

Lecture 09: Mutual Exclusion in async-finish Programs

Vivek Kumar

Computer Science and Engineering

IIIT Delhi

vivekk@iiitd.ac.in



Last Lecture

High productivity

```
void foo() {
    loop_domain_t loop = {0, 8, 1, 1};
    finish([&]() {
        forasync1D (&loop, [=](int i) {
            S(i); // can execute in parallel for all i
        }, MODE_TYPE);
    });
}
```

```
1. finish {
2.   async S1;
3.   //continuation of S1
4.   async S2;
5.   //continuation of S2
6.   S3;
7. }
```

Work-first

```
start_finish();
push_task_to_runtime(Line_3);
S1;
if(Line_3_stolen) return;
push_task_to_runtime(Line_5);
S2;
if(Line_5_stolen) return;
S3;
end_finish();
```

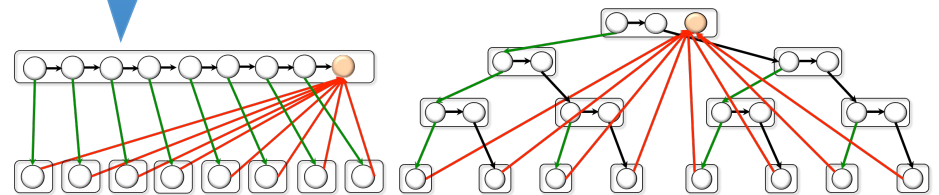
Help-first

```
start_finish();
push_task_to_runtime(S1);
push_task_to_runtime(S2);
S3;
end_finish();
```

Multiple producer
Multiple consumer
(faster task diffusion)

Avoids deque overflow

Smaller CPL



Today's Lecture

- Object based isolation
- Quiz-2

Mutual Exclusion

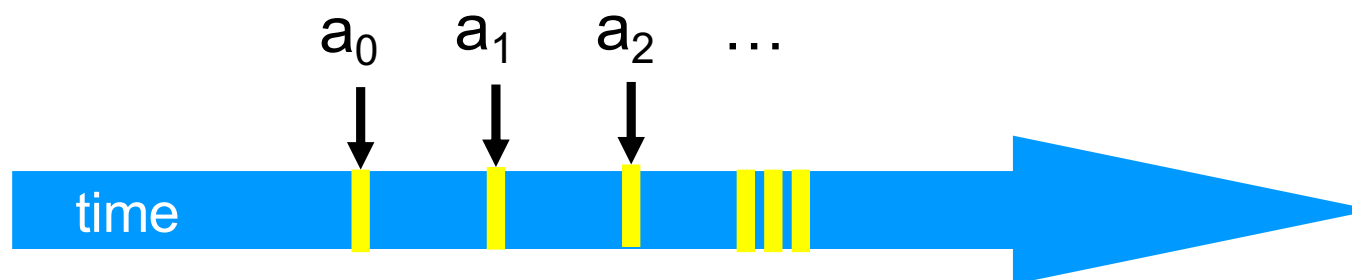
- ***Critical section:*** a block of code that access shared modifiable data or resource that should be operated on by only one thread at a time
- ***Mutual exclusion:*** a property that ensures that a critical section is only executed by a thread at a time.
 - *Otherwise it results in a race condition!*



Acknowledgement: Slides adopted from the companion slides for the book "The Art of Multiprocessor Programming" by Maurice Herlihy and Nir Shavit

Threads

- A *thread* A is (formally) a sequence a_0, a_1, \dots of events
 - Notation: $a_0 \rightarrow a_1$ indicates order



Acknowledgement: Slides adopted from the companion slides for the book "The Art of Multiprocessor Programming" by Maurice Herlihy and Nir Shavit

Example Thread Events

- Assign to shared variable
- Assign to local variable
- Invoke method
- Return from method
- Lots of other things ...

Acknowledgement: Slides adopted from the companion slides for the book "The Art of Multiprocessor Programming" by Maurice Herlihy and Nir Shavit

Concurrency

- Thread A



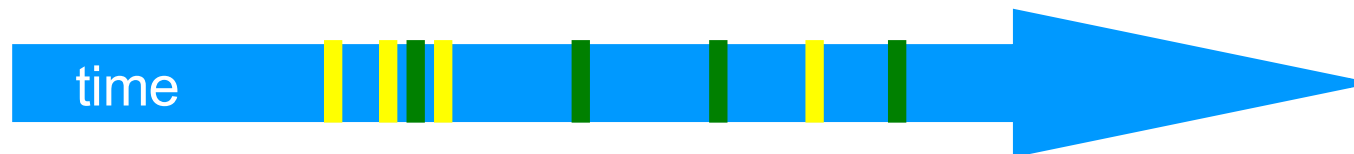
- Thread B



Acknowledgement: Slides adopted from the companion slides for the book "The Art of Multiprocessor Programming" by Maurice Herlihy and Nir Shavit

Interleavings

- Events of two or more threads
 - Interleaved
 - Not necessarily independent (why?)



Acknowledgement: Slides adopted from the companion slides for the book "The Art of Multiprocessor Programming" by Maurice Herlihy and Nir Shavit

What we Learned in Lecture 05

- We saw two different cases of data-races using the examples of parallel ArraySum and parallel MatrixMultiplication. We were able to resolve these data-races by correct placements of `async` and `finish`
- However, there are many cases in practice when two tasks legitimately need to perform conflicting accesses to shared locations without incurring data-races

Incrementing Pointer Content in Parallel

```
void increment(uint64_t* pointer, uint64_t iterations) {
    loop_domain_t loop = {0, iterations, 1, 1 };
    finish ([&]() {
        forasync1D(&loop, [=](uint64_t i) {
            *pointer += 1;
        }, FORASYNC_MODE_RECURSIVE);
    });
}
```

Critical Section

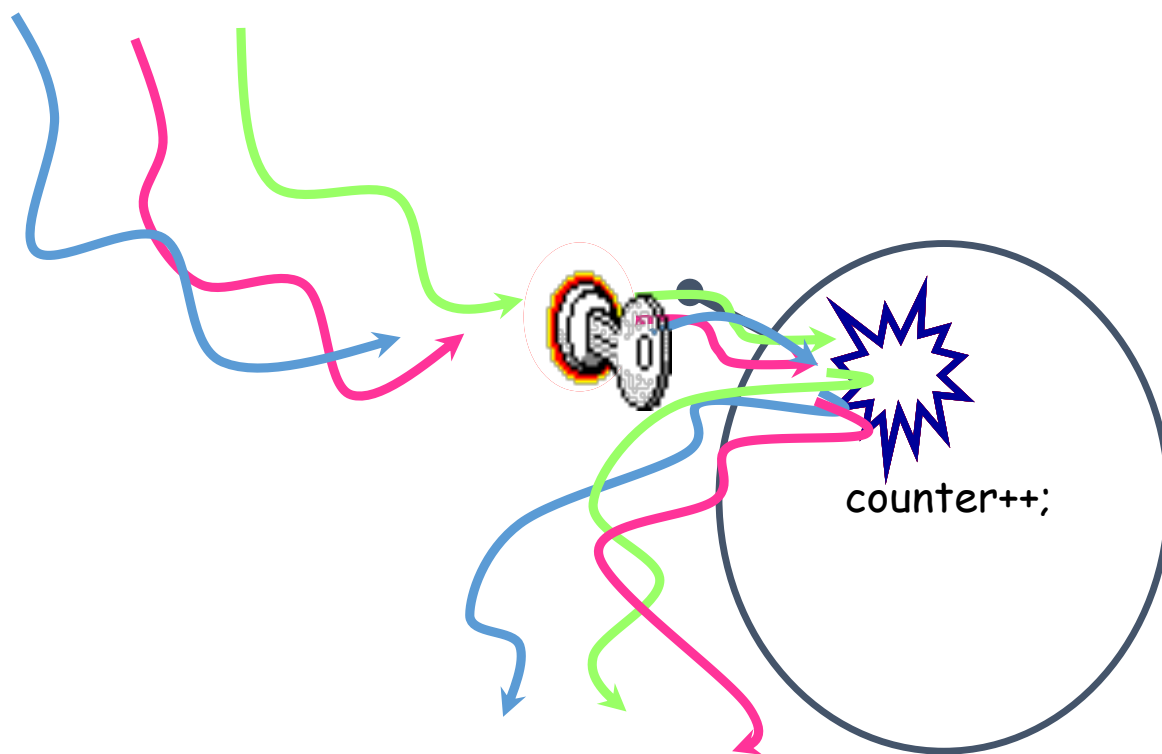
Incrementing Pointer Content in Parallel

```
void increment(uint64_t* pointer, uint64_t iterations) {
    loop_domain_t loop = {0, iterations, 1, 1 };
    finish ([&]() {
        forasync1D(&loop, [=](uint64_t i) {
            pthread_mutex_lock(&mutex);
            *pointer += 1;
            pthread_mutex_unlock(&mutex);
        }, FORASYNC_MODE_RECURSIVE);
    });
}
```

Critical Section

One way to resolve this data-race is by using a mutex lock

Visualizing Mutual Exclusion



- Only one **thread** can get the “key” to enter the critical section
- Rest all **threads** will be queued to get the key

Properties of a Good Locking Algorithm

- Safety guarantees
 - *Mutual exclusion*
- Progress guarantees
 - *Deadlock freedom*: system as a whole makes progress.
If some thread calls **lock()** and never returns, then other threads must complete **lock()** and **unlock()** calls infinitely often
 - *Starvation freedom*: A thread should not indefinitely hold the lock for doing some big computation while other threads keep waiting to get this lock

Acknowledgement: Slides adopted from the companion slides for the book "The Art of Multiprocessor Programming" by Maurice Herlihy and Nir Shavit

isolated Construct in HCLib

```
isolated([&]( ) { S; });
```

- Isolated construct identifies a critical section
 - Introduced by Habanero-Java that also has a very mature implementation of `isolated`
 - *HCLib currently has an experimental implementation of `isolated`*
- Two tasks executing isolated constructs are guaranteed to perform them in mutual exclusion
 - Isolation guarantee applies to (isolated, isolated) pairs of constructs, not to (isolated, non-isolated) pairs of constructs
- No parallelism constructs inside `isolated`
 - E.g., if `async` is spawned then isolation guarantee will only apply to the creation of `async`, not to its execution
- Isolated constructs can never cause a deadlock
 - Other techniques used to enforce mutual exclusion (e.g., locks) can lead to a deadlock, if used incorrectly

Source: <https://wiki.rice.edu/confluence/download/attachments/24426821/comp322-s16-lec20-slides-v1.key.pdf?version=1&modificationDate=1483206145246&api=v2>

Incrementing Pointer Content in Parallel

```
void increment(uint64_t* pointer, uint64_t iterations) {
    loop_domain_t loop = {0, iterations, 1, 1 };
    finish ([&]() {
        forasync1D(&loop, [=](uint64_t i) {
            isolated([=]() {
                *pointer += 1;
            });
        }, FORASYNC_MODE_RECURSIVE);
    });
}
```

Critical Section

Using isolated statement instead of a mutex lock

Code available on github: <https://github.com/vivkumar/cse502/blob/master/hclib/test/lec12/>

Another Example: Fibonacci Reducer

```
uint64_t result = 0;
void fib(uint64_t n) {
    if(n < THRESHOLD) {
        uint64_t value = fib_sequential(n);

        result += value;
    }
    else {
        async( [= ]() {
            fib(n-1);
        });
        fib(n-2);
    }
}

int main(int argc, char** argv) {
    finish ( [= ]() {
        fib(n);
    });
    printf("Fib(%" PRIu64 ") is %" PRIu64 "\n",n, result);
}
```

Is this Correct ?

Another Example: Fibonacci Reducer

```
uint64_t result = 0;
void fib(uint64_t n) {
    if(n < THRESHOLD) {
        uint64_t value = fib_sequential(n);
        isolated( [=]() {
            result += value;
        });
    }
    else {
        async( [=]() {
            fib(n-1);
        });
        fib(n-2);
    }
}

int main(int argc, char** argv) {
    finish ( [=]() {
        fib(n);
    });
    printf("Fib(%" PRIu64 ") is %" PRIu64 "\n",n, result);
}
```

Critical Section

Code available on github: <https://github.com/vivkumar/cse502/blob/master/hclib/test/lec12/>

Is that Enough?

```

uint64_t result = 0;
void fib(uint64_t n) {
    if(n < THRESHOLD) {
        uint64_t value = fib_sequential(n);
        isolated(=[=]() {
            result += value;
        });
    }
    else {
        async(=[=]() {
            fib(n-1);
        });
        fib(n-2);
    }
}

int main(int argc, char** argv) {
    finish (=[=]() {
        fib(n);
    });
    printf("Fib(%" PRIu64 ") is %" PRIu64 "\n",n, result);
}

```

This seems like it is just a syntactic sugar to replace `pthread_mutex_lock` and `pthread_mutex_unlock` ?
 What is so special about `isolated` ?



Money Transaction Between Accounts

```
class Account {
    int id;
    double balance;
    void debit(double amount);
    void credit(double amount);
};
```

```
class Transfer {
    Account source, destination;
    double amount;
    void run() {
        source.debit(amount);
        destination.credit(amount);
    }
};
```

```
class Bank {
    void fund_transfer() {
        Accounts numAccounts[N];
        Transfer pending[TOTAL];
        for(int i=0; i<TOTAL; i++) {
            pending[i].run();
        }
    }
};
```

- How to parallelize?

Money Transaction Between Accounts

```
class Account {
    int id;
    double balance;
    void debit(double amount);
    void credit(double amount);
};
```

```
class Transfer {
    Account source, destination;
    double amount;
    void run() {
        source.debit(amount);
        destination.credit(amount);
    }
};
```

```
class Bank {
    void fund_transfer() {
        Accounts numAccounts[N];
        Transfer pending[TOTAL];
        loop_domain_t loop = {0, TOTAL, 1,1};
        finish([&]() {
            forsync1D( [=](int i) {
                pending[i].run();
            }, FORASYNC_MODE_RECURSIVE);
        });
    }
};
```

- How to parallelize?
 - We can use forasync1D to parallelize along with a finish
 - **Any issues?**

Money Transaction Between Accounts

```
class Account {
    int id;
    double balance;
    void debit(double amount);
    void credit(double amount);
};
```

```
class Bank {
    void fund_transfer() {
        Accounts numAccounts[N];
        Transfer pending[TOTAL];
        loop_domain_t loop = {0, TOTAL, 1,1};
        finish([&]() {
            foasync1D(=[](int i) {
                pending[i].run();
            }, FORASYNC_MODE_RECURSIVE);
        });
    }
};
```

```
class Transfer {
    Account source, destination;
    double amount;
    void run() {
        isolated([&]() {
            source.debit(amount);
            destination.credit(amount);
        });
    }
};
```

- How to parallelize?
 - Do we still have parallelism?

Money Transaction Between Accounts

```
class Account {
    int id; double balance;
    void lock() {...} void unlock() {...}
    void debit(double amount);
    void credit(double amount);
};
```

```
class Transfer {
    Account source, destination;
    double amount;
    void run() {
        source.lock(); destination.lock();
        source.debit(amount);
        destination.credit(amount);
        destination.unlock(); source.unlock();
    }
};
```

```
class Bank {
    void fund_transfer() {
        Accounts numAccounts[N];
        Transfer pending[TOTAL];
        loop_domain_t loop = {0, TOTAL, 1,1};
        finish([&]() {
            foasync1D(=[](int i) {
                pending[i].run();
            }, FORASYNC_MODE_RECURSIVE);
        });
    }
};
```

- Is this correct?

Money Transaction Between Accounts

```
class Account {
    int id; double balance;
    void lock() {...}
    void debit(double amount) {...}
    void credit(double amount) {...}
};
```

```
class Bank {
    void fund_transfer(Account source, Account destination, double amount) {...}
    Accounts numAccounts;
    Transfer pending;
    loop_domain_t loop_domain;
    finish([&]() {
        foasync1D([
            pending[i],
        ], FORASYNC);
    });
};
```

```
class Transfer {
    Account source;
    Account destination;
    double amount;
```

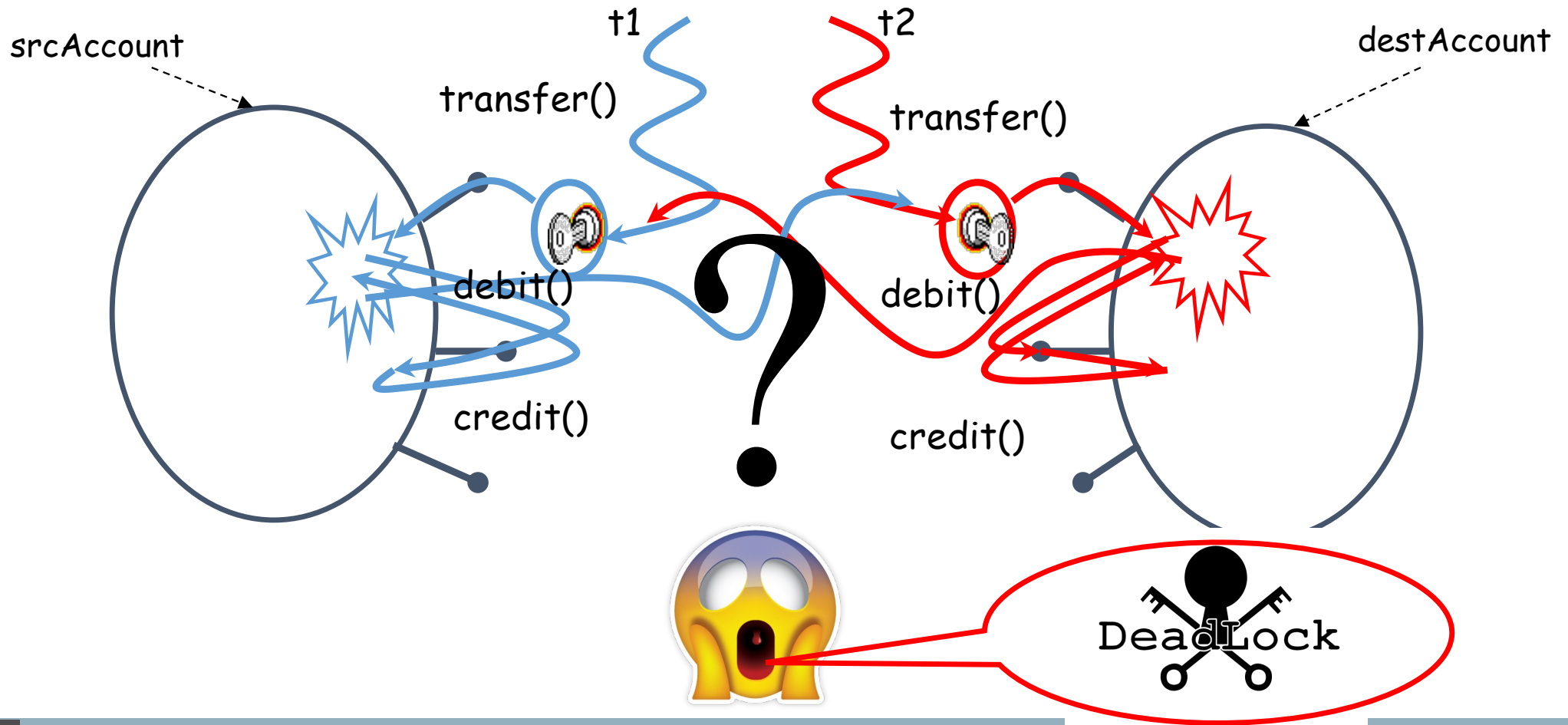
```
    .lock();
    ;
    source.unlock();
```



DEADLOCK

© CanStockPhoto.com - csp50107200

Let's Analyze Our Money Transaction



Deadlock Avoidance

- **Deadlock occurs when multiple threads need the same locks but obtain them in different order**
- Not so easy to avoid deadlocks
- It's an active research area

Deadlock Avoidance

- Lock timeout
 - Put a timeout on lock attempts
 - `pthread_mutex_timedlock`
- Lock ordering
 - Ensure that all locks are taken in same order by any thread
 - **Let's try using it to fix our Bank Transaction program**

Object Based Isolation for Avoiding Deadlock in async-finish Program

`isolated(obj1, obj2, ..., lambda_function)`

- In this case, programmer specifies list of objects for which isolation is required
- Mutual exclusion is only guaranteed for instances of isolated constructs that have a common object in their object lists
 - Standard isolated is equivalent to “isolated(*)” by default i.e., isolation across all objects
- Experimental implementation exists in HCLib (some APIs might change in future but not the concepts)

Money Transaction Between Accounts

```
class Account {
  int id;
  double balance;
  void debit(double amount);
  void credit(double amount);
};
```

```
class Transfer {
  Account source, destination;
  double amount;
  void run() {
    isolated(&source, &destination, [&]() {
      source.debit(amount);
      destination.credit(amount);
    });
  }
};
```

```
class Bank {
  void fund_transfer() {
    Accounts numAccounts[N];
    Transfer pending[TOTAL];
    loop_domain_t loop = {0, TOTAL, 1,1};
    enable_isolation_n(numAccounts, N);
    finish([&]() {
      foasync1D( [=](int i) {
        pending[i].run();
      }, FORASYNC_MODE_RECURSIVE);
    });
    disable_isolation_n(numAccounts, N);
  }
};
```

- This will never deadlock. You can have objects in any order
- **Experimental support, hence outside of finish. In future you may not even need these two API**

Implementation of Object Based Isolation

- **enable_isolation_n(numAccounts, N);**
 - Runtime will add these objects in a hashmap
 - Every object is associated with a unique **uint64_t counter** and a lock
- **isolated(&source, &destination, lambda);**
 - Runtime will get these objects from the hashmap and then sort them using the value of their **counter**
 - Lock is then acquired on each object in the ascending (or descending) value of the their **counter**
 - User provided critical section is executed and then each of these objects are unlocked (in same order)
 - This technique avoids the deadlock
- **disable_isolation_n(numAccounts, N);**
 - Remove these objects from the hashmap

Pros and Cons of Object Based Isolation

- Pros

- Productivity: simpler approach than “locks”
- Deadlock-freedom property is guaranteed

- Cons

- Programmer needs to worry about getting the object list right
- Objects in object list can only be specified at start of the isolated construct (new objects cannot be added later on)

Next Lecture

- Futures, promises, and data driven tasks