A practical course on

Advanced systems programming in C/Rust

Jörg Thalheim

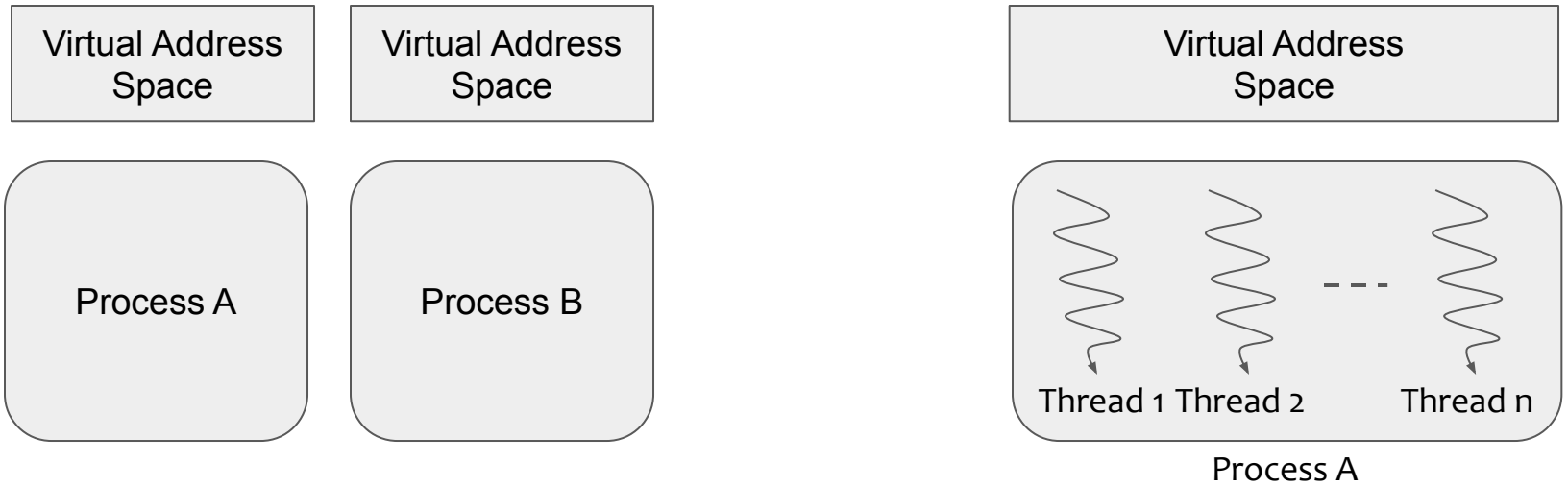
Today's topic!

Concurrency and Synchronization

Shady Issa

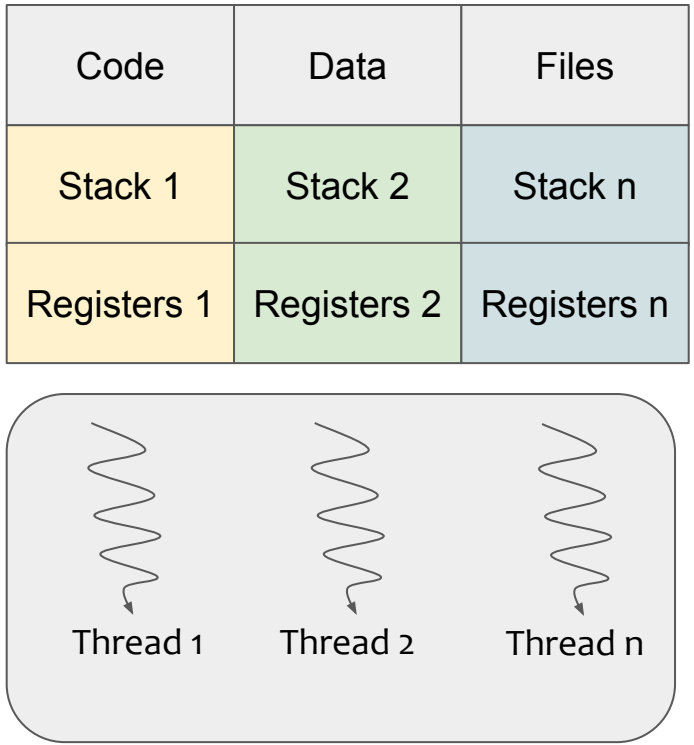
Processes vs Threads

- Last lecture: inter-process parallelism
- Today: intra-process parallelism
- Intra-process parallelism is more relevant with the ubiquitous multi-core architectures
- NB: these are not mutually exclusive modes of operation



Threads

- Threads allow for dividing a program into independent units of computation
- Each thread has:
  - Program counter
  - Thread id
  - Stack
  - Set of registers
- Created using the `clone` system call



POSIX Threads API

- Create a thread

```
int pthread_create(pthread_t *restrict thread,
                   const pthread_attr_t *restrict attr,
                   void *(*start_routine)(void *),
                    void *restrict arg);
```

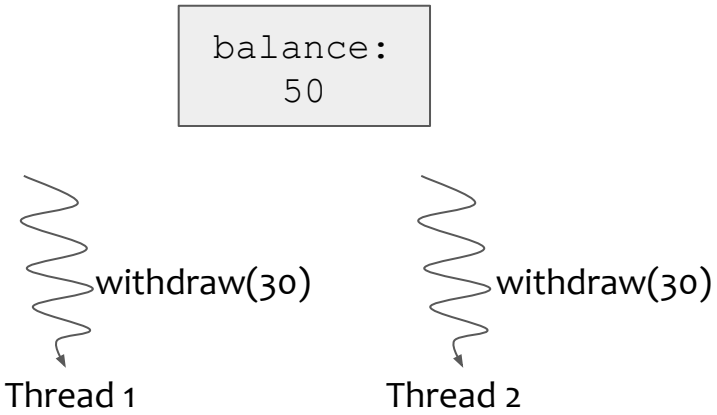
- Attributes: `pthread_attr_t`
  - joinable vs detachable
  - stack size
- Wait for another thread to finish

```
int pthread_join(pthread_t thread, void **retval);
```

The Need for Synchronization

- Threads share the same resources
  - heap, files, I/O
- Program correctness can not be guaranteed when two threads access the resource concurrently

```
withdraw (amount) {
 if (balance > amount)
   balance -= amount
}
```



Threads must synchronize accesses to shared resources

Synchronization

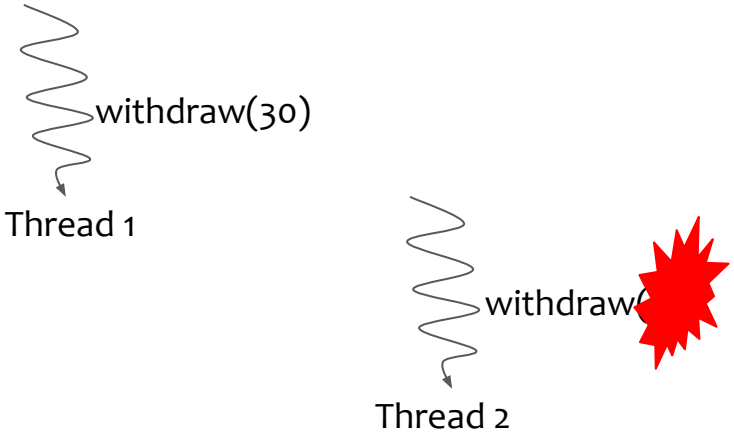
- There exists different mechanisms for synchronization
  - Mutual exclusion
  - Lock-free
  - Read-Copy-Update
  - Transactional memory

- Thread-safety

Mutual Exclusion

- To grant a thread exclusive access to a shared resource
- Mutex or a lock
  - A token that can be held by on thread at a time
  - Each thread must *acquire* the mutex *before* accessing the shared resource
  - Each thread must *release* the mutex *after* accessing the shared resource
- *Critical sections* will be executed *atomically*

```
withdraw (amount) {
acquire_mutex()
 if (balance > amount)
   balance -= amount
release_mutex()
}
```



Types of Mutexes

- Mutexes and locks are usually used interchangeably
  - The same mutex can be used across different processes
- Semaphores
  - Can allow multiple threads at a time
  - They are usually used for signalling
- Read-writer locks
  - Allows readers to execute concurrently, but only one writer a time
    - e.g.: check_balance accesses a shared resource but does not modify it

Implementing Locks

- There is a plethora of algorithms for implementing locks
  - Test-and-set, Test-test-and-set, MCS, ticket locks, etc,
- Hardware atomic primitives
  - Read-Modify-Write
  - Atomic-Increment-Fetch

```
read_modify_write(int *var, int old_value, int new_value){
    if (*var == old_value){
        *var = new_value;
    }
    return *var;
}
```

Atomic

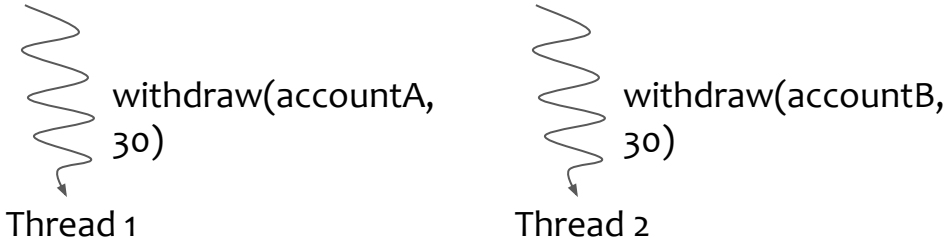
```
acquire_mutex(){
    while( read_modify_write(&mutex, 0, 1) != 1) ;
}
```

Pthreads Synchronization

- Pthreads library has several mutexes:
  - Pthread_mutex
    - threads enter sleep mode when lock is busy
    - pthread_mutex_lock, pthread_mutex_unlock, pthread_mutex_trylock
  - pthread_spinlock
    - threads keep polling the status of the lock
    - pthread_spin_lock, pthread_spin_unlock, pthread_spin_trylock
  - pthread_rwlock
    - pthread_rwlock_rdlock, pthread_rwlock_wrlock
  - sem
    - sem_post, sem_wait
- Other synchronization mechanisms:
  - pthread_cond
  - pthread_barrier

What about Performance?

```
withdraw (account, amount) {
acquire_mutex()
 if (account.balance > amount)
   balance -= amount
release_mutex()
}
```



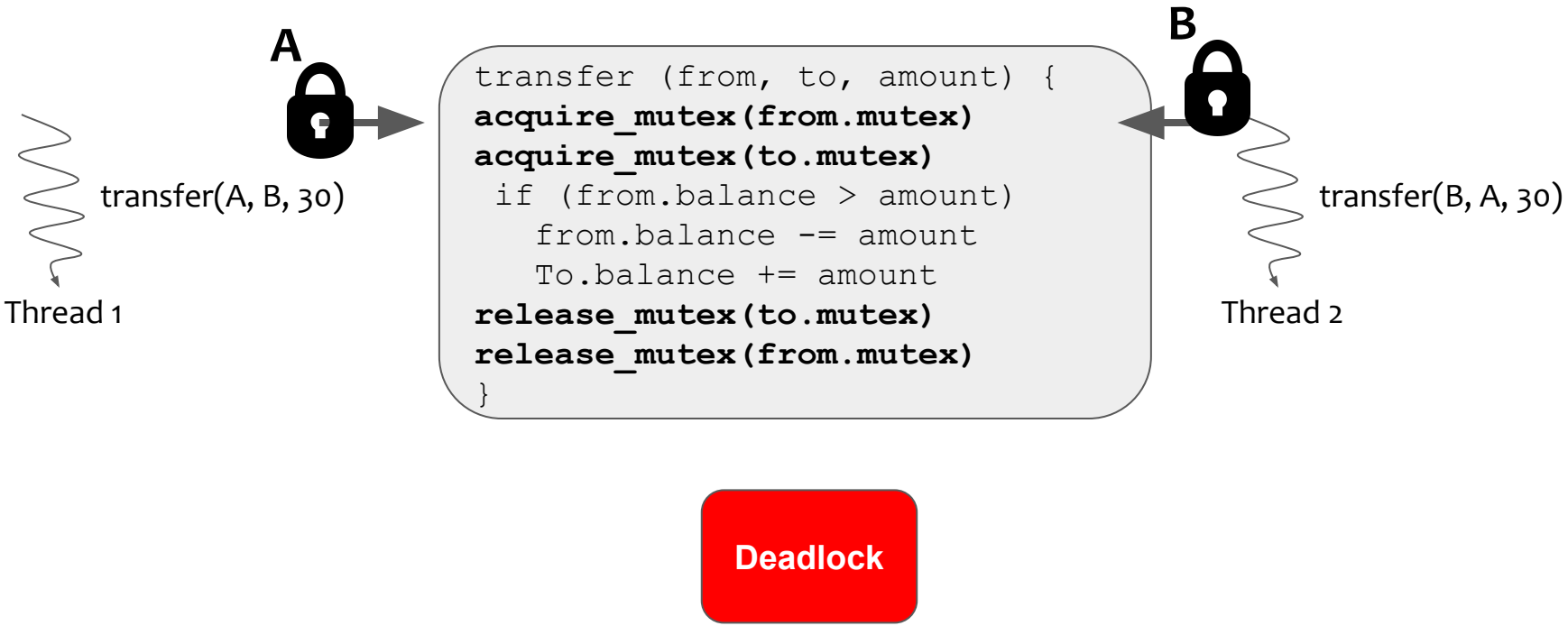
Synchronization can hamper scalability and performance

Scalable Synchronization

- Coarse-grained vs fine-grained locking
  - single global lock vs per-account lock
- Can allow higher parallelism
- Introduces memory overhead
  - Lock pery array entry
- Challenging to design

```
withdraw (account, amount) {
acquire_mutex(account.mutex)
 if (account.balance > amount)
   balance -= amount
release_mutex(account.mutex)
}
```

Scalable Synchronization



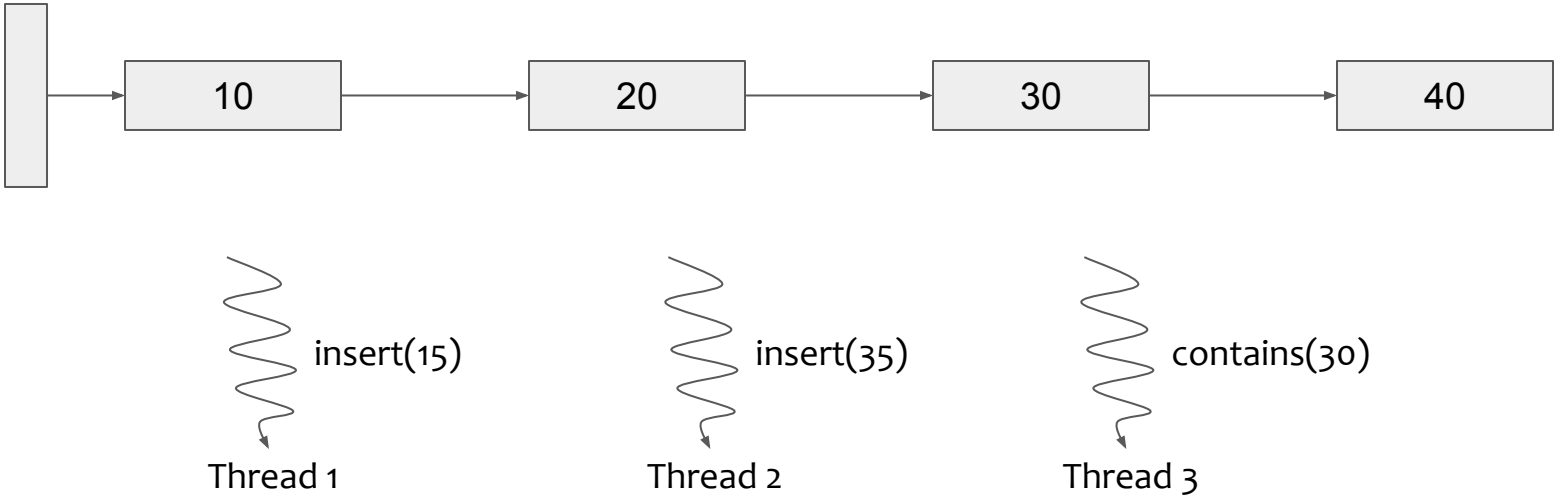
Scalable Synchronization

- Coarse-grained vs fine-grained locking
  - single global lock vs per-account lock
- Can allow higher parallelism
- Introduces memory overhead
  - e.g.: Lock pery array entry
- Challenging to design
  - Deadlocks: threads can not make progress
  - Livelocks: threads can not make useful progress
  - Fairness and starvation

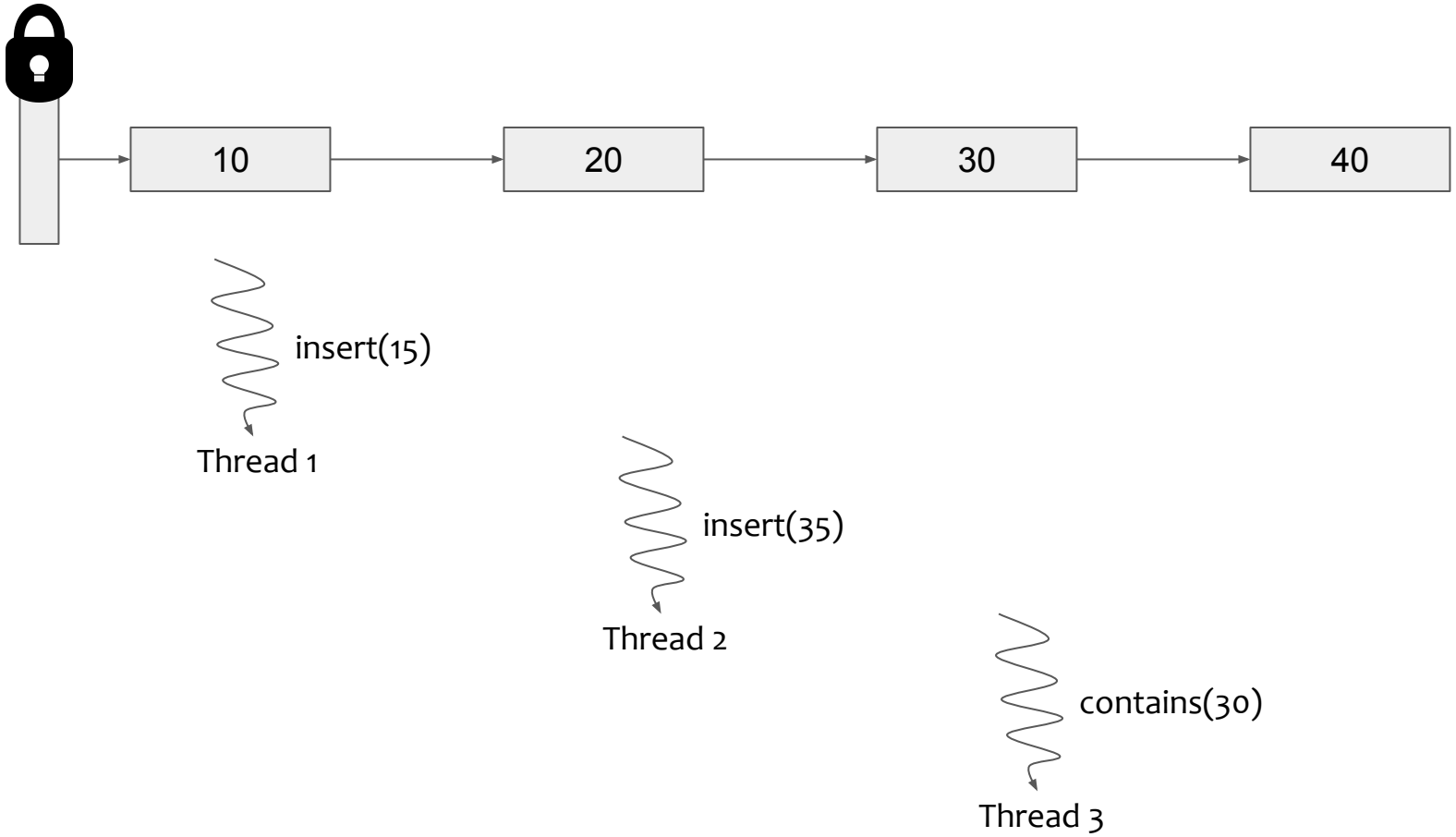
```
withdraw (account, amount) {
acquire_mutex(account.mutex)
 if (account.balance > amount)
   balance -= amount
release_mutex(account.mutex)
}
```

Lock-free Programming

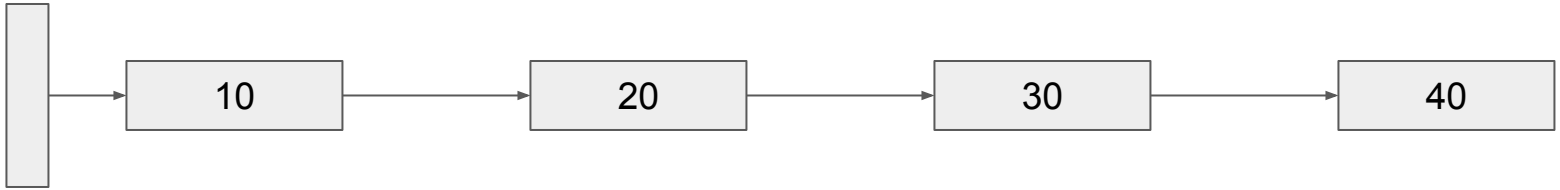
- Avoid using locks and rely on atomic primitives
  - Threads do not have to wait for a lock to be free



Lock-free Programming

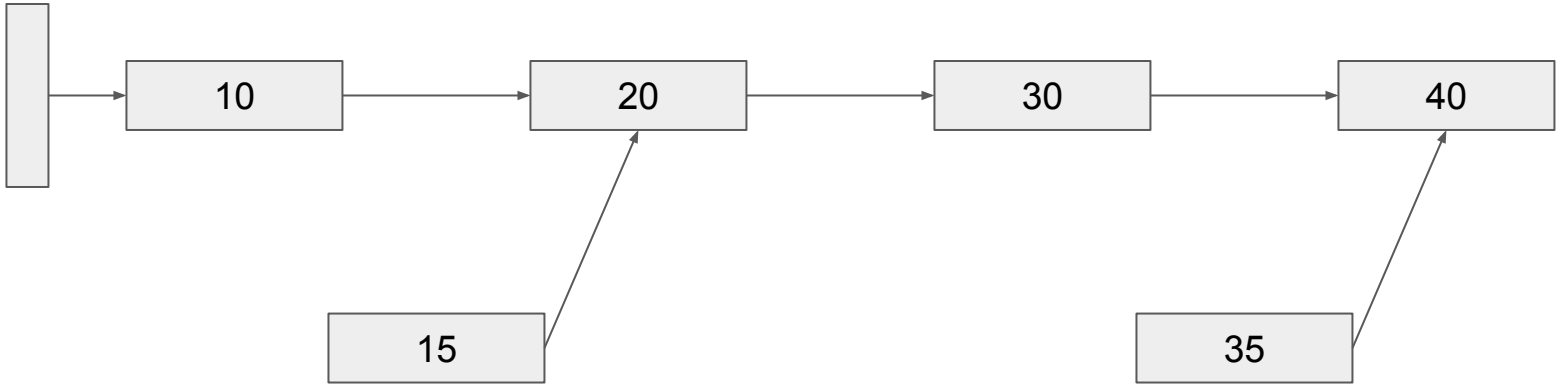


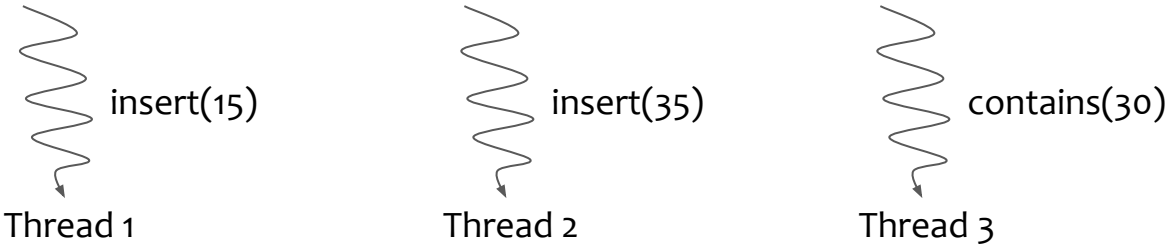
Lock-free Programming



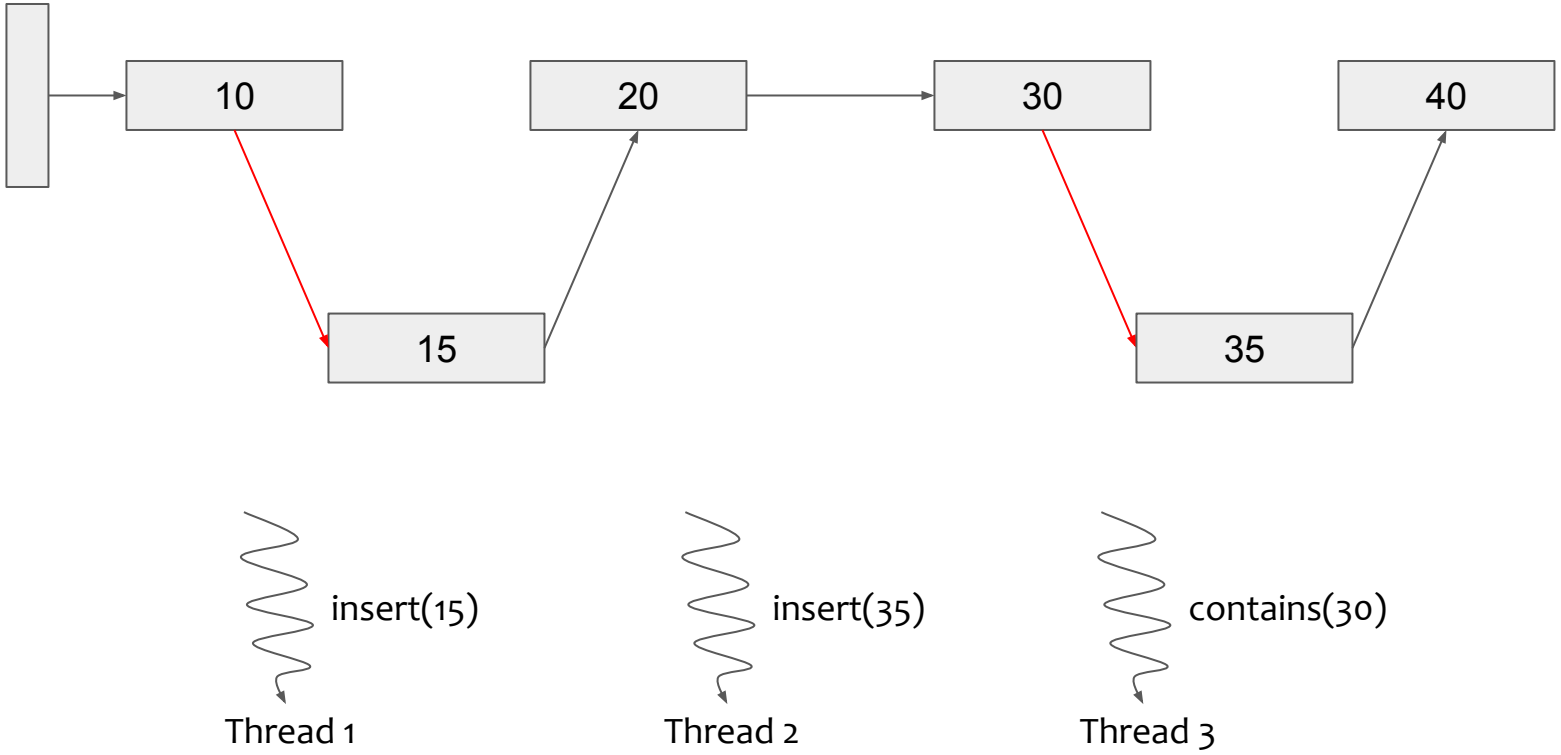

```
insert (value) {
     new_node.value = value
     node = find_prev(value)
     new_node.next = node.next
     read_modify_write(&node.next, node.next, new_node);
}
```

Lock-free Programming





Lock-free Programming



Challenges of Lock-free Programming

- Similar challenges to fine-grained locking
  - Starvation
  - Livelocks
- Memory reclamation
  - Threads can not tell if other threads are referencing some data
  - When is it safe to free an object?

Conclusion

- Threads allow for lightweight intra-process concurrency
- Synchronization is needed to ensure program correctness
- Hardware atomic primitives allow for implementing efficient synchronization mechanisms
- Scalable synchronization is challenging
- Lock-free algorithms can further increase performance