Lecture 07: Sequential Overheads from Concurrent Deque

Vivek Kumar

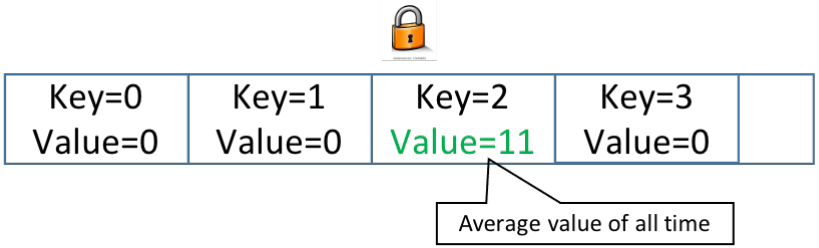
Computer Science and Engineering

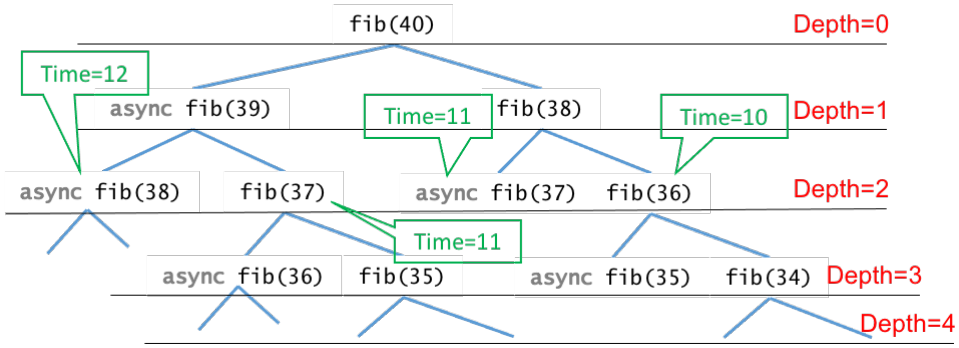
IIIT Delhi

vivekk@iiitd.ac.in

Last Lecture (Recap)

- Sequential overheads from fine granular task creation
  - Tasks near the bottom of tree are smaller computations
  - Deep procedure calling stack in thread due to recursion
- Automatically controlling task granularity in recursive task decomposition
  - Assumption is that the tree (computation graph) is well balanced
    - Dynamic task aggregation
    - Each task records its depth and the execution time at that depth
    - Above information is used to decide if any more tasks at certain depth has to be created or should be executed serially



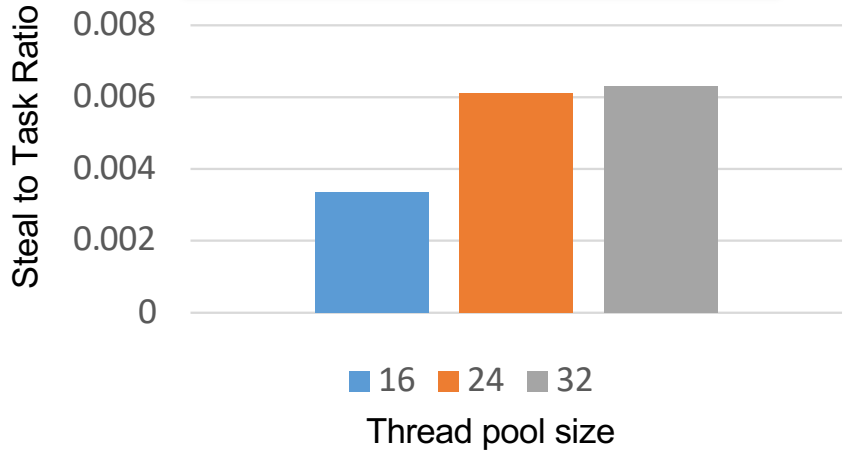


Today's Class

- Minimizing overheads from deque operations
  - Using a mix of list and deque
  - Using private deques

Sequential Overheads (1/2)

```
uint64_t fib(uint64_t n) {
  if (n < 2) {
    return n;
  } else {
    uint64_t x, y;
    finish([&]() {
      async([&x]() { x = fib(n-1); }
      y = fib(n-2);
    }
    return (x + y);
  }
}
```



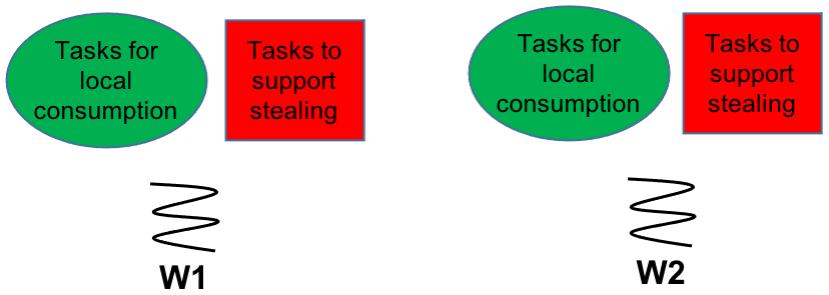
- Graph shows the ratio of total tasks stolen to total tasks created while executing Fibonacci(30) on a 32-core processor at different thread counts (16, 24, and 32)
  - Using HClib implementation of Fib that spawns task for every fib(n-1) recursive call until n<2
- We can observe the steal ratio is extremely low
  - Implies that most of the tasks created by a victim is consumed by itself

Sequential Overheads (2/2)

- 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
  - **It is important to control task granularity**
- 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
  - **It is important to use an efficient deque implementation**

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

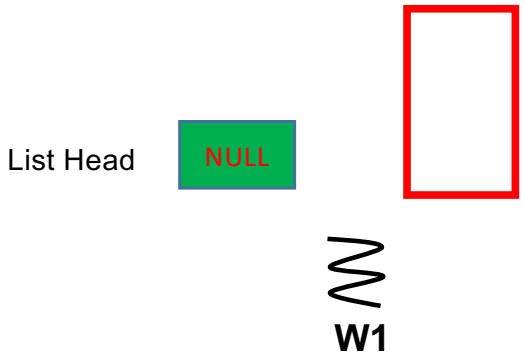


Reducing Concurrent Access: Using List & Deque

- Each worker uses a private linked list and a concurrent deque

```
uint64_t fib(uint64_t n) {
  if (n < 2) {
    return n;
  } else {
    uint64_t x, y;
    finish([&]() {
      async([&x]() { x = fib(n-1); }
      y = fib(n-2);
    }
    return (x + y);
  }
}
```

fib(40)

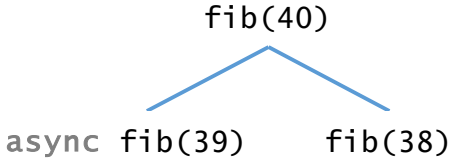


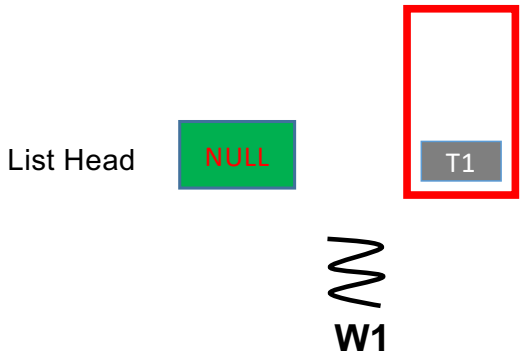
Paper based on a similar idea: https://terpconnect.umd.edu/~barua/ppopp164.pdf

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

```
uint64_t fib(uint64_t n) {
  if (n < 2) {
    return n;
  } else {
    uint64_t x, y;
    finish([&]() {
      async([&x]() { x = fib(n-1); }
      y = fib(n-2);
    }
    return (x + y);
  }
}
```



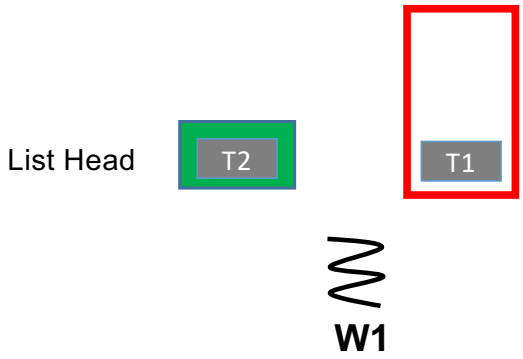


Paper based on a similar idea: https://terpconnect.umd.edu/~barua/ppopp164.pdf

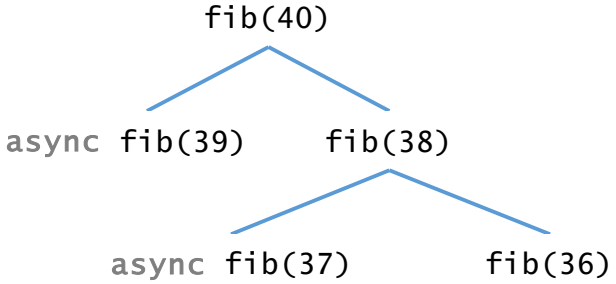
Reducing Concurrent Access: Using List & Deque

- 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

```
uint64_t fib(uint64_t n) {
  if (n < 2) {
    return n;
  } else {
    uint64_t x, y;
    finish([&]() {
      async([&x]() { x = fib(n-1); }
      y = fib(n-2);
    }
    return (x + y);
  }
}
```



push(task)

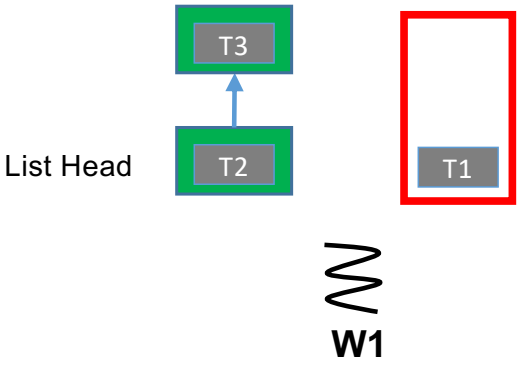


Paper based on a similar idea: https://terpconnect.umd.edu/~barua/ppopp164.pdf

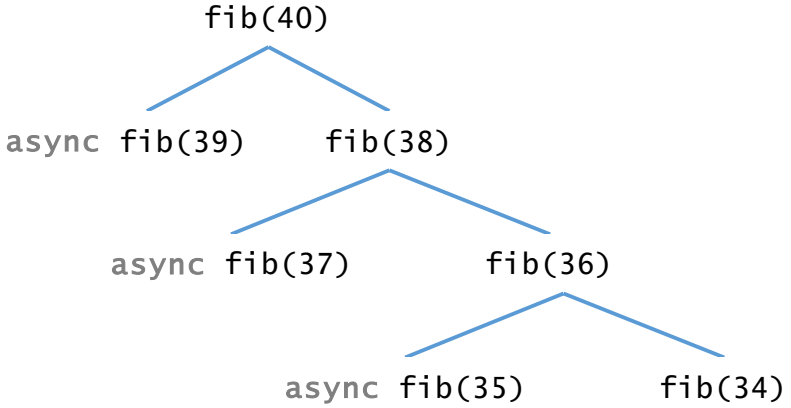
Reducing Concurrent Access: Using List & Deque

- 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

```
uint64_t fib(uint64_t n) {
  if (n < 2) {
    return n;
  } else {
    uint64_t x, y;
    finish([&]() {
      async([&x]() { x = fib(n-1); }
      y = fib(n-2);
    }
    return (x + y);
  }
}
```



push(task)

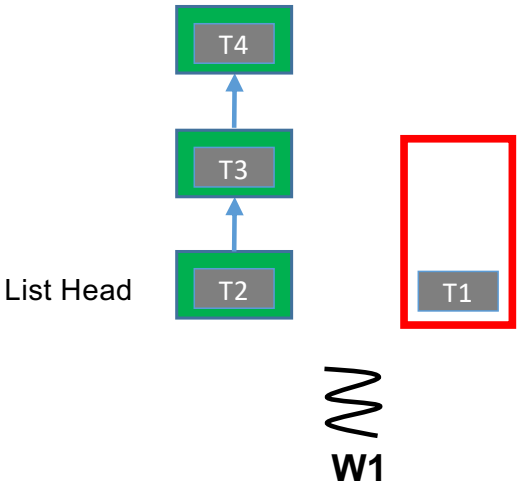


Paper based on a similar idea: https://terpconnect.umd.edu/~barua/ppopp164.pdf

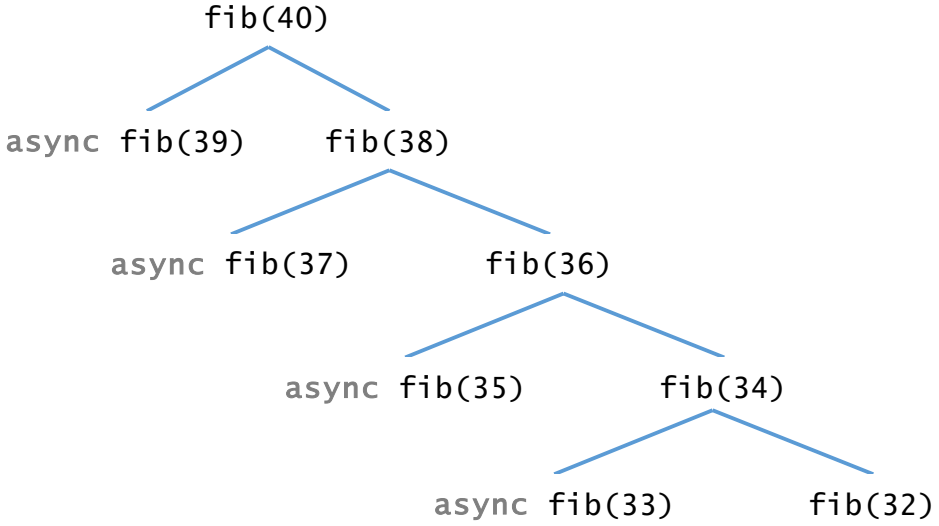
Reducing Concurrent Access: Using List & Deque

- 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

```
uint64_t fib(uint64_t n) {
  if (n < 2) {
    return n;
  } else {
    uint64_t x, y;
    finish([&]() {
      async([&x]() { x = fib(n-1); }
      y = fib(n-2);
    }
    return (x + y);
  }
}
```



push(task)

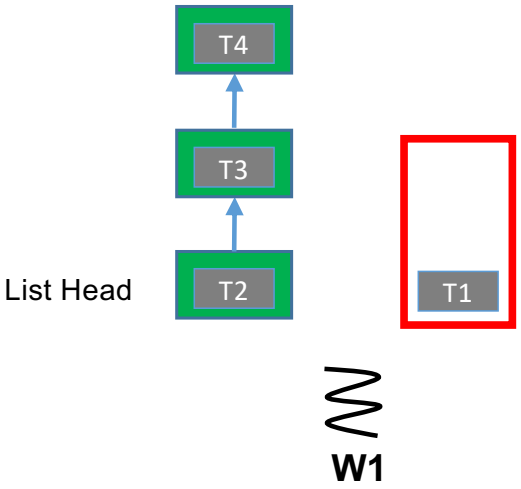


Paper based on a similar idea: https://terpconnect.umd.edu/~barua/ppopp164.pdf

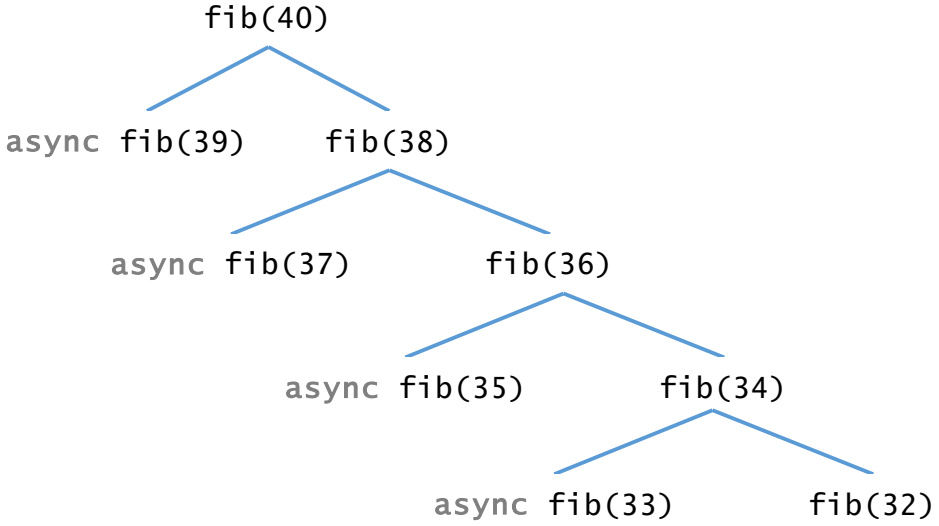
Reducing Concurrent Access: Using List & Deque

- 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

```
uint64_t fib(uint64_t n) {
  if (n < 2) {
    return n;
  } else {
    uint64_t x, y;
    finish([&]() {
      async([&x]() { x = fib(n-1); }
      y = fib(n-2);
    }
    return (x + y);
  }
}
```



pop(task)

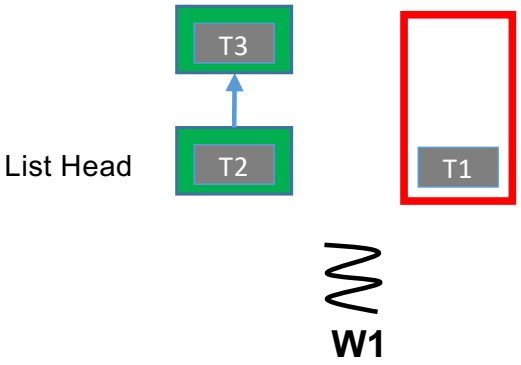


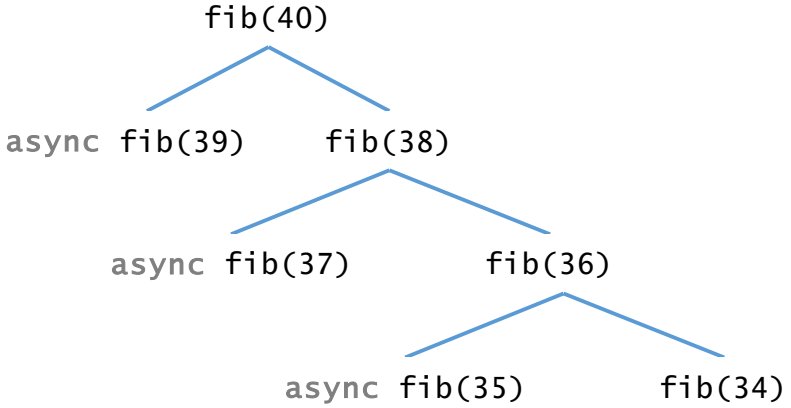
Paper based on a similar idea: https://terpconnect.umd.edu/~barua/ppopp164.pdf

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

```
uint64_t fib(uint64_t n) {
  if (n < 2) {
    return n;
  } else {
    uint64_t x, y;
    finish([&]() {
      async([&x]() { x = fib(n-1); }
      y = fib(n-2);
    }
    return (x + y);
  }
}
```



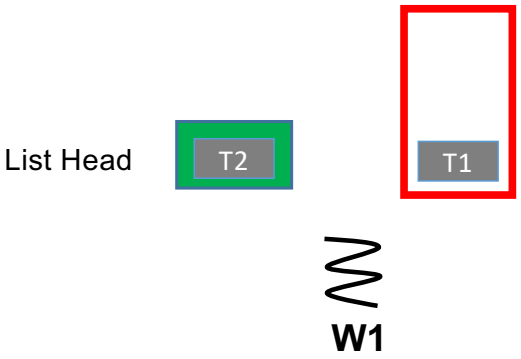


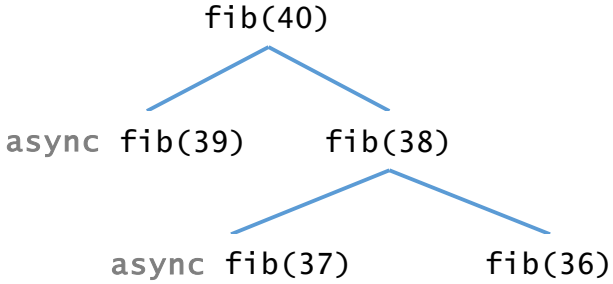
Paper based on a similar idea: https://terpconnect.umd.edu/~barua/ppopp164.pdf

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

```
uint64_t fib(uint64_t n) {
  if (n < 2) {
    return n;
  } else {
    uint64_t x, y;
    finish([&]() {
      async([&x]() { x = fib(n-1); }
      y = fib(n-2);
    }
    return (x + y);
  }
}
```



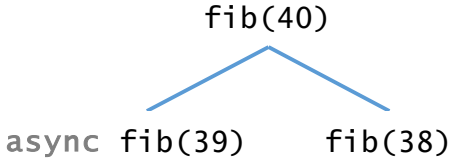


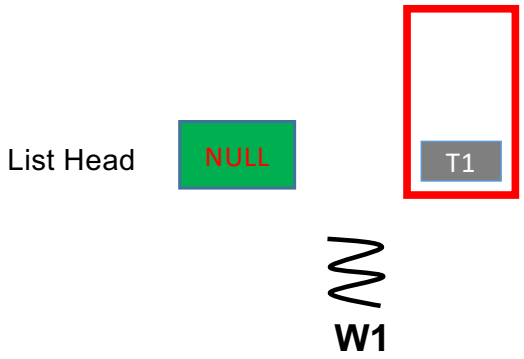
Paper based on a similar idea: https://terpconnect.umd.edu/~barua/ppopp164.pdf

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 {
    uint64_t x, y;
    finish([&]() {
      async([&x]() { x = fib(n-1); }
      y = fib(n-2);
    }
    return (x + y);
  }
}
```





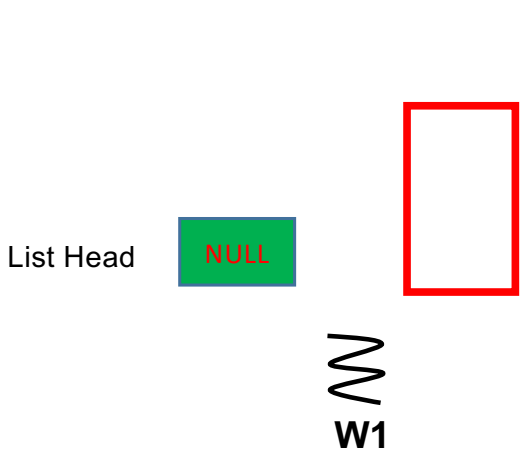
Paper based on a similar idea: https://terpconnect.umd.edu/~barua/ppopp164.pdf

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 {
    uint64_t x, y;
    finish([&]() {
      async([&x]() { x = fib(n-1); }
      y = fib(n-2);
    }
    return (x + y);
  }
}
```

fib(40)



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();
    if(current_worker->deque_size < DEQUE_LIMIT) {
      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);
     return true;
  } 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

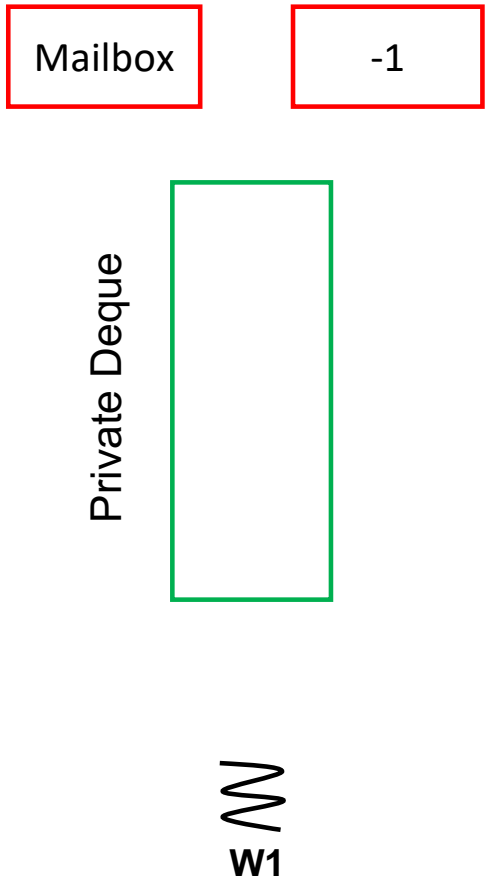
Reducing Concurrent Access: Using List & Deque

- 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

Today's Class

- Minimizing overheads from deque operations
  - Using a mix of list and deque
  - Using private deques

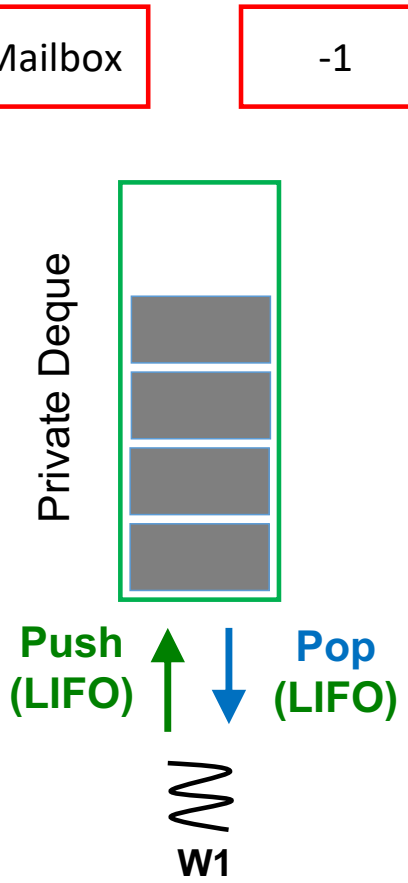
Reducing Concurrent Access: Using Private Deque



- Every worker maintains three data structures
  - A non-concurrent private deque
    - Same as the default deque, but without the support for concurrent (thread-safe) accesses
  - One mailbox
    - That can store one or more tasks
    - Contains a counter indicating total number of stored tasks
  - One shared counter

Paper: https://hal.inria.fr/hal-00863028/document

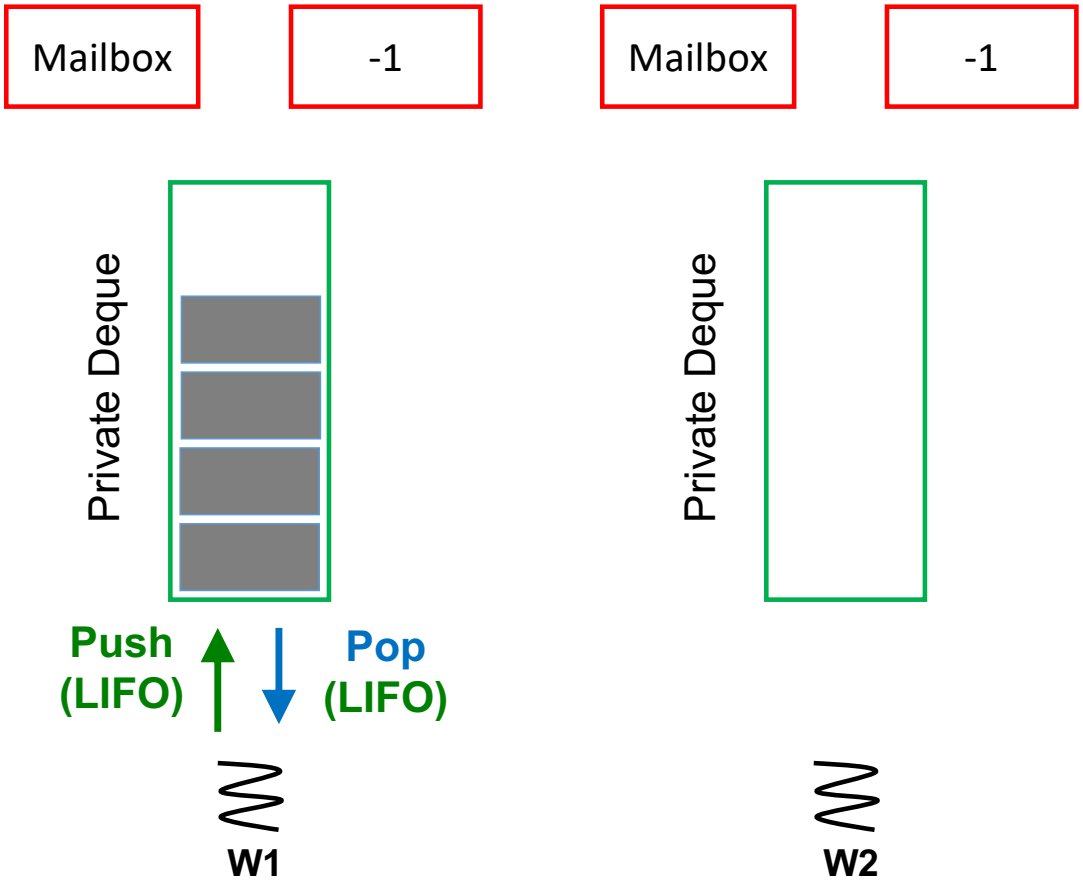
Reducing Deque Access: Using Private Deque



- Victim
  - Push/pop tasks into its private deque

Paper: https://hal.inria.fr/hal-00863028/document

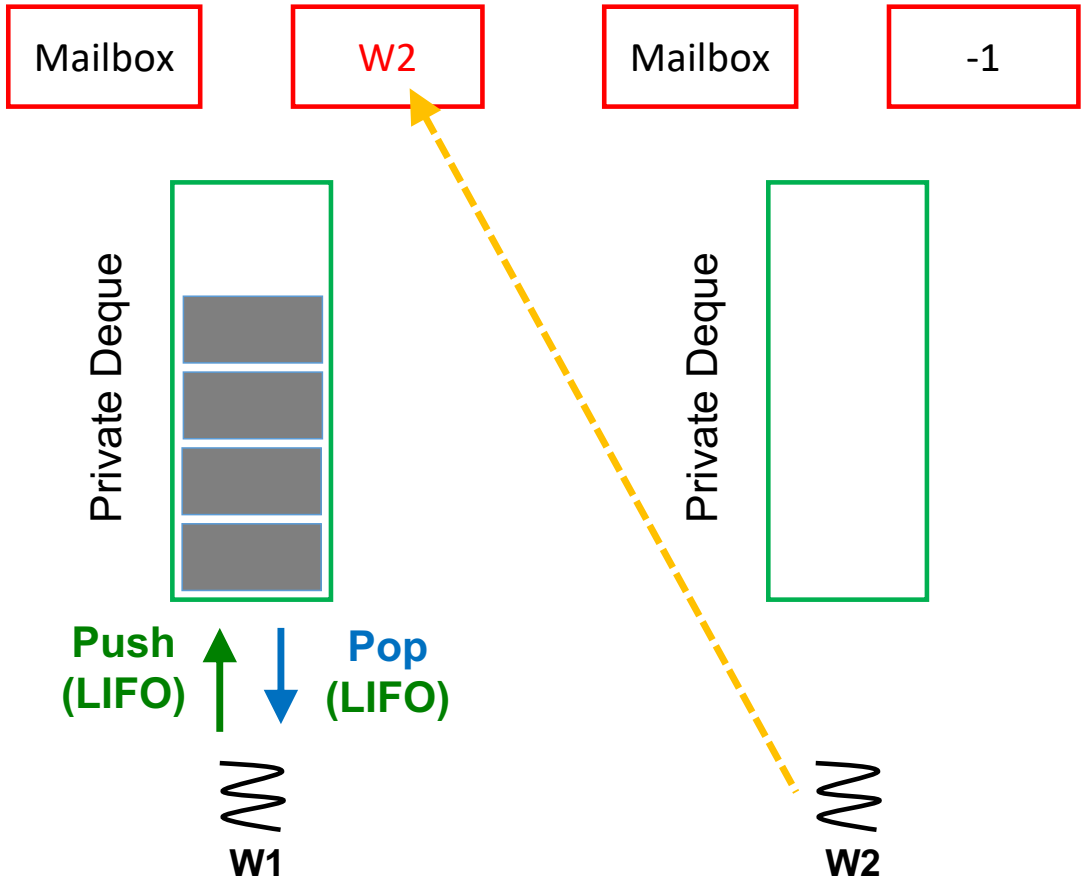
Reducing Deque Access: Using Private Deque



- Thief
  - Selects a random victim (W1) who has items on its deque
  - Checks victim deque size without any locks
    - The usual way of doing (head-tail)

Paper: https://hal.inria.fr/hal-00863028/document

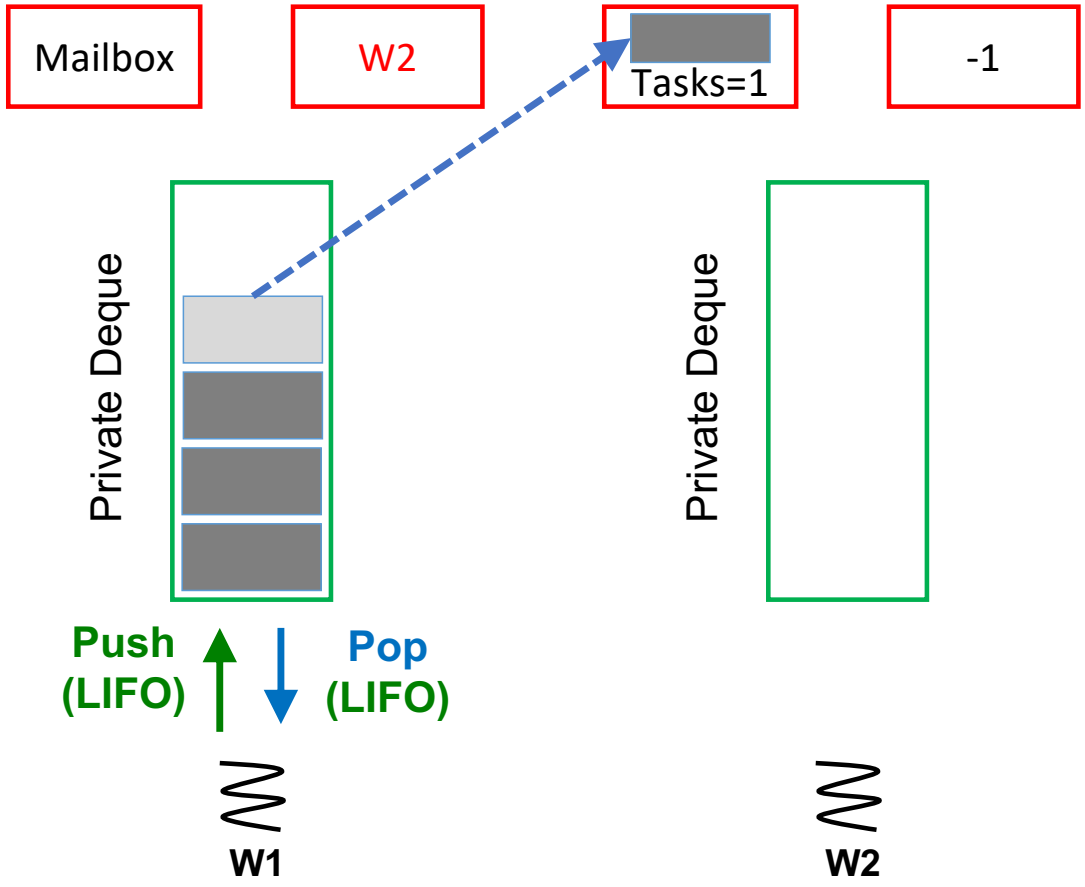
Reducing Deque Access: Using Private Deque



- Thief
  - Record its own id inside the request box at W1 (critical section), and goes inside condition wait
  - Only one thief at a time

Paper: https://hal.inria.fr/hal-00863028/document

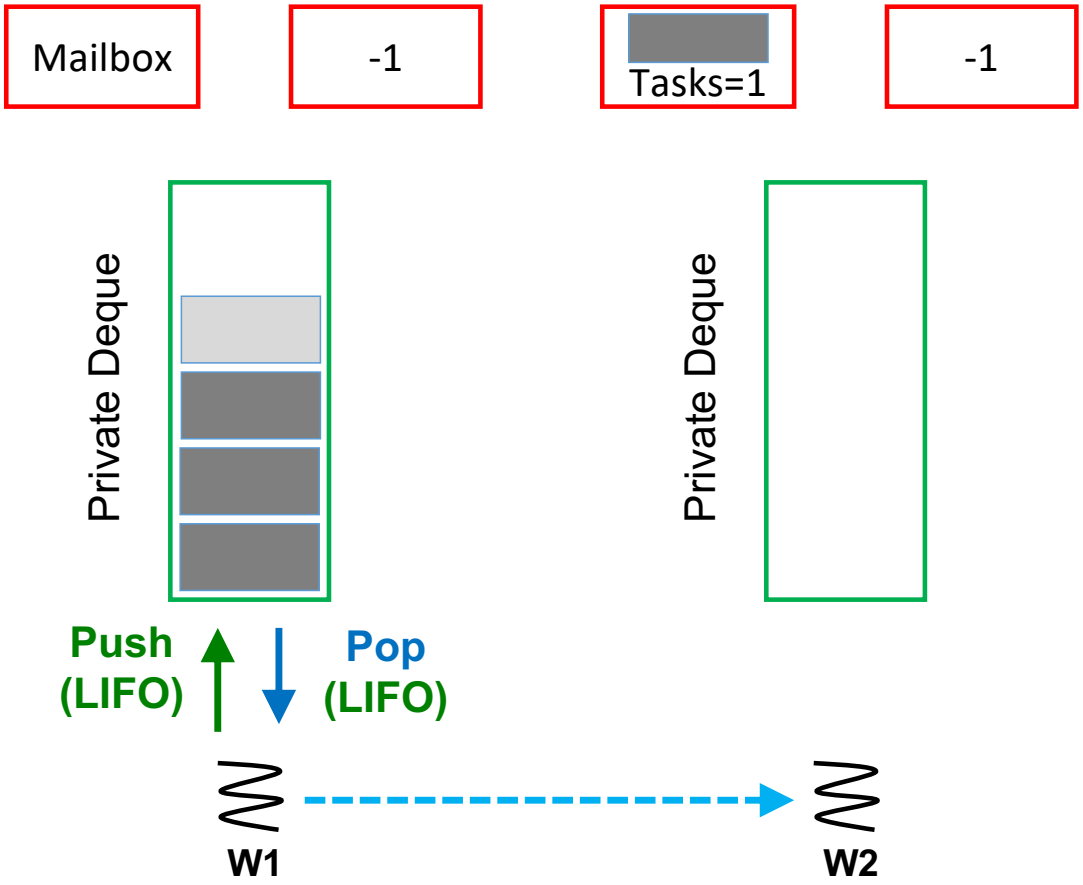
Reducing Deque Access: Using Private Deque



- Victim
  - Check its request box inside each push/pop/steal
  - If tasks are available on victim's private deque
    - Pop item(s) from the head and copies it into the waiting thief's mailbox (W2)
    - Update W2's mailbox with the total number of tasks copied

Paper: https://hal.inria.fr/hal-00863028/document

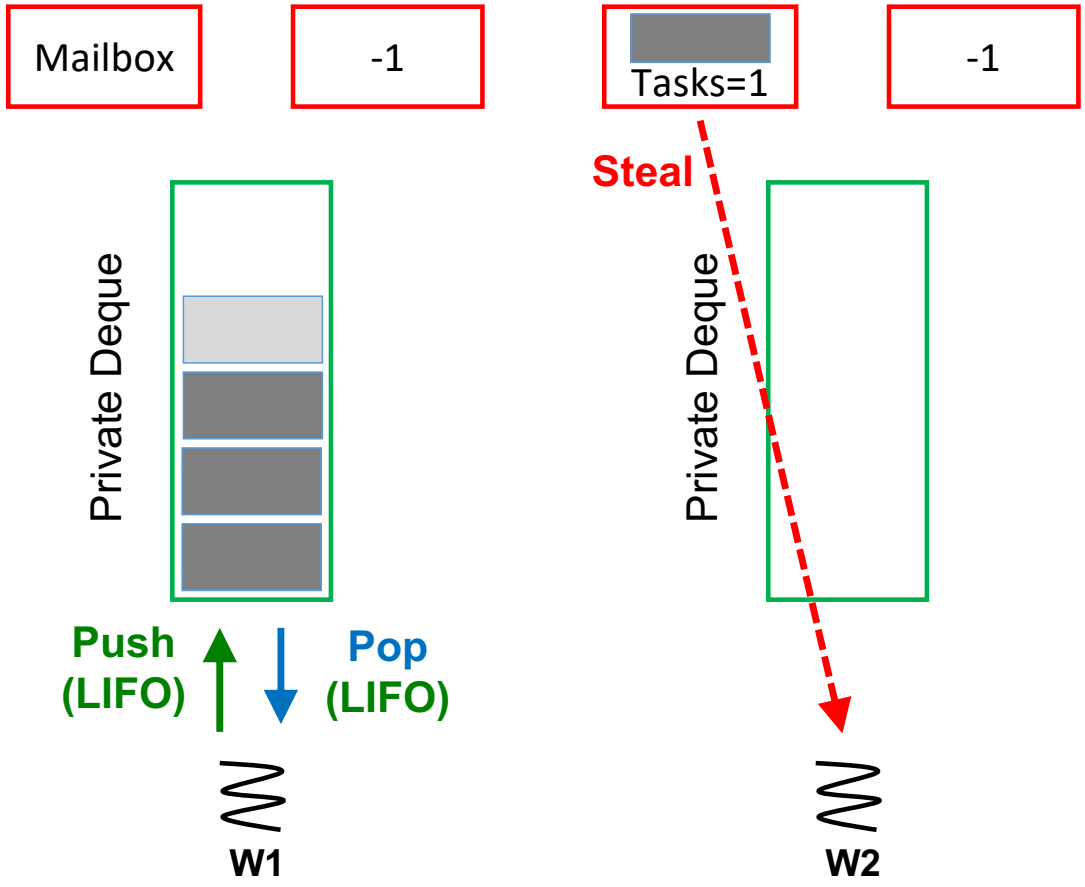
Reducing Deque Access: Using Private Deque



- Victim
  - Clears its request box
  - Signals the waiting thief W2

Paper: https://hal.inria.fr/hal-00863028/document

Reducing Deque Access: Using Private Deque



- Thief
  - Unblocks after being notified by W1
  - Steal tasks from its mailbox and start executing them
    - If more than one task received then extra tasks pushed to its private deque
  - Failed steal attempt if it did not receive any task (i.e., if W1 ran out of tasks)

Paper: https://hal.inria.fr/hal-00863028/document

Reading Materials

- Using list and deques together
  - https://terpconnect.umd.edu/~barua/ppopp164.pdf
- Private deques
  - https://hal.inria.fr/hal-00863028/document
- You may only read the implementation section and skip theorem/proofs (if any)

Next Lecture (L #08)

- Non uniform memory access architectures