CS614: Linux Kernel Programming

Concurrency, Locks, Semaphores

Debadatta Mishra, CSE, IIT Kanpur

Shared Address Space in User Space (Threads)

- Threads share the address space
  - Low context switch overheads
  - Global variables can be accessed from thread functions
  - Dynamically allocated memory can be passed as thread arguments
- Sharing data is convenient to design parallel computation
- Pthread API for multi-threaded programming

Threads sharing the address space

- Threads share the address space
  - Global variables can be accessed from thread functions
- Everything seems to be fine, what is the issue?
- How does OS fit into this discussion?
  - Data parallel processing: Data is partitioned into disjoint sets and assigned to different threads
  - Task parallel processing: Each thread performs a different computation on the same data

Sharing can be problematic!

```
static int counter = 0;
void *thfunc(void *)
{
   int ctr = 0;
   for(ctr=0; ctr<100000; ++ctr)
         counter++;
}
```

- If this function is executed by two threads, what will be the value of *counter* when two threads complete?

Sharing can be problematic!

```
static int counter = 0;
void *thfunc(void *)
{
    int ctr = 0;
    for(ctr=0; ctr<100000; ++ctr)
            counter++;
}
```

- If this function is executed by two threads, what will be the value of *counter* when two threads complete?
- Non-deterministic output
- Why?

Sharing can be problematic!

```
static int counter = 0;
void *thfunc(void *)
{
   int ctr = 0;
   for(ctr=0; ctr<100000; ++ctr)
         counter++;
}
```

counter++ in assembly

```
mov (counter), R1
Add 1, R1
Mov R1, (counter)
```

Even on a single processor system, scheduling of threads between the above instructions can be problematic!

Sharing can be problematic!

T1: mov (counter), R1   // R1 = 0

T1: Add 1, R1

{switch-out, R1=1 saved in PCB}

- Assume that T1 is executing the first iteration
- On context switch, value of R1 is saved onto the PCB
- Thread T2 is scheduled and starts executing the loop

Sharing can be problematic!

T1: mov (counter), R1 // R1 = 0
T1: Add 1, R1
{switch-out, R1=1 saved in PCB}
T2: mov (counter), R1 // R1 = 0
T2: Add 1, R1 // R1 = 1
T2 mov R1, (counter) // counter = 1
{switch-out, T1 scheduled, R1 = 1}

- T2 executes all the instructions for one iteration of the loop, saves 1 to counter (in memory) and then, scheduled out
- T1 is switched-in, R1 value (=1) loaded from the PCB

Sharing can be problematic!

T1: mov (counter), R1 // R1 = 0
T1: Add 1, R1
{switch-out, R1=1 saved in PCB}
T2: mov (counter), R1 // R1 = 0
T2: Add 1, R1 // R1 = 1
T2 mov R1, (counter) // counter = 1
{switch-out, T1 scheduled, R1 = 1}
T1: mov R1, (counter) // counter = 1!

- T1 stores one into counter
- Value of counter should have been two
- What if "counter++" is compiled into a single instruction, e.g., "inc (counter)" ?
-

Sharing can be problematic!

```
T1: mov (counter), R1     // R1 = 0
T1: Add 1, R1
{switch-out, R1=1 saved in PCB}
T2: mov (counter), R1     // R1 = 0
T2: Add 1, R1             // R1 = 1
T2 mov R1, (counter)      // counter = 1
{switch-out, T1 scheduled, R1 = 1}
T1: mov R1, (counter)     // counter = 1!
```

- T1 stores one into counter
- Value of counter should have been two
- What if "counter++" is compiled into a single instruction, e.g., "inc (counter)" ?
- Does not solve the issue on multi-processor systems!

Sharing can be problematic!

```
static int counter = 0;
void *thfunc(void *)
{
    int ctr = 0;
    for(ctr=0; ctr<100000; ++ctr)
            counter++;
}
```

- If this function is executed by two threads, what will be the value of *counter* when two threads complete?
- Non-deterministic output
- Why?
- Accessing shared variable in a concurrent manner results in incorrect output

Definitions

- Atomic operation: An operation is atomic if it is *uninterruptible* and *indivisible*
- Critical section: A section of code accessing one or more shared resource(s), mostly shared memory location(s)
- Mutual exclusion: Technique to allow exactly one execution entity to execute the critical section
- Lock: A mechanism used to orchestrate entry into critical section
- Race condition: Occurs when multiple threads are allowed to enter the critical section

Threads sharing the address space

- Threads share the address space
  - Global variables can be accessed from thread functions

- Everything seems to be fine, what is the issue?
- Correctness of program impacted because of concurrent access to the shared data causes race condition
- How does OS fit into this discussion?

  assigned to different threads
  - Task parallel processing: Each thread performs a different computation on the same data

Critical sections in OS

- OS maintains shared information which can be accessed from different OS mode execution (e.g., system call handlers, interrupt handlers etc.)
- Example (1): Same page table entry being updated concurrently because of swapping (triggered because of low memory) and change of protection flags (because of mprotect( ) system call)
- Example (2): The queue of network packets being updated concurrently to deliver the packets to a process and receive incoming packets from the network device

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)

Threads sharing the address space

- Threads share the address space
- Everything seems to be fine, what is the issue?
- Correctness of program impacted because of concurrent access to the shared data causes race condition
- How does OS fit into this discussion?
- Concurrency issues in OS is challenging as finding the race condition itself is non-trivial

on the same data

Design issues of locks

```
pthread_mutex_t lock;    // Initialized using pthread_mutex_init
static int counter = 0;
```

- Efficiency of lock and unlock operations
- Lock acquisition delay vs. wasted CPU cycles
- Fairness of the locking scheme

```
    pthread_mutex_lock(&lock);      // One thread acquires lock, others wait
    counter++;                      // Critical section
    pthread_mutex_unlock(&lock);    // Release the lock
  }
}
```

Lock ADT

```
lock_t *L;

lock(L)
{
  // Return ⇒ Lock acquired
}
unlock(L)
{
  // Return ⇒ Lock released
}
```

```
lock_t *L1, L2;
....
lock(L1)
Critical Section
unlock(L1)
....
lock(L2)
Critical Section
unlock(L2)
....
Lock(L1)
Critical Section
unlock(L2)
```

Lock ADT: Efficiency

```
lock_t *L;

lock(L)
{
  // Return ⇒ Lock acquired
}
unlock(L)
{
  // Return ⇒ Lock released
}
```

- Efficiency of lock/unlock operations directly influence performance
- Implementation choices?

Lock ADT: Efficiency

```
lock_t *L;

lock(L)
{
  // Return ⇒ Lock acquired
}
unlock(L)
{
  // Return ⇒ Lock released
}
```

- Efficiency of lock/unlock operations directly influence performance
- Implementation choices?
- Hardware assisted implementations
  - Use hardware synchronization primitives like atomic operations

Lock ADT: Efficiency

```
lock_t *L;

lock(L)
{
  // Return ⇒ Lock acquired
}
unlock(L)
{
  // Return ⇒ Lock released
}
```

- Efficiency of lock/unlock operations directly influence performance
- Implementation choices?
- Hardware assisted implementations
  - Use hardware synchronization primitives like atomic operations
- Software locks are implemented without assuming any hardware support
  - Not used in practice because of high overheads

Design issues of locks

```
pthread_mutex_t lock;     // Initialized using pthread_mutex_init
static int counter = 0;
```

- Efficiency of lock and unlock operations
- Hardware-assisted lock implementations are used for efficiency
- Lock acquisition delay vs. wasted CPU cycles
- Fairness of the locking scheme

```
    counter++;                          // Critical section
    pthread_mutex_unlock(&lock);  // Release the lock
  }
}
```

Lock: busy-wait (spinlock) vs. Waiting

T1	T2
lock(L) //Acquired	
Critical section	lock(L) //Lock is busy. Reschedule or Spin?
unlock(L)	Critical section
	unlock(L)

Lock: busy-wait (spinlock) vs. Waiting

T1	**T2**
lock(L) //Acquired	
Critical section	lock(L) //Lock is busy. Reschedule or Spin?
unlock(L)	Critical section
	unlock(L)

- With busy waiting, context switch overheads saved, wasted CPU cycles due to spinning
- Busy waiting is prefered when critical section is small and the context executing the critical section is not rescheduled (e.g., due to I/O wait)

Design issues of locks

```
pthread_mutex_t lock;    // Initialized using pthread_mutex_init
static int counter = 0;
```

- Efficiency of lock and unlock operations
- Hardware-assisted lock implementations are used for efficiency
- Lock acquisition delay vs. wasted CPU cycles
- Use waiting locks and spinlocks depending on the requirement
- Fairness of the locking scheme

```
  }
}
```

Fairness

- Given *N* threads contending for the lock, number of unsuccessful attempts for lock acquisition for all contending threads should be same

Fairness

- Given *N* threads contending for the lock, number of unsuccessful attempts for lock acquisition for all contending threads should be same
- Bounded wait property
  - Given *N* threads contending for the lock, there should be an upper bound on the number of attempts made by a given context to acquire the lock

Design issues of locks

```
pthread_mutex_t lock;     // Initialized using pthread_mutex_init
```

- Efficiency of lock and unlock operations
- Hardware-assisted lock implementations are used for efficiency
- Lock acquisition delay vs. wasted CPU cycles
- Use waiting locks and spinlocks depending on the requirement
- Fairness of the locking scheme
- Contending threads should not starve for the lock indefinitely

```
    pthread_mutex_unlock(&lock); // Release the lock
  }
}
```

Spinlock: Buggy attempt

```
1.  lock_t *L; // Initial value = 0
2.  lock(L)
3.  {
4.     while(*L);
5.     *L = 1;
6.  }
7.  unlock(L)
8.  {
9.     *L = 0;
10. }
```

- Does this implementation work?

Spinlock: Buggy attempt

```
1.  lock_t *L; // Initial value = 0
2.  lock(L)
3.  {
4.      while(*L);
5.      *L = 1;
6.  }
7.  unlock(L)
8.  {
9.      *L = 0;
10. }
```

- Does this implementation work?
- No, it does not ensure *mutual exclusion*
- Why?

Spinlock: Buggy attempt

```
1.  lock_t *L; // Initial value = 0
2.  lock(L)
3.  {
4.      while(*L);
5.      *L = 1;
6.  }
7.  unlock(L)
8.  {
9.      *L = 0;
10. }
```

- Does this implementation work?
- No, it does not ensure *mutual exclusion*
- Why?
  - Single core: Context switch between line #4 and line #5
  - Multicore: Two cores exiting the while loop by reading lock = 0

Spinlock: Buggy attempt

```
1.  lock_t *L; // Initial value = 0
2.  lock(L)
3.  {
4.     while(*L);
5.     *L = 1;
6.  }
7.  unlock(L)
8.  {
9.     *L = 0;
10. }
```

- Does this implementation work?
- No, it does not ensure *mutual exclusion*
- Why?
  - Single core: Context switch between line #4 and line #5
  - Multicore: Two cores exiting the while loop by reading lock = 0
- Core issue: Compare and swap has to happen atomically!

Spinlock using atomic exchange

```
1. lock_t *L; // Initial value = 0
2. lock(L)
3. {
4.    while(atomic_xchg(*L, 1));
5. }
6. unlock(L)
7. {
8.    *lock = 0;
9. }
```

- 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

```
1. lock_t *L; // Initial value = 0
2. lock(L)
3. {
4.     while( CAS(*L, 0, 1) );
5. }
6. unlock(L)
7. {
8.     *lock = 0;
9. }
```

- 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

1. if rax == [destination]
2. then
3. flags[ZF] = 1
4. [destination] = source
5. else
6. flags[ZF] = 0
7. 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

Read-write locks

- Spinlock does not distinguish between read and write access to shared variables/data structures
- Many real life scenarios exhibit that behavior
- Example 1: Search and insert on a list
- Example 2: Search a file block in disk cache with concurrent insertions
- Allow multiple readers when no write is going on, how?

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);
}
```

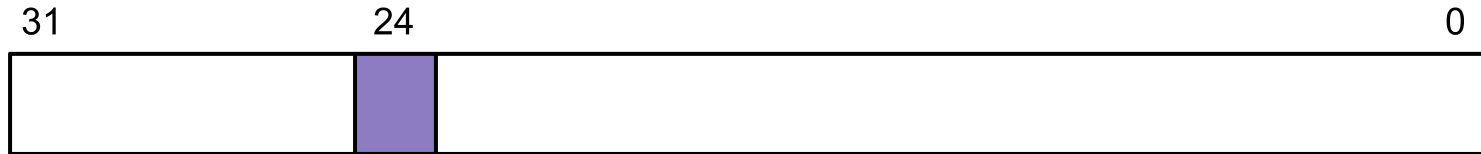
```
read_unlock (struct rw_lock *L){
    spinlock(L->R);
    L->count--;
    if(L->count == 0)
        spin_unlock(L->G);
    spin_unlock(L->R);
}
```

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

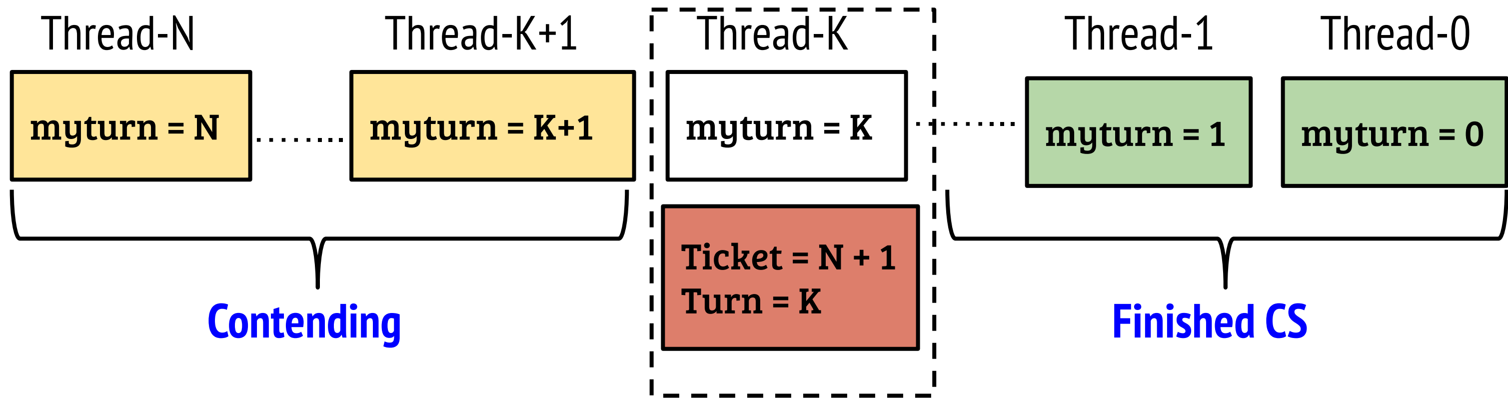
Ticket spinlocks (OSTEP Fig. 28.7)

```
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);
}
```

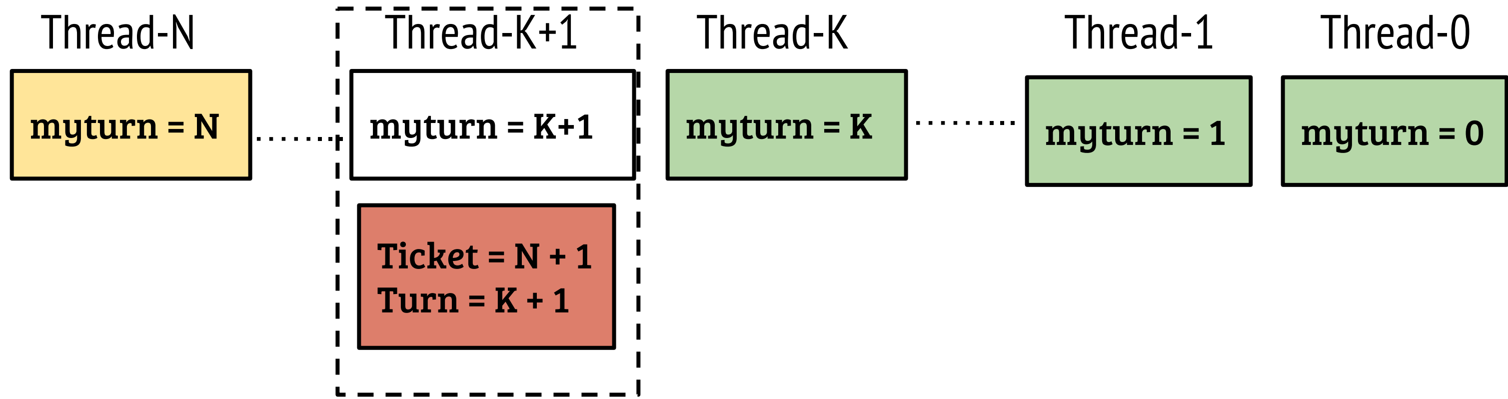
- Example: Order of arrival: T1 T2 T3
- T1 (in CS) : myturn = 0, L = {1, 0}
- T2: myturn = 1, L = {2, 0}
- T3: myturn = 2, L = {3,0}
- T1 unlocks, L = {3, 1}. T2 enters CS

Ticket spinlock



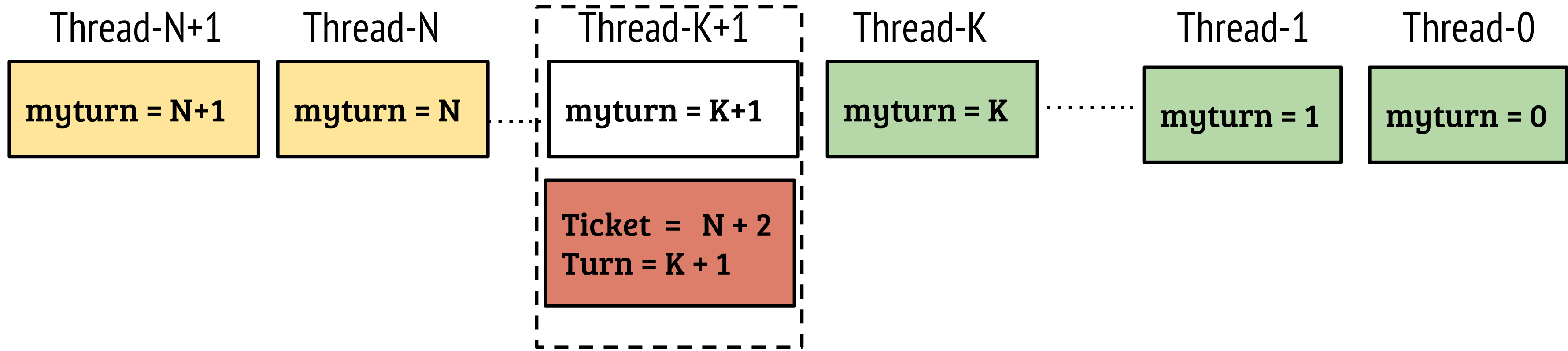
- Local variable "myturn" is equivalent to the order of arrival
- If a thread is in CS ⇒ Local Turn must be same as "Turn"
- Threads waiting = Ticket - Turn -1

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)

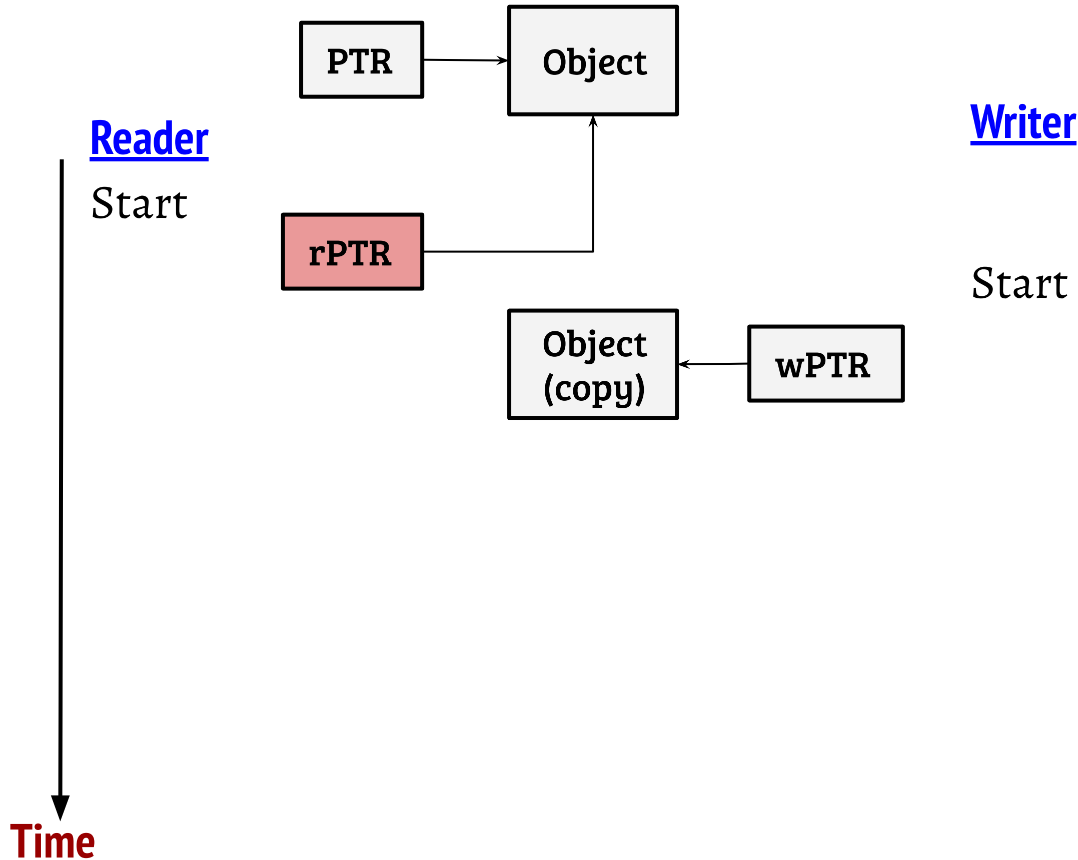
Allowing concurrent access

- 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: 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: 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)

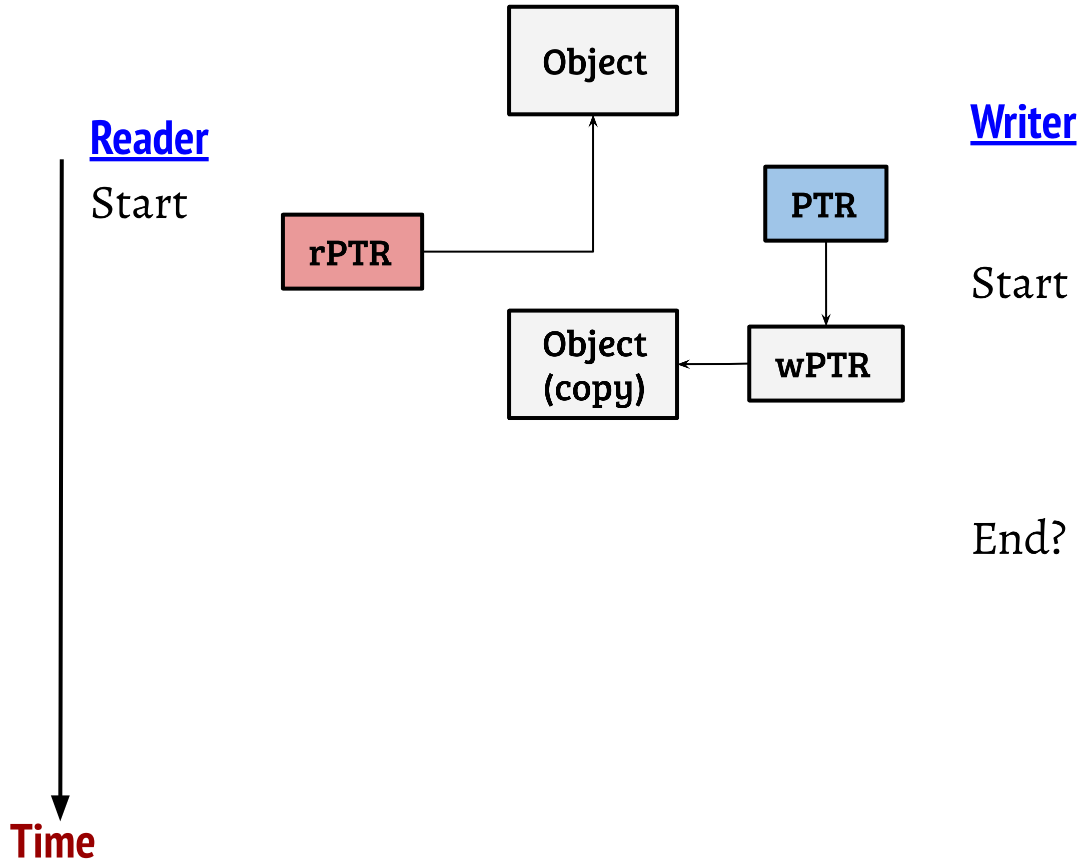


Writer

Start

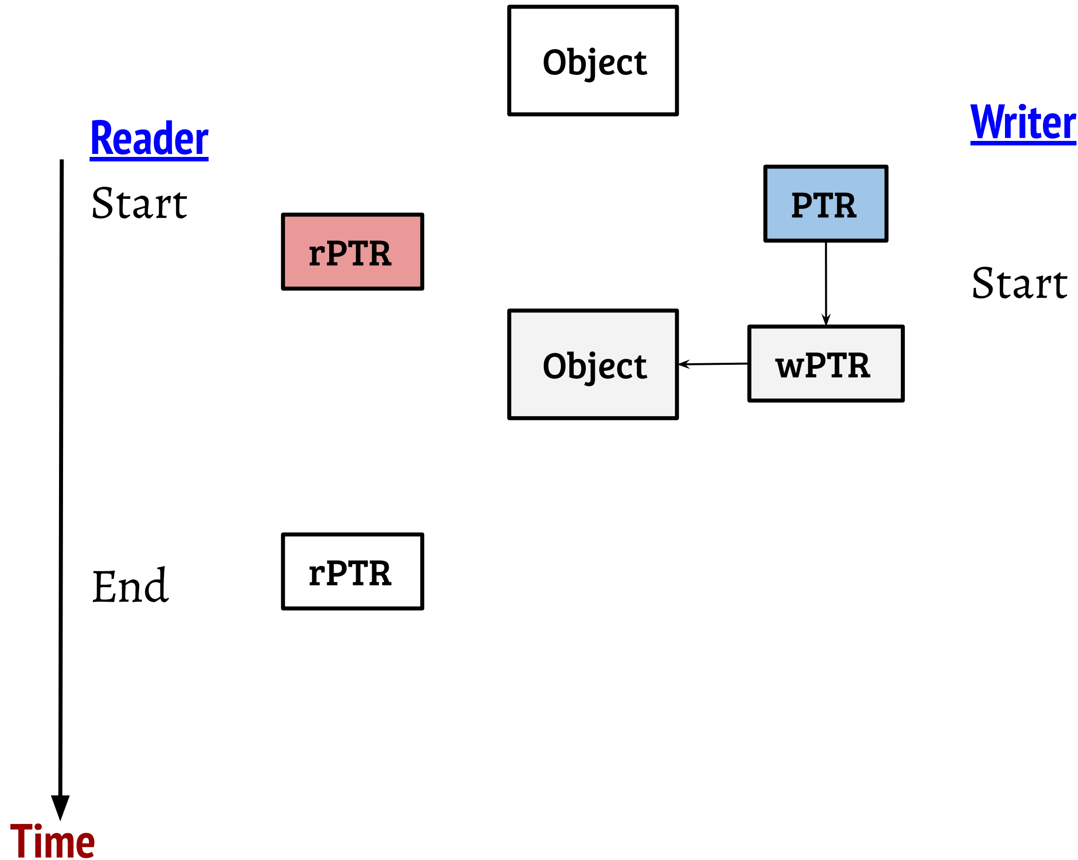
- Reader has a reference to the shared object
- Writer performs copy of the object pointed to from a local pointer and updates its content

Read-Copy-Update (Example)



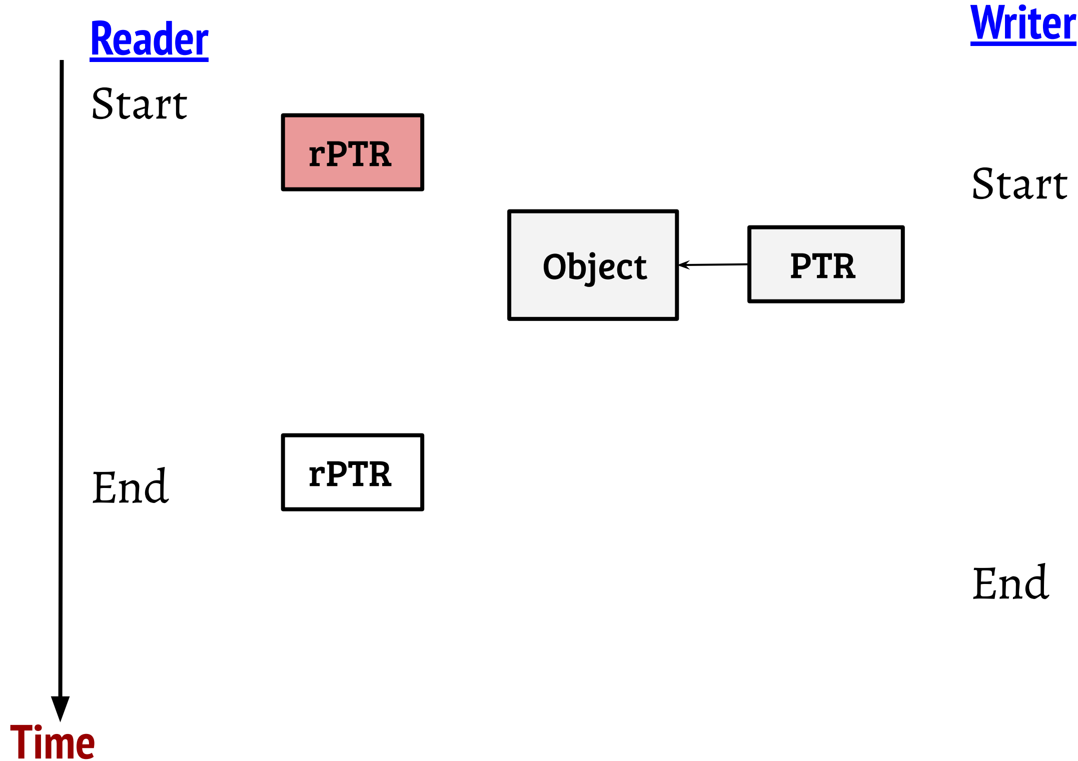
- Reader has a reference to the shared object
- Writer performs copy of the object pointed to from a local pointer and updates its content
- The global PTR is *atomically* updated to point to the updated object, Done?

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

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

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
- Two operations: wait and post (other notations {wait, signal}, {P,V}, {down, up})

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?