# Lecture 06: Managing Overheads from Blocking Tasks & Deques

Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

# Last Lecture (Recap)

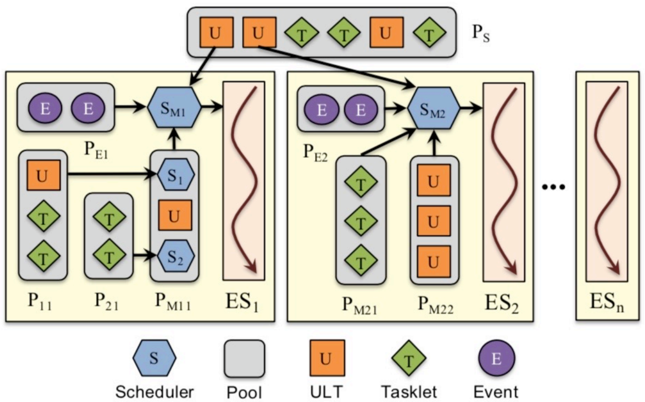
- Producer consumer using a single fiber manager
- Work-stealing scheduling using fibers
- Argobots runtime

```
boost::fibers::mutex mtx;
boost::fibers::condition_variable cnd;
std::string str;

boost::fibers::fiber f1([=]() {
  std::unique_lock<boost::fibers::mutex> lck(mtx);
  if(str.size() == 0) {
    cnd.wait(lck);
  }
  cout << str << endl;
});

boost::fibers::fiber f2([=]() {
  std::unique_lock<boost::fibers::mutex> lck(mtx);
  str = "Hello Fiber";
  cnd.notify_one();
});

f1.join();
f2.join();
```




```
// mutex and condition variable for shutdown
boost::fibers::mutex mtx;
boost::fibers::condition_variable cnd;
int pool_size;

int main() {
  // Step-1: Launch pool_size-1 number of workers calling
"worker_routine"
  // Step-2: Set scheduling policy at fiber manager
  boost::fibers::use_scheduling_algorithm
        <boost::fibers::algo::work_stealing>(pool_size);
  // Step-3: Handshake with each of the spawned workers
  // Step-4: Launch compute kernel having fiber based concurrency
  int result = compute_kernel(/*some input*/);
  // Step-5: Shutdown the thread pool
  std::unique_lock<boost::fibers::mutex> lck(mtx);
  cnd.notify_all();
}
```

```
// mutex and condition variable for shutdown
boost::fibers::mutex mtx;
boost::fibers::condition_variable cnd;
int pool_size;
void worker_routine(int id) {
  // Step-1: Set scheduling policy at fiber manager
  boost::fibers::use_scheduling_algorithm
        <boost::fibers::algo::work_stealing>(pool_size);
  // Step-2: Handshake with the master worker
  // Step-3: Suspend current fiber until master signals
  std::unique_lock<boost::fibers::mutex> lck(mtx);
  cnd.wait(lck);
}
```

```
int compute_kernel(int arg) {
  // Step-1: Wrap callable target to asynchronously compute the return value
  boost::fibers::packaged_task<int()> task([=]() {
    // Launch computation that may also recursively spawn more fibers
    return value;
  });
  // Step-2: Get the future object associated with the above target
  boost::fibers::future<int> future = task.get_future();
  // Step-3: Spawn the fiber and detach it to enable work-stealing
  boost::fibers::fiber(std::move(task)).detach();
  // Step-4: Wait for the fiber to complete
  return future.get();
}
```

# Today's Class

- Mixing blocking tasks with async tasks
- Sequential overheads
  - Alternative deques

# Application with Mixed Task Types

- Task parallelism primarily focuses on optimizing compute intensive applications
  - Can we support both blocking & non-blocking tasks in this programming model?

# Program with Async & Blocking Future.Get

```
uint64_t fib(uint64_t n) {
  if(n<2) {
    return n;
  } else {
    std::future<uint64_t> f1 = std::async([=](){ return fib(n-1); });
    std::future<uint64_t> f2 = std::async([=](){ return fib(n-2); });
    //get will block until result is ready
    return f1.get() + f2.get();
  }
}
```

What are pros and cons of designing a work-stealing runtime that either uses plain task or ULT for creating an async?

- Parallel runtime that uses ULTs for an async
  - **Pros**: Calling get on not-ready futures will move ULT to suspended queue automatically by the corresponding ULT manager
  - **Cons**: ULTs have significant high memory footprint than plain async
- Parallel runtime that uses plain task for an async
  - **Pros**: Small memory footprint as memory required only for storing the async lambda (few bytes)
  - **Cons**: Calling get on not-ready futures will block the worker (KLT)
- General approach for above type of tasking pattern
  - Create plain tasks for async that do not yield, and use boost fcontext to context switch to another thread stack while encountering a blocking future.get (similar to HClib,TBB, etc.)
    - **Cons**: Plain task implementation of async will not be able to yield, and will always run to completion

# Program with Async & Blocking IO

- IO based operations
  - Reading / writing over the socket
  - Getting input from user using keyboard or mouse

# Program with Async & Blocking IO

```
std::thread T1([=]() { /* Some IO operation */ });
T1.join();
.....
future<int> F1 = async([&]() { x = fib(n-1); });
future<int> F2 = async([&]() { y = fib(n-2); });

.....
std::thread T1([=]() { /* Some IO operation */ });
T2.join(); // Would F2 be moved into the suspended queue?
```

```
boost::fibers::fiber F1([=]() { /* Some IO operation */ });
F1.get(); // Would F1 be moved into the suspended queue?

boost::fibers::fiber F2([=]() { /* Some IO operation */ });
F2.get(); // Would F2 be moved into the suspended queue?
```

*Several recently published papers targeted this scenario*

- Spawning a new thread to handle IO will lead to time sharing of CPUs with thread pool worker
- Fiber library doesn't know about the blocking IO operations (similar to using std::condition_var instead of boost::fibers::condition_var)
    - **Solution:** Extra runtime logic needed to move IO fiber/ULT into suspended queue, and start working on an item from the ready queue

# Asynchronous IO in Linux

We don't want this approach

```
/* Wasted CPU cycles probing the connections */
for(int i=0; i<NUM_IO_CONNECTIONS; i++) {
  if(FD[i] has new input) {
    process_IO_connection(i);
  }
}
```

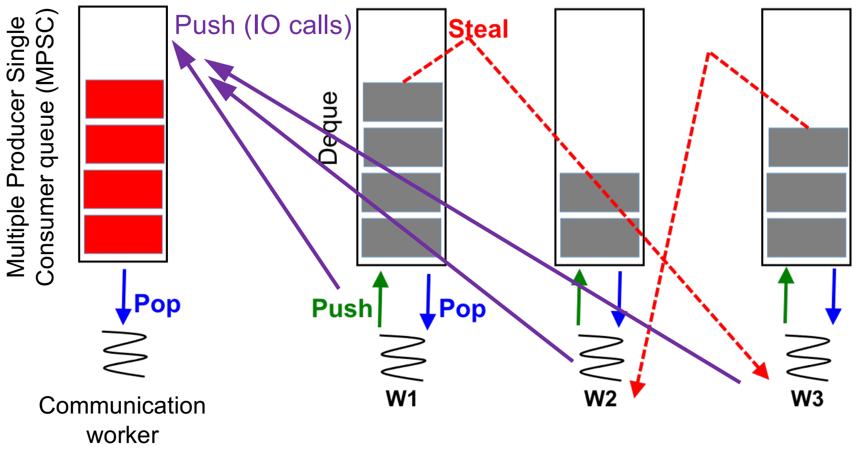
- All I/O devices in Linux are presented as files

```
/* Create an epoll instance */
int EP = epoll_create(0);
/* Add file descriptors to epoll watch list */
struct epoll_event EV[NUM_IO_CONNECTIONS];
for(int i=0; i<NUM_IO_CONNECTIONS; i++) {
  EV[i].data.fd = FD[i];
  epoll_ctl(EP, ..., &EV[i]);
}
while(!shutdown) {
  /* Do something else while waiting for IO */
  int nFDReady = epoll_wait(EP, EV,  NUM_IO_CONNECTIONS, 0 /*timeout*/);
  if(nFDReady > 0) process_IO_connection(...);
  else Do_something_else();
}
close(EP);
```

We want this approach

# Runtime to Handle Blocking IO Tasks

```
// File descriptor
FD = open_IO_connection(...);
// Created by computation workers
future io1 = async_read(FD, Buffer, Nbytes);
future io1 = async_write(FD, Buffer, Nbytes);
// Context switch happens
// get satisfied by communication worker
io1.get();
io2.get();
```




*Similar implementation: https://www.cse.wustl.edu/~angelee/home_page/papers/futureIO.pdf*

- Create one communication worker apart from the regular "**N**" computation workers
- Communication worker pinned to Core-0 along with the computation worker-1
- async_read / async_write
  - Create a task that contains: a) FD, b) Buffer, c) Nbytes, d) promise object

    std::promise<T> **P**;

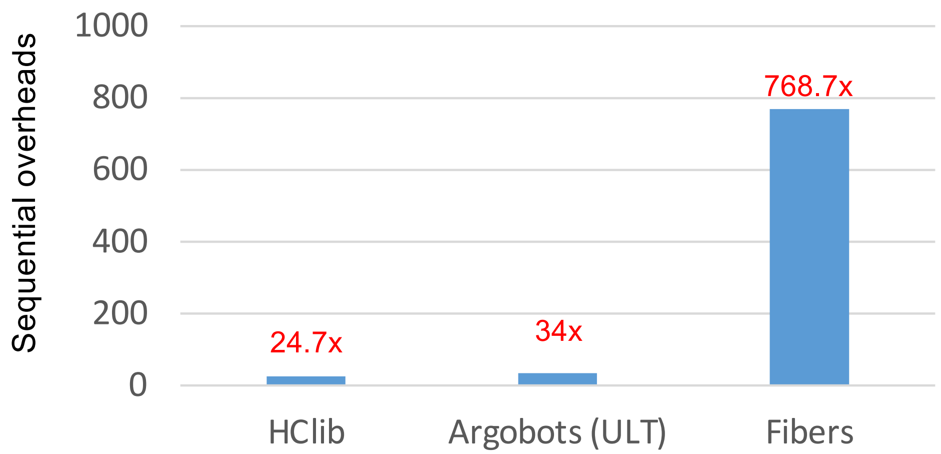
    std::future<T> **F** = **P**.get_future();
  - Push this task to communication worker's MPSC queue and return the future associated with the promise object
- Communication worker remains asleep and awakes at regular intervals to:
  1. Pop and process pending IO task from its MPSC queue
     - Step-1: add to epoll watch list
     - Step-2: Create promise object and return
  2. Notify about the IO device (FD) with pending request that has now become ready
     - Complete the ready IO operation
     - Performs a **put** operation on the associated promise object (**P**) which will move the task waiting on this future from the blocking queue into the ready queue
       - P.set_value(…)

# Today's Class

- Mixing blocking tasks with async tasks
- Sequential overheads
  - Alternative deques

# Sequential Overhead in Fibonacci

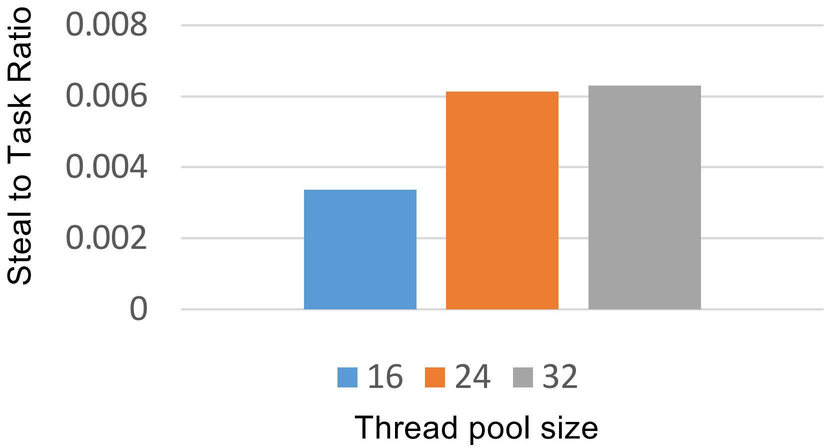
```
uint64_t fib(uint64_t n) {
  if (n < 2) {
    return n;
  } else {
    uint64_t x = fib(n-1); // Spawned as async task
    uint64_t y = fib(n-2); // Called sequentially
    return (x + y);
  }
}
```




- Graph shows the sequential overhead of calculating recursive Fibonacci(30) that spawns task / ULT for every fib(n-1) recursive call until n<2
  - HClib uses tasks
  - Fibers and Argobots uses ULT
- Sequential overhead = $Time_{seq}$ / $Time_{Par}$
  - $Time_{seq}$ is time for Fibonacci with serial elision
  - $Time_{seq}$ is for the corresponding parallel version, but by using a single thread (sequential execution)
- Platform details
  - AMD EPYC 7551 32-core processor
  - Ubuntu 18.04.3 LTS
  - GCC version 7.5.0
    - -O3 flag used
  - Boost version 1_80_0
  - Argobots commit id dce6e72

# Steal to Task Ratio in Fibonacci

```
uint64_t fib(uint64_t n) {
  if (n < 2) {
    return n;
  } else {
    uint64_t x = fib(n-1); // Spawned as async task
    uint64_t y = fib(n-2); // Called sequentially
    return (x + y);
  }
}
```




- Graph shows the ratio of total tasks stolen to total tasks created while executing Fibonacci(30) at different thread counts (16, 24, and 32)
  - Using HClib implementation of Fib that spawns task for every fib(n-1) recursive call until n<2
- We can observe the steal ratio is extremely low
  - Implies that most of the tasks created by a victim is consumed by itself
- Platform details
  - AMD EPYC 7551 32-core processor
  - Ubuntu 18.04.3 LTS
  - GCC version 7.5.0
    - -O3 flag used

# Why Overheads?

- Creating an async is not same as executing it sequentially
  - Each async has some metadata associated with it
  - Coping user lambda on heap so that it can be used later even if the function that created that task has gone out of scope
- Deque operations are costly*
  - For implementing any thread-safe (concurrent) data structure we always have to use some sort of mutual exclusion that avoids the race condition
    - Imagine using an integer counter that is private to a thread v/s using an integer counter that is to be updated concurrently by several threads

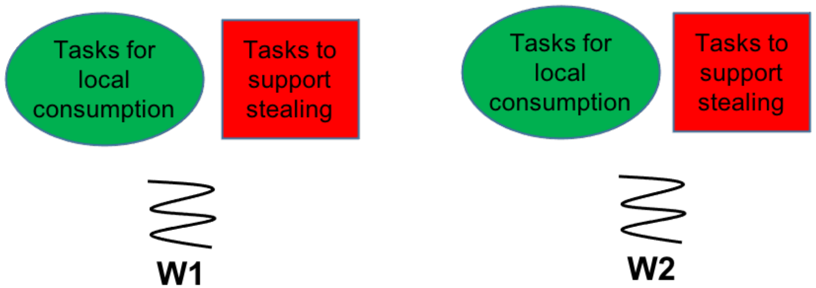
* *The exact costly operation is executing the memory fences, but let's avoid this discussion for now. We will discuss memory fences during lectures on memory consistency*

# Today's Class

- Mixing blocking tasks with async tasks
- Sequential overheads
    - Alternative deques

# Reducing Concurrent Access: General Idea

- Steals are rare
  - Majority of the tasks produced by the victim are consumed by itself
- Recall, deques are concurrent data structure, hence to reduce the overheads, each victim should minimize accessing its “concurrent” deque for push/pop
  - Then where to store async tasks at victims?
    - Use a mix of private and shared task pools
      - Push/pop from private pool, but ensure task(s) availability in shared pool to support stealing

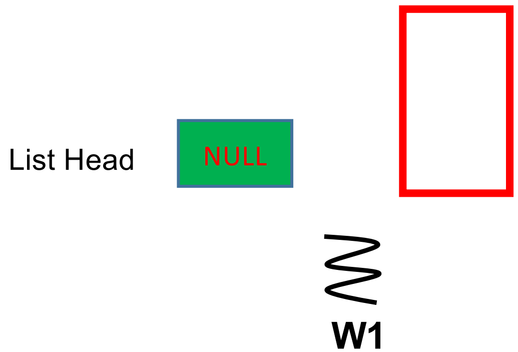


# Reducing Concurrent Access: Using List & Deque

- Each worker uses a private linked list and a concurrent deque
- Victim ensures there are some minimum number of tasks always available in concurrent deque to support steals
- If there are sufficient tasks available in concurrent deque then victim always push/pop from its private list
- Victim checks total tasks on its deque during each push and pop operations
- Thief always steal from the deque as it was doing in default case

```
uint64_t fib(uint64_t n) {
  if(n<2) {
    return n;
  } else {
    std::future<uint64_t> f = std::async([=](){
                                  return fib(n-1);
                                });
    int y = fib(n-2);
    return f.get() + y;
  }
}
```

fib(40)

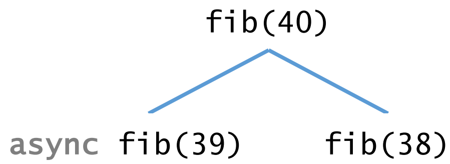


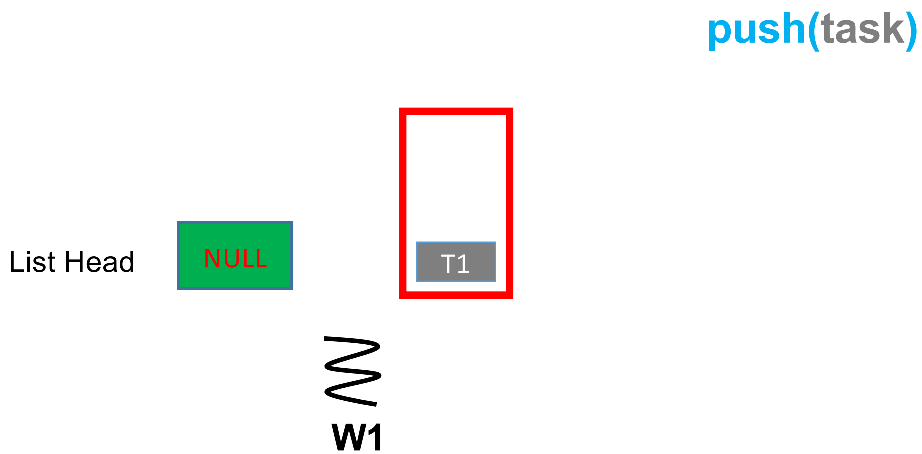

Paper based on a similar idea: https://terpconnect.umd.edu/~barua/ppopp164.pdf

# Reducing Concurrent Access: Using List & Deque

- Each worker uses a private linked list and a concurrent deque
- Victim ensures there are some minimum number of tasks always available in concurrent deque to support steals
- If there are sufficient tasks available in concurrent deque then victim always push/pop from its private list
- Victim checks total tasks on its deque during each push and pop operations
- Thief always steal from the deque as it was doing in default case

```
uint64_t fib(uint64_t n) {
  if(n<2) {
    return n;
  } else {
    std::future<uint64_t> f = std::async([=](){
                                 return fib(n-1);
                              });
    int y = fib(n-2);
    return f.get() + y;
  }
}
```



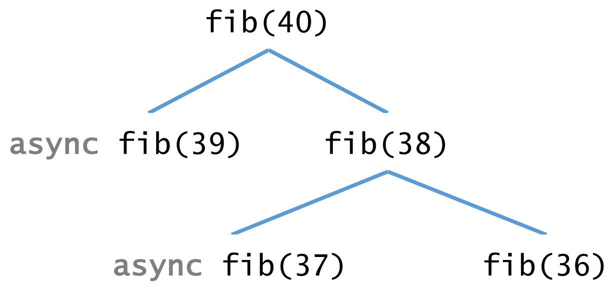




Paper based on a similar idea: https://terpconnect.umd.edu/~barua/ppopp164.pdf

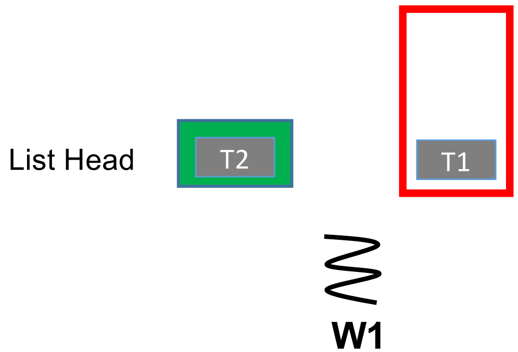
# Reducing Concurrent Access: Using List & Deque

- Each worker uses a private linked list and a concurrent deque
- Victim ensures there are some minimum number of tasks always available in concurrent deque to support steals
- If there are sufficient tasks available in concurrent deque then victim always push/pop from its private list
- Victim checks total tasks on its deque during each push and pop operations
- Thief always steal from the deque as it was doing in default case

```
uint64_t fib(uint64_t n) {
  if(n<2) {
    return n;
  } else {
    std::future<uint64_t> f = std::async([=](){
                                    return fib(n-1);
                                 });
    int y = fib(n-2);
    return f.get() + y;
  }
}
```




push(task)



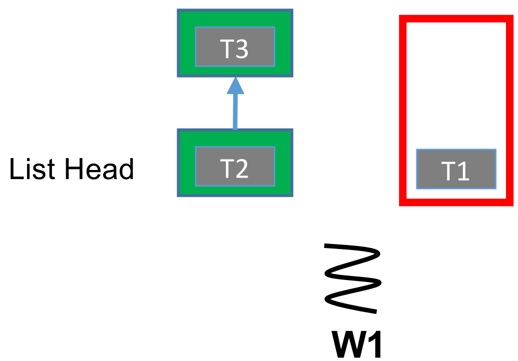

Paper based on a similar idea: https://terpconnect.umd.edu/~barua/ppopp164.pdf

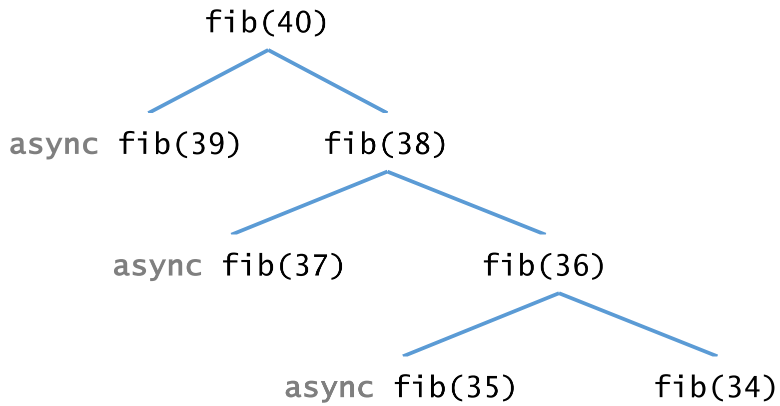
# Reducing Concurrent Access: Using List & Deque

- Each worker uses a private linked list and a concurrent deque
- Victim ensures there are some minimum number of tasks always available in concurrent deque to support steals
- If there are sufficient tasks available in concurrent deque then victim always push/pop from its private list
- Victim checks total tasks on its deque during each push and pop operations
- Thief always steal from the deque as it was doing in default case

```
uint64_t fib(uint64_t n) {
  if(n<2) {
    return n;
  } else {
    std::future<uint64_t> f = std::async([=](){
                                return fib(n-1);
                              });
    int y = fib(n-2);
    return f.get() + y;
  }
}
```

push(task)



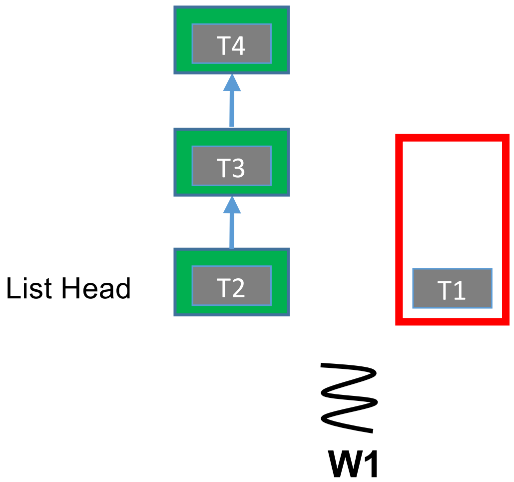




Paper based on a similar idea: https://terpconnect.umd.edu/~barua/ppopp164.pdf

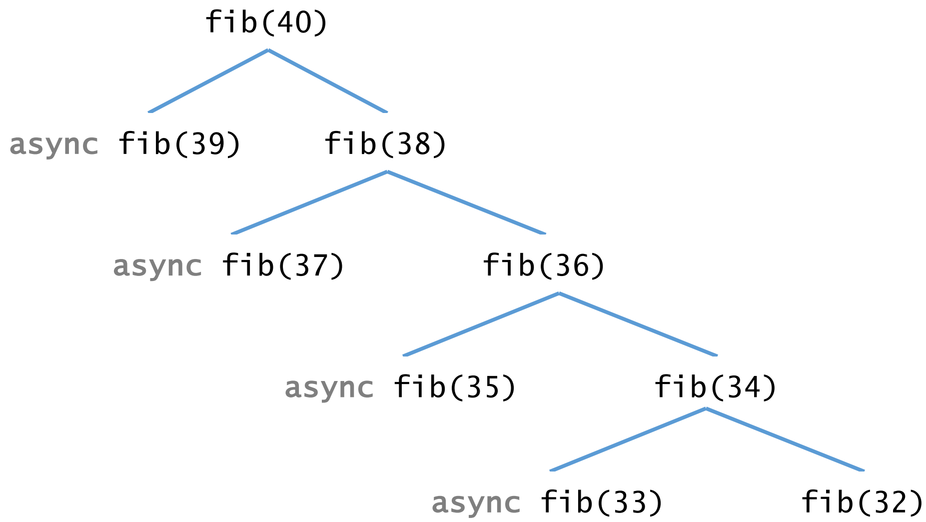
# Reducing Concurrent Access: Using List & Deque

- Each worker uses a private linked list and a concurrent deque
- Victim ensures there are some minimum number of tasks always available in concurrent deque to support steals
- If there are sufficient tasks available in concurrent deque then victim always push/pop from its private list
- Victim checks total tasks on its deque during each push and pop operations
- Thief always steal from the deque as it was doing in default case

```
uint64_t fib(uint64_t n) {
  if(n<2) {
    return n;
  } else {
    std::future<uint64_t> f = std::async([=](){
                                return fib(n-1);
                             });
    int y = fib(n-2);
    return f.get() + y;
  }
}
```




push(task)

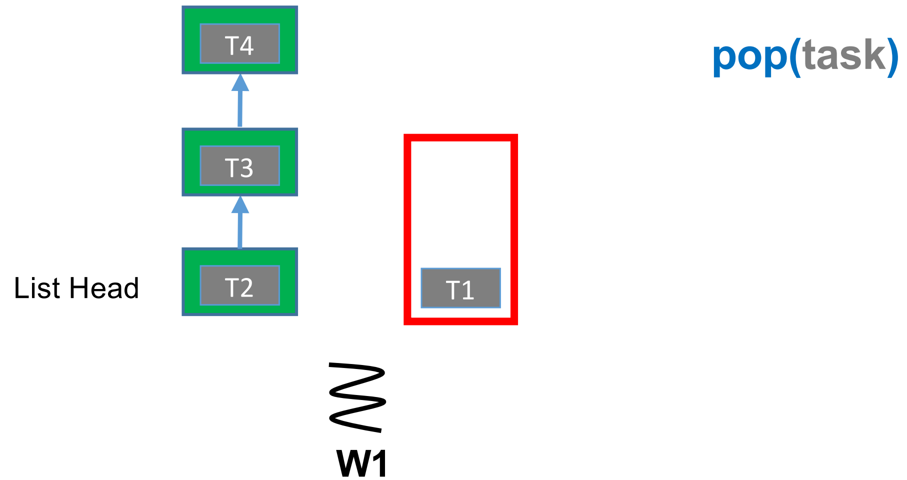


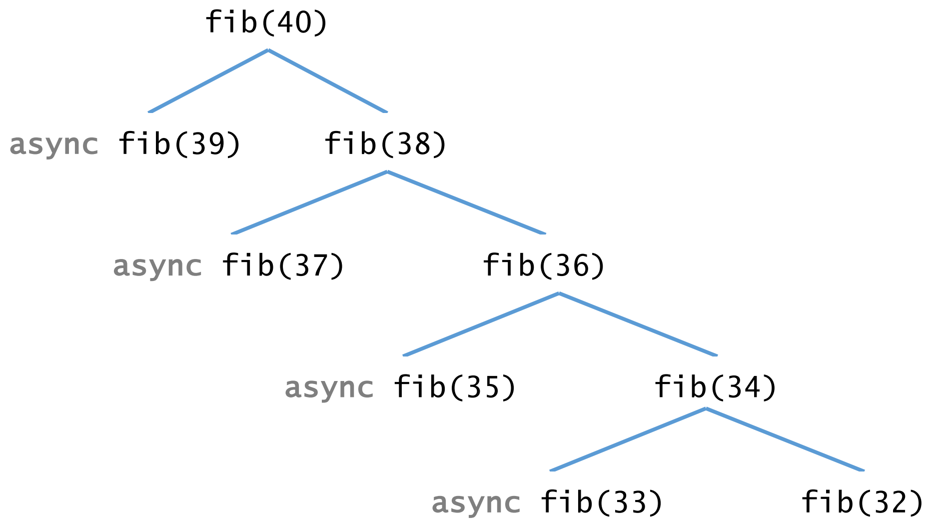

Paper based on a similar idea: https://terpconnect.umd.edu/~barua/ppopp164.pdf

# Reducing Concurrent Access: Using List & Deque

- Each worker uses a private linked list and a concurrent deque
- Victim ensures there are some minimum number of tasks always available in concurrent deque to support steals
- If there are sufficient tasks available in concurrent deque then victim always push/pop from its private list
- Victim checks total tasks on its deque during each push and pop operations
- Thief always steal from the deque as it was doing in default case

```
uint64_t fib(uint64_t n) {
  if(n<2) {
    return n;
  } else {
    std::future<uint64_t> f = std::async([=](){
                                  return fib(n-1);
                              });
    int y = fib(n-2);
    return f.get() + y;
  }
}
```



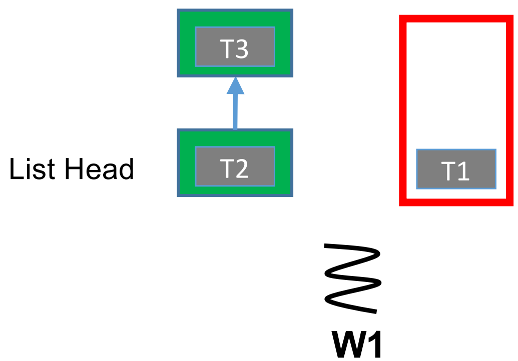

Paper based on a similar idea: https://terpconnect.umd.edu/~barua/ppopp164.pdf

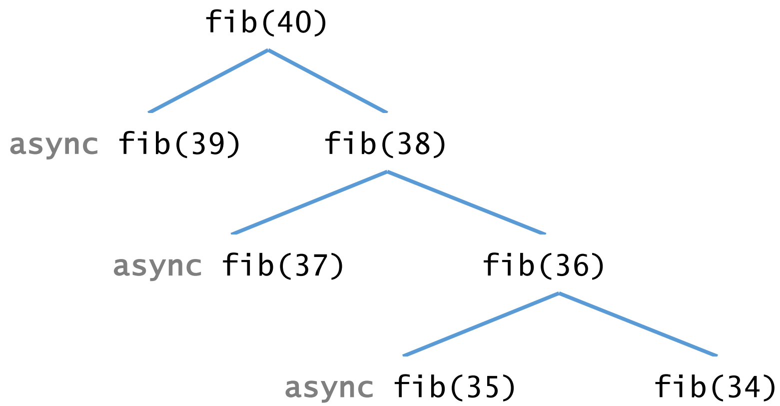
# Reducing Concurrent Access: Using List & Deque

- Each worker uses a private linked list and a concurrent deque
- Victim ensures there are some minimum number of tasks always available in concurrent deque to support steals
- If there are sufficient tasks available in concurrent deque then victim always push/pop from its private list
- Victim checks total tasks on its deque during each push and pop operations
- Thief always steal from the deque as it was doing in default case

```
uint64_t fib(uint64_t n) {
  if(n<2) {
    return n;
  } else {
    std::future<uint64_t> f = std::async([=](){
                                 return fib(n-1);
                              });
    int y = fib(n-2);
    return f.get() + y;
  }
}
```

pop(task)



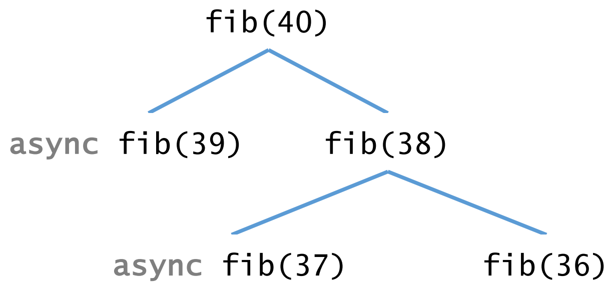




Paper based on a similar idea: https://terpconnect.umd.edu/~barua/ppopp164.pdf

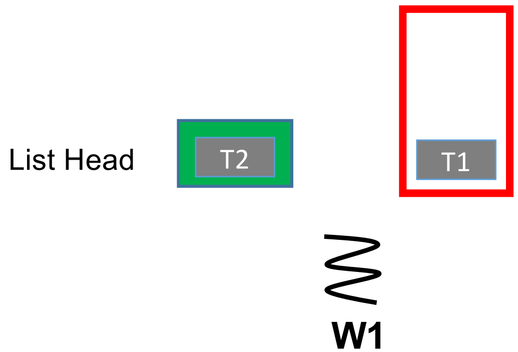
# Reducing Concurrent Access: Using List & Deque

- Each worker uses a private linked list and a concurrent deque
- Victim ensures there are some minimum number of tasks always available in concurrent deque to support steals
- If there are sufficient tasks available in concurrent deque then victim always push/pop from its private list
- Victim checks total tasks on its deque during each push and pop operations
- Thief always steal from the deque as it was doing in default case

```
uint64_t fib(uint64_t n) {
  if(n<2) {
    return n;
  } else {
    std::future<uint64_t> f = std::async([=](){
                                  return fib(n-1);
                               });
    int y = fib(n-2);
    return f.get() + y;
  }
}
```




pop(task)

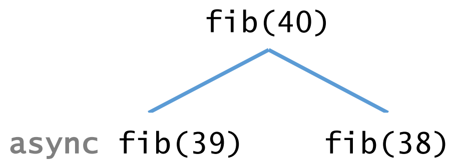



Paper based on a similar idea: https://terpconnect.umd.edu/~barua/ppopp164.pdf

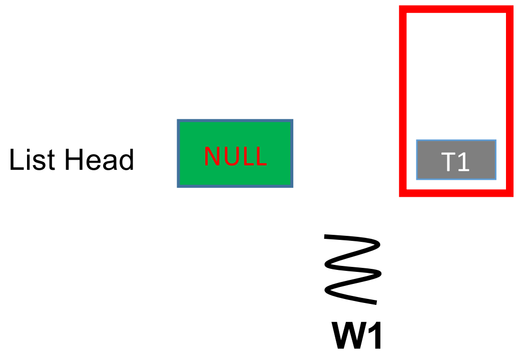
# Reducing Concurrent Access: Using List & Deque

- Each worker uses a private linked list and a concurrent deque
- Victim ensures there are some minimum number of tasks always available in concurrent deque to support steals
- If there are sufficient tasks available in concurrent deque then victim always push/pop from its private list
- Victim checks total tasks on its deque during each push and pop operations
- Thief always steal from the deque as it was doing in default case

```
uint64_t fib(uint64_t n) {
  if(n<2) {
    return n;
  } else {
    std::future<uint64_t> f = std::async([=](){
                                return fib(n-1);
                              });
    int y = fib(n-2);
    return f.get() + y;
  }
}
```




**pop(task)**




Paper based on a similar idea: https://terpconnect.umd.edu/~barua/ppopp164.pdf

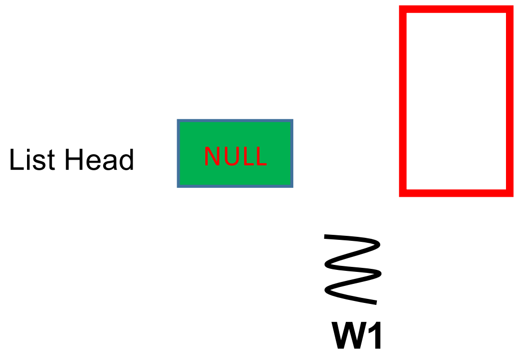
# Reducing Concurrent Access: Using List & Deque

- Each worker uses a private linked list and a concurrent deque
- Victim ensures there are some minimum number of tasks always available in concurrent deque to support steals
- If there are sufficient tasks available in concurrent deque then victim always push/pop from its private list
- Victim checks total tasks on its deque during each push and pop operations
- Thief always steal from the deque as it was doing in default case

```
uint64_t fib(uint64_t n) {
  if(n<2) {
    return n;
  } else {
    std::future<uint64_t> f = std::async([=](){
                                 return fib(n-1);
                              });
    int y = fib(n-2);
    return f.get() + y;
  }
}
```

fib(40)

pop(task)




Paper based on a similar idea: https://terpconnect.umd.edu/~barua/ppopp164.pdf

# Reducing Concurrent Access: Using List & Deque

```
Task* pop() {
  Task* t = NULL;
  if(current_worker->Tail != NULL) {
    t = current_worker->pop_from_list_tail();
    move_task_from_list_to_deque();
  } else {
    t = current_worker->deque_pop();
  }
  return t;
}
```

```
bool move_task_from_list_to_deque() {
  Task* t = pop_from_list_head();
  if(t) {
     current_worker->deque_push(t);
  } else {
    return false;
  }
}
```

Popping items from Head for adding into deque has some benefits with recursive task creation? Why?

```
#define DEQUE_LIMIT /* Some value */

struct Node {
  User_Lambda task;
  Node* next;
}

Node *Head, *Tail; /* Thread local */

void push(T lambda) {
  bool success = true;
  /* Add task to my deque if required */
  if(current_worker->deque_size < DEQUE_LIMIT) {
    success = move_task_from_list_to_deque();
  }
  if(!success) current_worker->deque_push(lambda);
  else current_worker->push_to_list_tail(lambda);
}
```

Paper based on a similar idea: [https://terpconnect.umd.edu/~barua/ppopp164.pdf](https://terpconnect.umd.edu/~barua/ppopp164.pdf)

# Reducing Concurrent Access: Using List & Deque

- Issues
  - Doesn't support stealing more than one tasks at a time
    - Stealing more than one task can reduce the steal frequency
  - Maintaining a linked list means more mallocs/frees for adding/removing nodes
    - Tasks are anyway copied on heap

# Reading Materials

- Handling blocking IO asynchronously
  - *https://www.cse.wustl.edu/~angelee/home_page/papers/futureIO.pdf*
- Using list and deques together
  - https://terpconnect.umd.edu/~barua/ppopp164.pdf
- You may only read the implementation section and skip theorem/proofs (if any)

# Next Lecture (L #07)

- Managing concurrent deque overheads (contd.)
- Runtime techniques for controlling task granularity
- Quiz-1
  - Syllabus: L#02 - L#04
  - During lecture hours