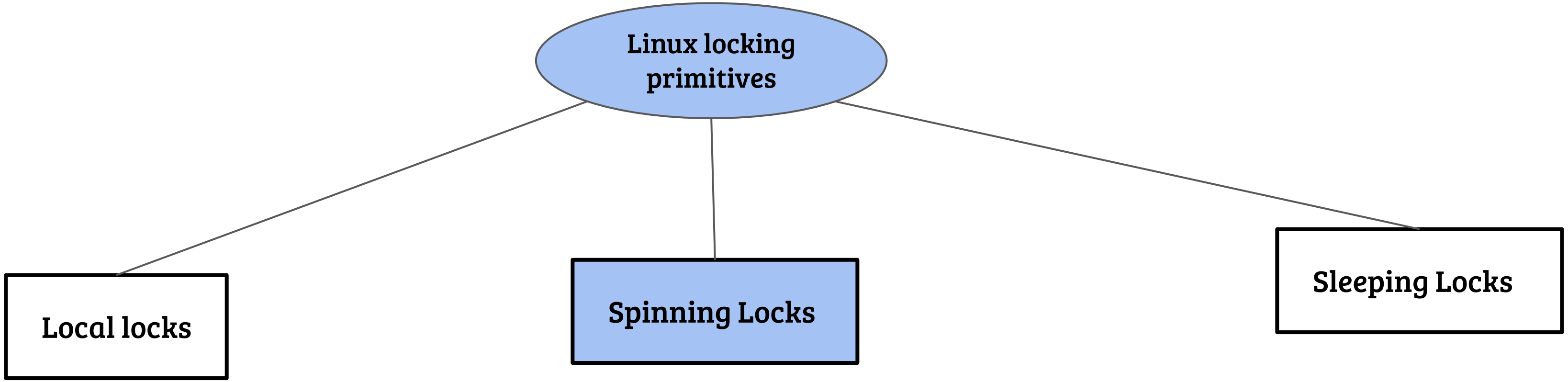
Local locks

Spinning Locks

Sleeping Locks

- Wrapper for preemption and interrupt disabling (on local CPU)
- APIs
  - local_lock(&l) → preempt_disable( )
  - local_unlock(&l) → preempt_enable( )
  - local_lock_irq(&lock) → local_irq_disable( )
  - ...

Linux locking overview: spin locks


Linux locking primitives

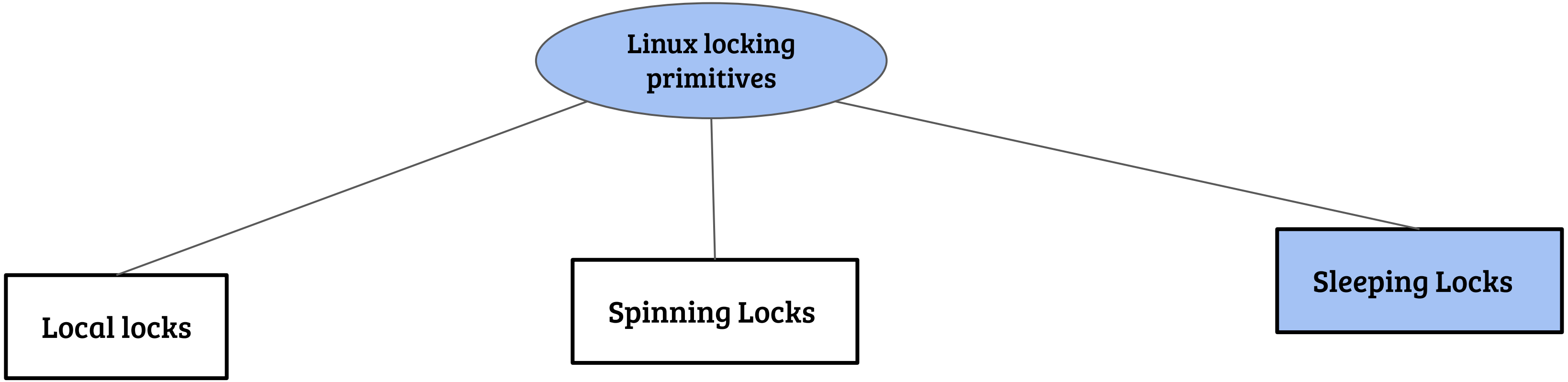
Local locks

Spinning Locks

Sleeping Locks

- Implicitly disable preemption. Variants for further protection (irq, bh)
- Lock examples: spinlock_t, rwlock_t
- APIs
  - spin_lock(&l), spin_unlock(&l), read_lock(&l), write_lock(&l)
  - spin_(un)lock_irq(&l) → Enable (or disable) interrupt and acquire (or release) the lock
  - spin_lock_bh(&l) → Disable softirq and acquire the lock

Linux locking overview: sleeping locks



- Scheduling involved, preemption is expected
- Examples: mutex, semaphore (counting semaphore), rw_semaphore(multiple readers, one writer)
- APIs
  - Mutex: mutex_lock(&l), mutex_unlock(&l)
  - Semaphore: down(&sem), up(&sem), down_timeout(&sem, timeout)
  - R/W Semaphore: down_read(&sem), down_write(&sem), up_read(&l), up_write(&l)

Strategy to handle race conditions in OS

Contexts executing critical sections	Uniprocessor systems	Multiprocessor systems
System calls	Disable preemption	Locking
System calls, Interrupt handler	Disable interrupts	Locking + Interrupt disabling (local CPU)
Multiple interrupt handlers	Disable interrupts	Locking + Interrupt disabling (local CPU)

- Use sleeping locks when there is a chance of "waiting for an event" such as I/O in the critical section

Test and set spinlock: atomic exchange

```
lock_t *L; // Initial value = 0
lock(L)
{
    while(atomic_xchg(*L, 1));
}
unlock(L)
{
    *lock = 0;
}
```

- Atomic exchange: exchange the value of memory and register atomically
- atomic_xchg (int *PTR, int val) returns the value at PTR before exchange
- Ensures mutual exclusion if “val” is stored on a register
- No fairness guarantees

Spinlock using XCHG on X86

```
lock(lock_t *L )
{
    asm volatile(
    “mov $1, %%rax; ”
    “loop: xchg %%rax, (%%rdi); ”
    “cmp $0, %%rax;”
     “jne loop; ”
      : : : “memory” );
}
unlock(int *L ) { *L = 0;}
```

- XCHG *R*, *M* ⇒ Exchange value of register *R* and value at memory address *M*
- *RDI* register contains the lock argument
- Exercise: Visualize a context switch between any two instructions and analyse the correctness

Spinlock using compare and swap

```
lock_t *L; // Initial value = 0
lock(L)
{
    while( CAS(*L, 0, 1) );
}
unlock(L)
{
    *lock = 0;
}
```

- Atomic compare and swap: perform the condition check and swap atomically
- CAS (int **PTR*, int *cmpval*, int *newval*) sets the value of *PTR* to *newval* if *cmpval* is equal to value at *PTR* . Returns 0 on successful exchange
- No fairness guarantees!

CAS on X86: cmpxchg

cmpxchg source[Reg] destination [Mem/Reg]
Implicit registers : rax and flags

```
if rax == [destination]
then
    flags[ZF] = 1
    [destination] = source
else
    flags[ZF] = 0
    rax = [destination]
```

- “cmpxchg” is not atomic in X86, should be used with a “lock” prefix

Spinlock using CMPXCHG on X86

```
lock(lock_t *L )
{
asm volatile(
    “mov $1, %%rcx;”
    “loop: xor %%rax, %%rax;”
   “lock cmpxchg %%rcx, (%%rdi);”
    “jnz loop; ”
    : : : “rcx”,  “rax”, “memory”);
}
unlock(lock_t *L ) { *L = 0;}
```

- Value of RAX (=0) is compared against value at address in register RDI and exchanged with RCX (=1), if they are equal
- Exercise: Visualize a context switch between any two instructions and analyse the correctness

A simple read-write lock

```
struct rw_lock{
                Spinlock R;
                Spinlock G;
                int count;
             };
```

```
#define write_lock(L)      spin_lock(L->G)
#define write_unlock(L)    spin_unlock(L->G)
```

```
read_lock (struct rw_lock *L){
   spin_lock(L->R);
   L->count++;
   If (L->count == 1)
     spin_lock(L->G);
  spin_unlock(L->R);
}
```

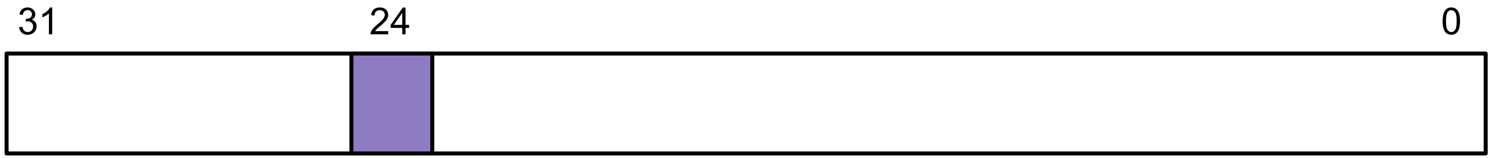
```
read_unlock (struct rw_lock *L){
     spinlock(L->R);
      L->count--;
     if(L->count == 0)
         spin_unlock(L->G);
     spin_unlock(L->R);
   }
```

Improved read-write lock

- Simple R/W lock requires two spinlocks and read accesses are not fully concurrent
- How to improve? Can we get rid of the two locks?

Improved read-write lock

- Simple R/W lock requires two spinlocks and read accesses are not fully concurrent
- How to improve? Can we get rid of the two locks?



- Example R/W lock with 32-bit integer
- 0x1000000 ⟶ Free, 0x0 ⟶ Acquired for write
- [0xFFFFFF, 0x0] ⟶ Readers, {0xFFFFFF⟶ One reader, 0xFFFFFE ⟶ Two readers ... }
- HW: Implement this strategy to design a R/W lock

Fairness in spinlocks

- Spinlock implementations discussed so far are not fair,
    - no bounded waiting
- To ensure fairness, some notion of ordering is required
- What if the threads are granted the lock in the order of their arrival to the lock contention loop?
    - A single lock variable may not be sufficient
    - Example solution: Ticket spinlocks

Atomic fetch and add (xadd on X86)

xadd R, M

```
TmpReg T = R + [M]
R = [M]
[M] = T
```

- Example: M = 100; RAX = 200
- After executing “lock xadd %RAX, M”, value of RAX = 100, M = 300
- Require “lock” prefix to be atomic

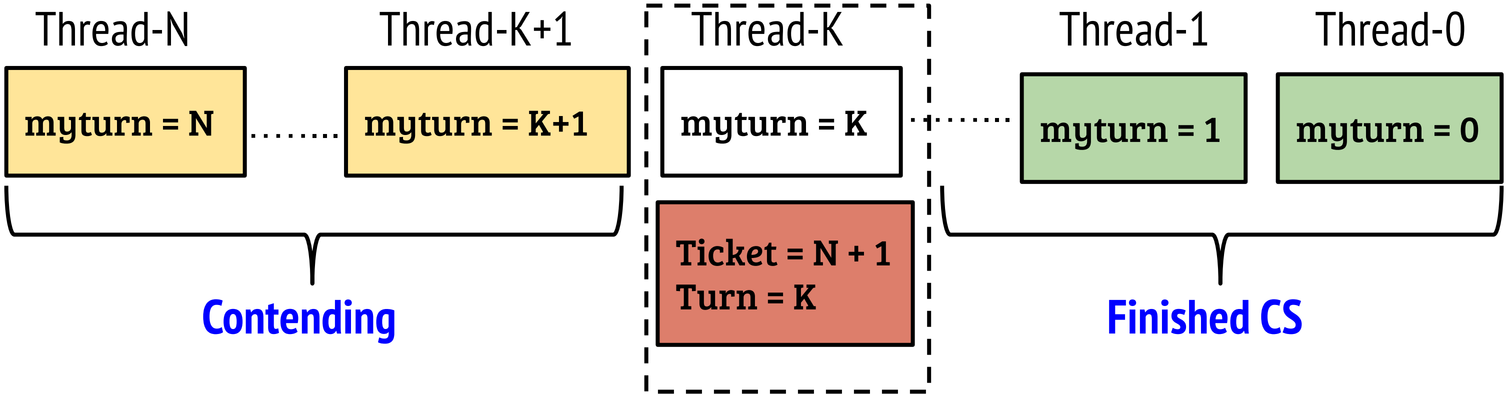
Ticket spinlocks (OSTEP Fig. 28.7)

```
struct lock_t{
        long ticket;
        long turn;
};
void init_lock (struct lock_t *L){
  L → ticket = 0;  L → turn = 0;
}
void unlock(struct lock_t *L){
     L → turn++;
}
```

```
void lock(struct lock_t *L){
  long myturn = xadd(&L → ticket, 1);
  while(myturn != L → turn)
        pause(myturn - L → turn);
}
```

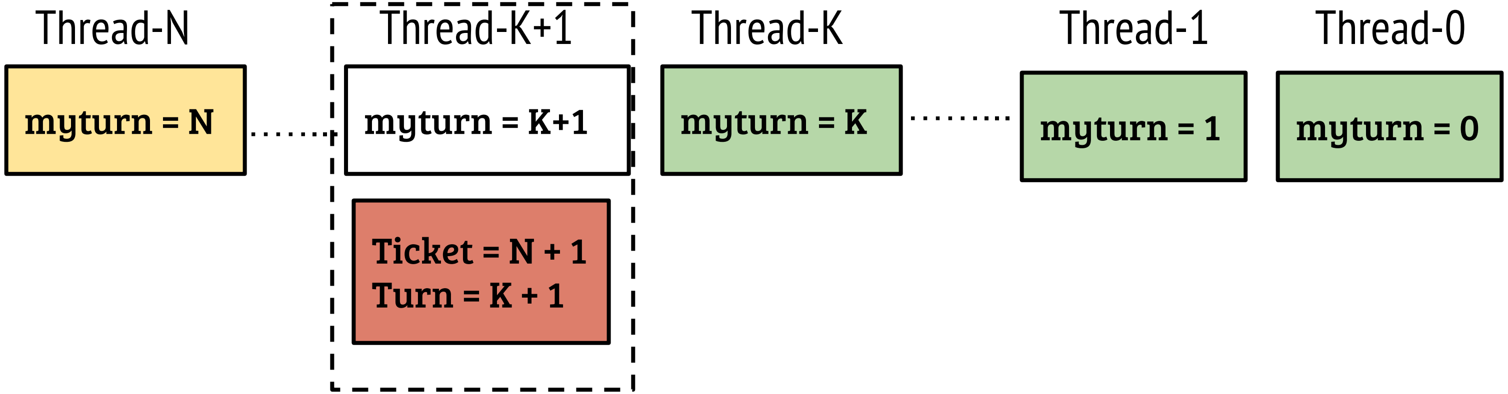
- Example: Order of arrival: T1 T2 T3
- T1 (in CS) : myturn = 0, L = {1, 0}
- T2: myturn = 1, L = {2, 0}
- T3: myturn = 2, L = {3,0}
- T1 unlocks, L = {3, 1}. T2 enters CS

Ticket spinlock



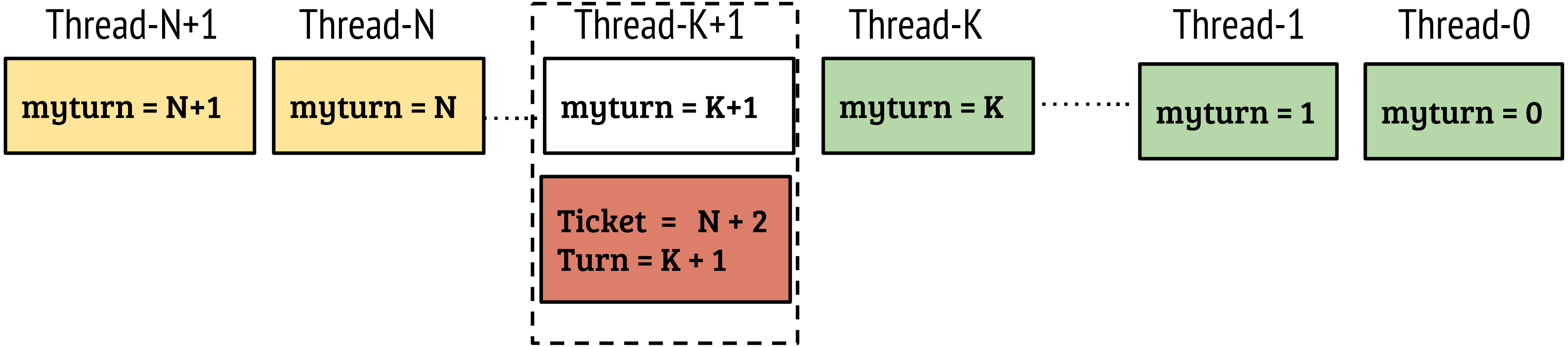
- Local variable “myturn” is equivalent to the order of arrival
- If a thread is in CS ⇒ Local Turn must be same as “Turn”
- Threads waiting = Ticket - Turn -1

Ticket spinlock



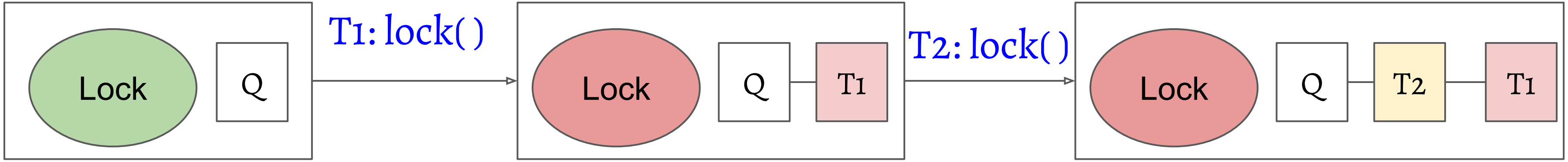
- Value of turn incremented on lock release
- Thread which arrived just after the current thread enters the CS
- When a new thread arrives, it gets the lock after the other threads ahead of the new thread acquire and release the lock

Ticket spinlock



- Ticket spinlock guarantees bounded waiting
- If *N* threads are contending for the lock and execution of the CS consumes *T* cycles, then bound = N * T (assuming negligible context switch overhead)

Queued spinlock (Linux)



- Locks are granted in the order of arrival to the queue
- Lock: check and spin till there are elements ahead in the queue
- Unlock: normal unlock
- Linux kernel implementation of qspinlock merges the queue and lock to a single atomic variable

Semaphores

```
typedef struct semaphore{
                int value;
                spinlock *LOCK;
                Queue *waitQ;
            }sem_t;
```

```
int wait (sem_t *s)
{
   s->value--;
   Wait if s->value < 0
}
```

```
int post (sem_t *s)
{
   s->value++;
   Wakeup one if one or more are waiting
}
```

- Generally, semaphores are initialized to a positive integer K

Semaphore implementation

```
wait (sem_t *s)
{
  lock(s->LOCK);
  s->value--;
  if (s->value < 0){
     insert_tail(s->waitQ, self);
     self->state = WAITING;
     schedule( );
  }
 unlock(s->LOCK);
}
```

```
post (sem_t *s)
{
  lock(s->LOCK);
  s->value++;
  if (s->value <= 0){
     p = remove_head(s->waitQ);
     p->state = READY;
   }
   unlock(s->LOCK);
}
```

- Is the implementation correct?

Semaphore implementation

```
wait (sem_t *s)
{
  lock(s->LOCK);
  s->value--;
  if (s->value < 0){
    insert_tail(s->waitQ, self);
    self->state = WAITING;
    schedule( );
  }
 unlock(s->LOCK);
}
```

```
post (sem_t *s)
{
  lock(s->LOCK);
  s->value++;
  if (s->value <= 0){
    p = remove_head(s->waitQ);
    p->state = READY;
   }
   unlock(s->LOCK);
}
```

- Is the implementation correct? Process can be descheduled while holding lock

Semaphore implementation

```
wait (sem_t *s)
{
  lock(s->LOCK);
  s->value--;
  if (s->value < 0){
    insert_tail(s->waitQ, self);
    self->state = WAITING;
    unlock(s->LOCK);
    schedule( );
    return;
  }
  unlock(s->LOCK);
}
```

```
post (sem_t *s)
{
  lock(s->LOCK);
  s->value++;
  if (s->value <= 0){
    p = remove_head(s->waitQ);
    p->state = READY;
  }
  unlock(s->LOCK);
}
```

- Homework: “wait” is correct under an assumption, can you find it?

Allowing concurrent access

- The locking scheme discussed so far can not allow concurrent read and write access to a shared memory object
- A restricted scenario: Allowing one writer (updater) and many readers
- Solution: Sequential locks and Read-Copy-Update (RCU)

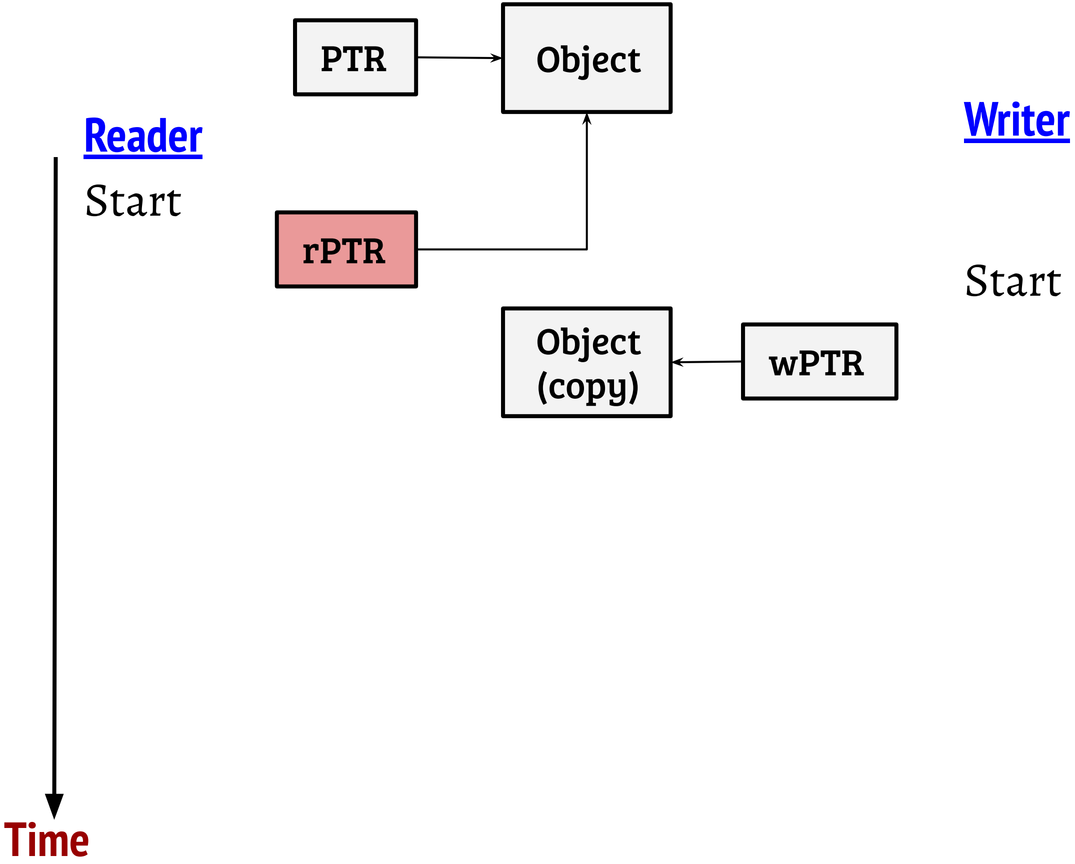
Allowing concurrent access

- The locking scheme discussed so far can not allow concurrent read and write access to a shared memory object
- A restricted scenario: Allowing one writer (updater) and many readers
- Solution: Sequential locks and Read-Copy-Update (RCU)
- Idea
    - Sequential locks consists of a spinlock and a counter
    - Writers acquire spinlock and increments the counter before entering CS
    - Writers increment counter before releasing the spinlock
    - Readers gets the value of counter before entering into CS, perform read and check the value of counter to detect "writer interference"
    - Example: sock_write_timestamp

Allowing concurrent access

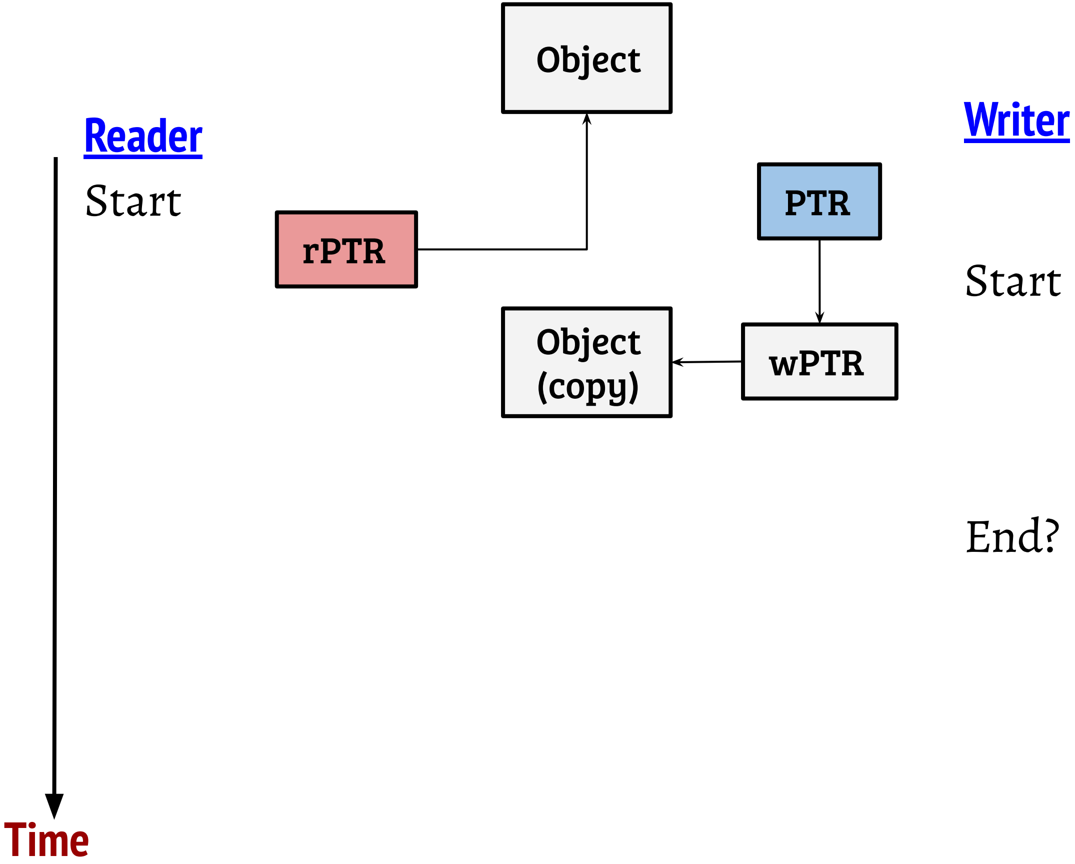
- The locking scheme discussed so far can not allow concurrent read and write access to a shared memory object
- A restricted scenario: Allowing one writer (updater) and many readers
- Solution: Sequential locks and Read-Copy-Update (RCU)
- Idea:
    - Readers access a shared object using a PTR without taking any locks
    - Updater works with a separate copy of the object concurrently
    - Atomically update the PTR to point to the new object

Read-Copy-Update (Example)



- Reader has a reference to the shared object
- Writer performs copy of the object pointed to from a local pointer and updates its content

Read-Copy-Update (Example)



Reader

Start

Object

rPTR

PTR

Object (copy)

wPTR

Writer

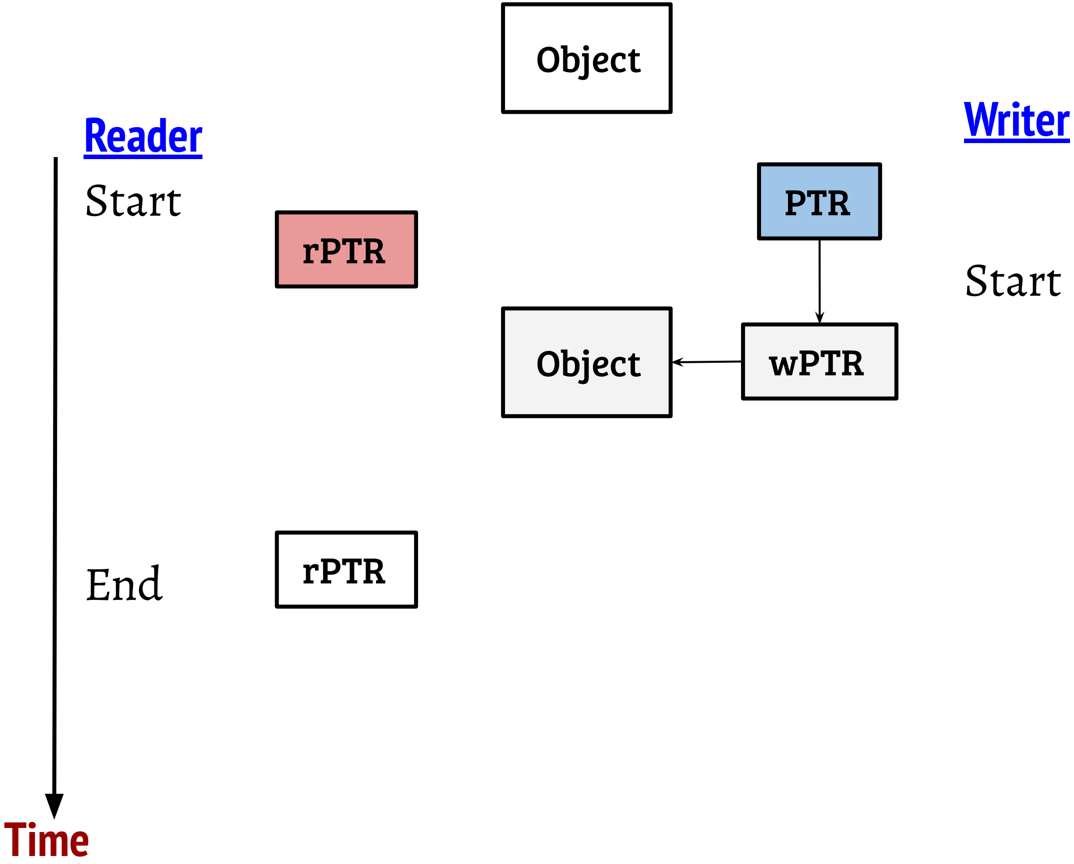
Start

End?

Time

- Reader has a reference to the shared object
- Writer performs copy of the object pointed to from a local pointer and updates its content
- The global PTR is *atomically* updated to point to the updated object, Done?

Read-Copy-Update (Example)



Reader

Start

rPTR

End

rPTR

Time

Object

PTR

Object

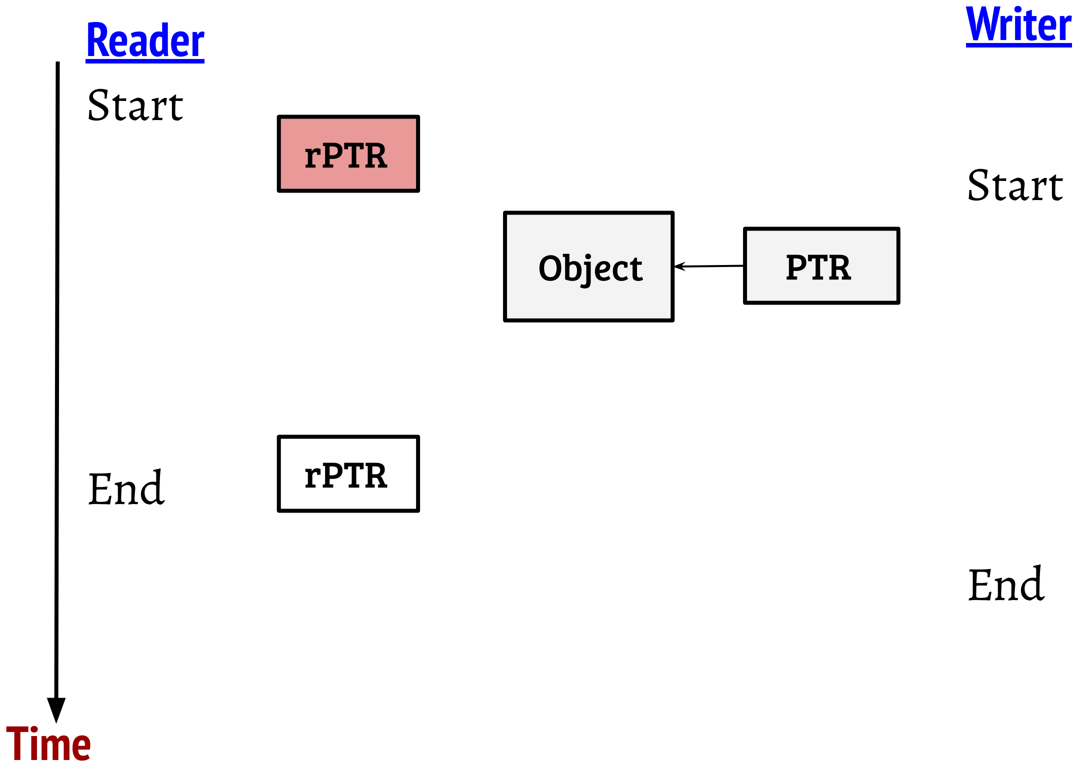
wPTR

Writer

Start

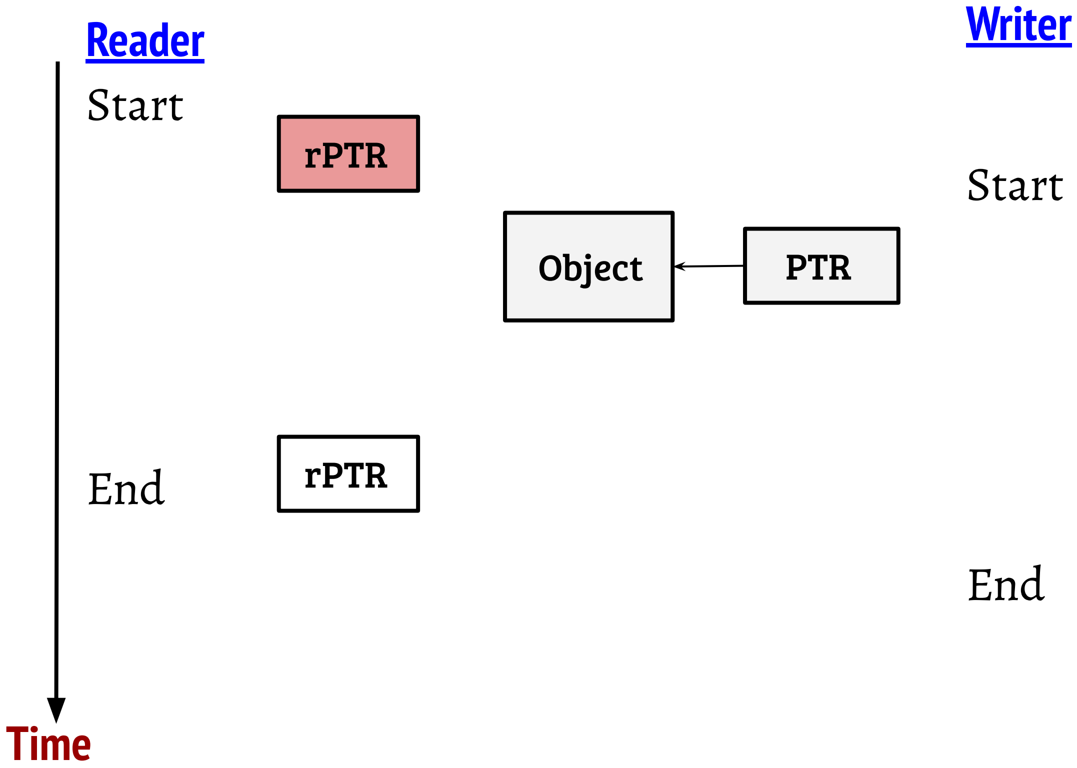
- Reader has a reference to the shared object
- Writer performs copy of the object pointed to from a local pointer and updates its content
- The global PTR is *atomically* updated to point to the updated object
- Need to cleanup (collect) the old copy

Read-Copy-Update (Example)



- Reader has a reference to the shared object
- Writer performs copy of the object pointed to from a local pointer and updates its content
- The global PTR is *atomically* updated to point to the updated object
- Need to cleanup (collect) the old copy

Read-Copy-Update (Example)



- Reader has a reference to the shared object
- Writer performs copy of the object pointed to from a local pointer and updates its content
- The global PTR is *atomically* updated to point to the updated object
- Need to cleanup (collect) the old copy. Challenges
  - know when no readers are using the old copy
  - How long to wait?

Read-Copy-Update: Subtle issues

- Reader need to notify the “start” and “end” of its usage
  - If the reader is after PTR update but before reclaim, should it use new or old?
- The old copy can not be freed before the reference count to the old copy is zero
  - How long an updater wait? Can we defer the reclaim?
  - How to design a time bound reclamation?

Read-Copy-Update: Subtle issues

- Reader need to notify the “start” and “end” of its usage
  - If the reader is after PTR update but before reclaim, should it use new or old?
  - No problems if the new readers are allowed to use the new copy
- The old copy can not be freed before the reference count to the old copy is zero
  - How long an updater wait? Can we defer the reclaim?
  - If the updater does not want to wait, it can defer this task to future
  - How to design a time bound reclamation?
  - If readers are not preempted during usage, different events can be used to infer no reference to the object