

ECE326

PROGRAMMING LANGUAGES

Lecture 34 : Concurrent Programming

Kuei (Jack) Sun

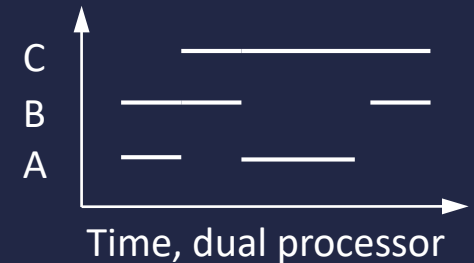
ECE

University of Toronto

Fall 2019

Concurrent Programming

- Multiple tasks execute simultaneously
- Thread
 - Independent sequence of execution
 - Has its own stack, but shares the heap with other threads
- Parallel computing
 - Threads executing at same physical time instant
 - Only possible if each thread runs on its own processor
- Concurrency
 - Threads may *interleave* on the same processor



Purpose

- Speed up program
 - Usually with more people, a job can get done faster
- Criteria for speed up
 - Threads can work relatively independently
 - Seldom need to wait for other threads
 - E.g. to access shared data
 - E.g. to wait for input produced by another thread
 - Threads are often waiting for IO (e.g. read from disk, network)
 - Only important if threads are sharing a processor
 - While one thread waits, other threads can still do work on processor

Concurrent Programming

- Most programming languages provide *library support*
 - Creating and managing threads done through function calls

```
use std::thread;
use std::time::Duration;

thread::spawn(|| {
    for i in 1..10 {
        println!("hi number {} from thread!", i);
        thread::sleep(Duration::from_millis(1));
    }
});
```

spawn can take a
closure as argument

- Go has language support for creating threads

```
go f(x, y, z); // starts a new thread (aka goroutine)
```

Basics

- A program always starts with one thread: main
- main creates new threads, and those can create more
- Creator *should* wait for the threads it created to end

```
fn main() {  
    let handle = thread::spawn(|| {  
        for i in 1..10 {  
            println!("hi number {} from thread!", i);  
        }  
    });  
  
    for i in 1..5 {  
        println!("hi number {} from main!", i);  
    }  
  
    handle.join().unwrap();    // wait for created thread to finish  
}
```

Ownership

- A thread can potentially live longer than its creator
 - E.g. the creator chooses not to call join before exiting
- Problem arises if closure references outer variable
 - Therefore, all outer variables must be “moved” into closure

```
fn main() {  
    let v = vec![1, 2, 3];  
    let handle = thread::spawn(move || {  
        println!("Here's a vector: {:?}", v);  
    });  
    // you may no longer use 'v' here.  
    // main may exit before thread does!  
}
```

By default, closures
borrow outer variables

Must specify move to
make closure move
outer variables into it.

Challenge

- Sharing data
 - Ownership
 - Threads need to jointly own an object
 - Updates to same data can result in *race condition*
 - Caused by problematic interleaving of threads
 - Depending on timing of thread execution, which is difficult to control
 - Race condition can lead to unexpected and often incorrect results
- Synchronization
 - Threads may need to communicate with each others
 - One thread may need to wait for another thread to advance

Reference Counting

- A commonly used technique to share an object
- Analogy
 - First person to walk into living room turns on TV
 - Subsequent people entering can sit down immediately
 - Last person to leave will turn off the TV
- Reference Counting
 - Creator of object sets reference count to 1
 - Others will increment count before use
 - Everyone decrements count after use
 - If count is 0, free the object

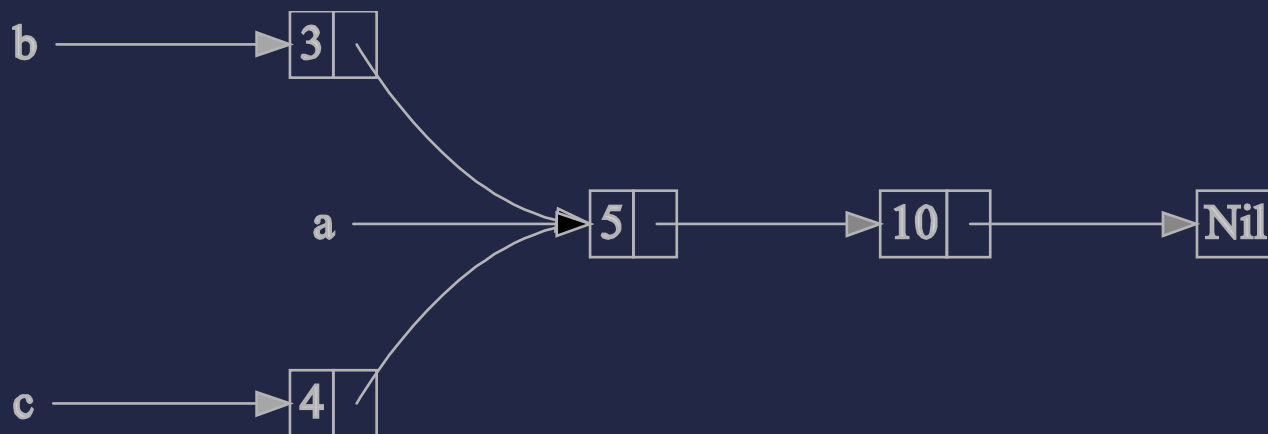
Smart Pointer

- A wrapper class over a pointer, and acts like a pointer
- C++ Example
 - `unique_ptr`
 - Automatically frees pointed-to object when it goes out of scope
 - `shared_ptr`
 - A reference counting smart pointer
 - Allows multiple threads to share pointed-to object
 - Last reference holder will delete the object
 - May not be the original creator of the object

Rc<T>

- Allows sharing data in single-threaded setting

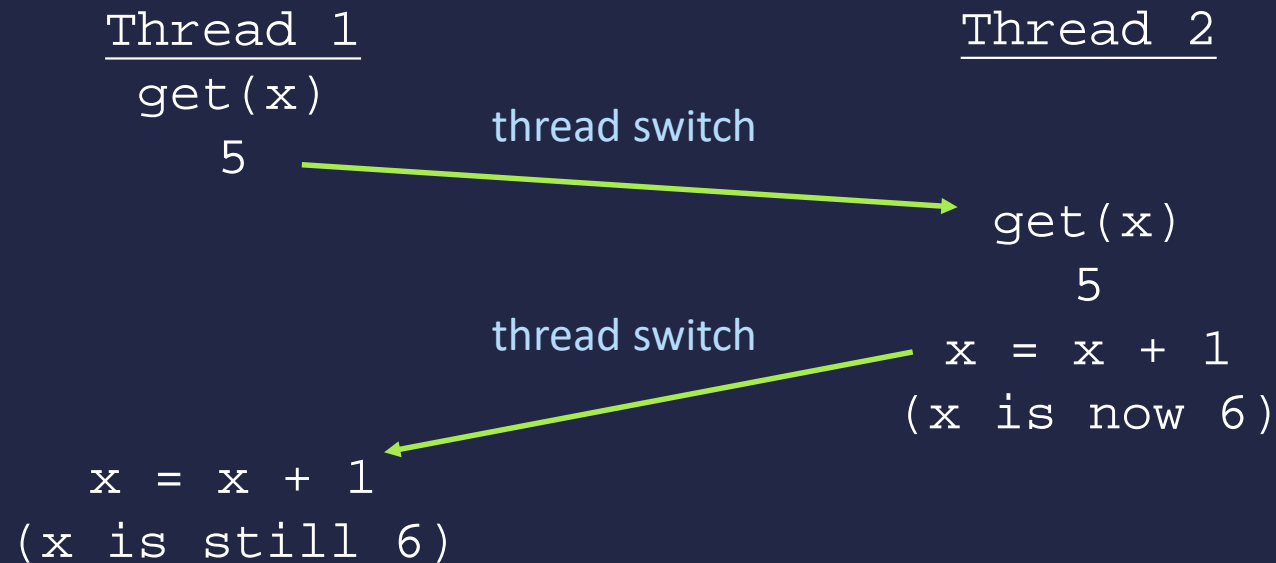
```
enum List { Cons(i32, Rc<List>), Nil, }  
use crate::List::{Cons, Nil};  
use std::rc::Rc;  
  
fn main() {  
    let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));  
    let b = Cons(3, Rc::clone(&a));  
    let c = Cons(4, Rc::clone(&a));  
}
```



Each time `Rc::clone` is called, reference count of `a` increases by 1.

Lost Update

- One potential problem caused by race condition
 - Assume both threads are running on same processor



- Solution: atomic instructions

Arc<T>

- Allows sharing data across different threads
 - A in Arc stands for *atomic*
- Atomic instruction
 - A single, uninterruptible instruction on processor
 - Can complete without interference from other threads
 - Generally not used because it is more expensive (time-wise)
 - E.g. fetch-and-add

```
function FetchAndAdd(address location, int inc) {  
    int value := *location  
    *location := value + inc  
    return value  
}
```

Arc<T>

- Arc<T> uses atomic instructions to update counter
 - Unlike regular Rc<T>, which is not *thread safe*
- Thread safety
 - Function that behaves correctly during simultaneous execution by multiple threads
 - E.g. freedom from race condition

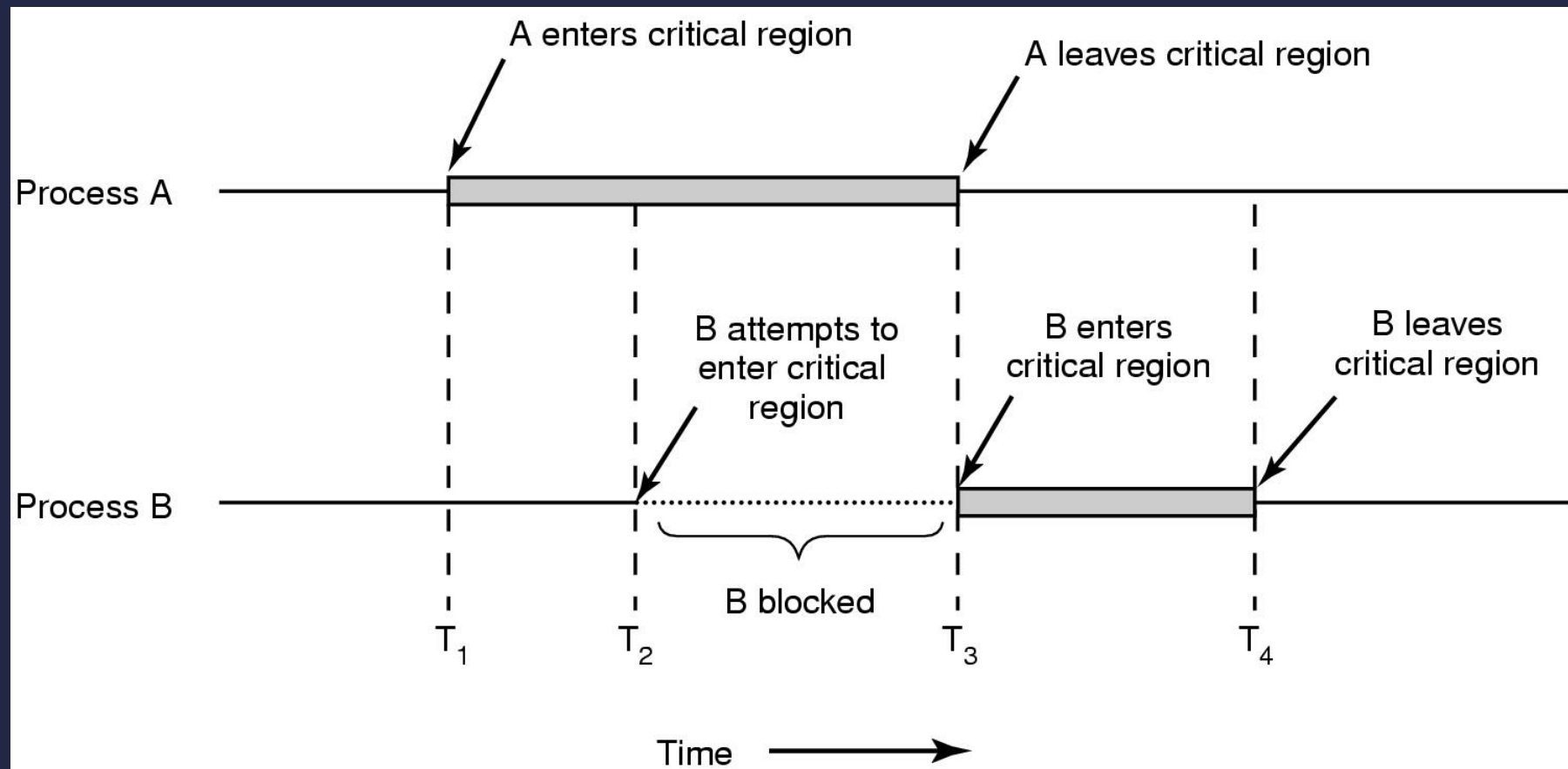
```
use std::sync::Arc;  
let foo = Arc::new(vec![1.0, 2.0, 3.0]);  
// The two syntaxes below are equivalent.  
let a = foo.clone();  
let b = Arc::clone(&foo);  
// a, b, and foo all point to the same shared memory location
```

Arc<T>

- Limitations
 - Arc<T> makes sharing objects thread safe
 - However, it does *not* make using the objects thread safe
 - E.g. methods of the object may be thread unsafe
 - Object within Arc<T> is *immutable*
- Need
 - A construct that allows shared mutable object
 - A construct that makes using objects thread safe

Mutual Exclusion

- Ensures only one thread can access shared data at once



Mutex<T>

- Provides mutual exclusion
- When mutex is locked, no other thread can use object
 - Locking mutex creates a MutexGuard

```
use std::sync::Mutex;
```

```
fn main() {  
    let m = Mutex::new(5);  
    {  
        // num is a MutexGuard around the data  
        let mut num = m.lock().unwrap();  
        *num = 6;  
    } // num goes out of scope and unlocks m  
  
    println!("m = {:?}", m);  
}
```


Mutex<T>

- Provides *interior mutability*
 - The mutex is immutable, but the data it contains is mutable
- Caveat
 - Mutex is not sharable (ownership rule)
- Must be combined with Arc<T>

```
let counter = Arc::new(Mutex::new(0));
for _ in 0..10 {
    let counter = Arc::clone(&counter);
    let handle = thread::spawn(move || {
        let mut num = counter.lock().unwrap();
        *num += 1;
    });
}
```

RwLock<T>

- Same purpose as mutex
 - Optimized for read-mostly objects (i.e. seldom updated)
- Enables multiple readers, single writer

```
use std::sync::RwLock;
let lock = RwLock::new(5);
{
    // many reader locks can be held at once
    let r1 = lock.read().unwrap();
    let r2 = lock.read().unwrap();
    assert_eq!(*r1, 5); assert_eq!(*r2, 5);
}
// read locks are dropped at this point
{
    // only one write lock may be held, however
    let mut w = lock.write().unwrap();
    *w += 1;
    assert_eq!(*w, 6);
}
// write lock is dropped here
```

If you are not sure,
stick with `Mutex<T>`
because `RwLock`'s
write lock is more
expensive than
`Mutex` lock

synchronized

- Language support in Java for mutual exclusion

```
class ThreadedSend extends Thread {  
    private String msg;  
    Sender sender;          // shared among different threads  
  
    ThreadedSend(String m, Sender obj) {  
        msg = m; sender = obj;  
    }  
  
    public void run() {  
        // Only one thread can send message at a time.  
        synchronized(sender) {  
            // synchronizing the send object  
            sender.send(msg);  
        }  
    }  
}
```

synchronized

- Alternatively, can make an entire method critical region

```
class Sender {  
    // Same effect as previous slide, only one thread can send  
    public synchronized void send(String msg) {  
        System.out.println("Sending\t" + msg );  
        try {  
            Thread.sleep(1000);  
        }  
        catch (Exception e) {  
            System.out.println("Thread interrupted.");  
        }  
        System.out.println("\n" + msg + "Sent");  
    }  
}
```

Polling

- Also known as busy looping
 - Continuously lock shared data to check on condition in a loop
 - OK on multiple processors if wait time is short
 - On uniprocessor, reduces performance of entire system
- Example: bounded buffer problem

```
mutex l; char buf[n]; // circular buffer

void send(char msg) {
    lock(l);
    /* buffer is full, keep checking if space becomes available */
    while ((in-out+n)%n == n - 1) { unlock(l); lock(l); }
    buf[in] = msg;
    in = (in + 1) % n;
    unlock(l);
}
```

Message Passing

- Threads communicate by sending message with data
- Allows threads to *synchronize*
 - i.e. thread waits for condition to satisfy before continuing
- Thread sleeps while waiting for message
 - Sleeping thread will not be scheduled to run by OS
- Another thread can wake it up by sending a message
 - Once woken up, thread can check the message

channel<T>

- Creates a sender and a receiver end, thread safe
- Must send/receive same data type

```
use std::thread;
use std::sync::mpsc;

fn main() {
    let (tx, rx) = mpsc::channel();
    thread::spawn(move || {
        let val = String::from("hi");
        tx.send(val).unwrap(); // val moves into send()
    });
    let received = rx.recv().unwrap();
    println!("Got: {}", received);
}
```

mpsc::channel

- Multiple producer, single consumer
- iteration on rx finishes when channel is closed
 - i.e. when all senders close their end

```
let (tx, rx) = mpsc::channel();  
for i in 1..10 {  
    let tx = mpsc::Sender::clone(&tx);  
    thread::spawn(move || {  
        tx.send(String::from("hello")).unwrap();  
    });  
}  
for received in rx {  
    println!("Got: {}", received);  
}
```

Can clone tx to allow
for multiple producers.
rx cannot be cloned!

Monitor

- Allows for both mutual exclusion and synchronization
- Allows for multiple producers and multiple consumers
- Mutual exclusion
 - Provided by a mutex object
- Synchronization
 - Provided by one or more *condition variables*
 - Allows program to define arbitrary condition for synchronization
 - i.e. logic for going to sleep, and waking up others

Condition Variable

- Allows thread to relinquish lock and go to sleep
 - Automatically re-acquires lock prior to wake up

```
use std::sync::{Arc, Mutex, Condvar};
let pair = Arc::new((Mutex::new(false), Condvar::new()));
let pair2 = pair.clone();

thread::spawn(move || {
    let (lock, cvar) = &*pair2;
    let mut started = lock.lock().unwrap();
    *started = true;
    cvar.notify_one(); // notify that the value has changed.
});

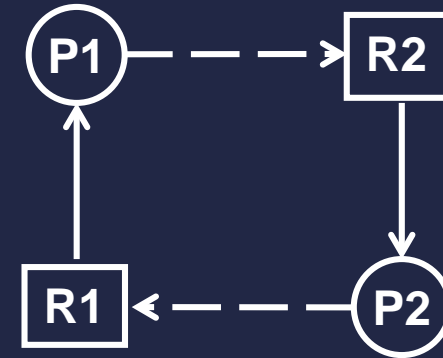
let (lock, cvar) = &*pair;
let mut started = lock.lock().unwrap();
while !*started { // wait for started to become true
    started = cvar.wait(started).unwrap();
}
```

Condvar

- `wait(&self, val: MutexGuard)`
 - Thread waits until condvar is notified, and re-acquire lock on `val` before waking up
- `notify_one(&self)`
 - Wakes up exactly one thread waiting on the condvar
 - Equivalent to `signal()` in other literature
- `notify_all(&self)`
 - Wakes up all threads waiting on the condvar
 - Equivalent to `broadcast()` in other literature

Deadlock

- Circular waiting
- Each thread is holding a resource that the other needs to be able to continue (e.g., two pieces of shared data)
- Rust cannot prevent deadlocks
- Possible solutions
 - Lock ordering
 - Always acquire a set of locks in same order
 - Try lock
 - If one of the locks already taken, release all locks you own and restart



Example

- Multiple producer, multiple consumer problem

```
const MAXLEN: usize = 8;
struct Bounded { buffer: [i32; MAXLEN], top: usize, bottom: usize, }
impl Bounded {
    fn push(& mut self, val: i32) {
        self.buffer[self.top] = val;
        self.top = (self.top + 1) % MAXLEN;
    }
    fn pop(& mut self) -> i32 {
        let val = self.buffer[self.bottom];
        self.bottom = (self.bottom + 1) % MAXLEN;
        val
    }
    fn is_empty(& self) -> bool { self.bottom == self.top }
    fn is_full(& self) -> bool {
        (self.bottom + 1) % MAXLEN == self.top
    }
}
```

Example

- Create a monitor around the bounded buffer

```
use std::sync::{Arc, Mutex, Condvar};
struct Monitor<T> {
    mutex: Mutex<T>,
    empty: Condvar,
    full: Condvar,
}
fn main() {
    let mut threads = vec![];
    let monitor = Arc::new(Monitor {
        mutex: Mutex::new(Bounded {
            buffer: [0; MAXLEN], top: 0, bottom: 0
        }),
        empty: Condvar::new(),
        full: Condvar::new(),
    });
    ...
}
```

Stores all the handles
for each thread so we
can call join() on them.

Example

▪ Producer threads

```
const NPRODUCER: i32 = 3;          const NPRODUCT: i32 = 10;
for i in 1..=NPRODUCER {
    let monitor = monitor.clone();
    threads.push(thread::spawn(move || {
        for j in 0..NPRODUCT {
            let val = i * 10 + j;
            thread::sleep(Duration::from_micros(1));
            let Monitor {mutex, empty, full} = &*monitor;
            let mut circ = mutex.lock().unwrap();
            while circ.is_full() {
                circ = full.wait(circ).unwrap();
            }
            circ.push(val);
            empty.notify_all();
        }
    }));
}
```

Sleep here to mix up
thread execution order

Wait for the bounded
buffer to have space.

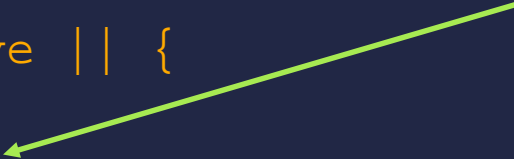
Notify consumer
that data is available

Example

▪ Consumer threads

```
for i in 1..=NCONSUMER {  
    let monitor = monitor.clone();  
    threads.push(thread::spawn(move || {  
        let mut v = vec![];  
  
        while v.len() < NCONSUMED {  
            let Monitor {mutex, empty, full} = &*monitor;  
            let mut circ = mutex.lock().unwrap();  
            while circ.is_empty() {  
                circ = empty.wait(circ).unwrap();  
            }  
            v.push(circ.pop());  
            full.notify_all();  
        }  
  
        println!("thread {} consumed: {:?}" , i, v);  
    }));  
}
```

Collect up to 15 pieces
of data and exit.



Example

- Wait for all threads to finish before main exits

```
fn main() {  
    let mut threads = vec![];  
  
    /* creating producer and consumer threads */  
  
    for child in threads {  
        child.join().unwrap();  
    }  
}
```

- Output

thread 1 consumed: [10, 20, 11, 12, 21, 34, 13, 22, 14, 36, 15, 38, 23, 39, 16]
thread 2 consumed: [30, 31, 32, 33, 35, 37, 24, 17, 18, 25, 19, 26, 27, 28, 29]