Lecture 20:

Scheduling Fork-Join Parallelism

Parallel Computer Architecture and Programming
CMU 15-418/15-618, Spring 2015

Tunes

Robinella

Break it Down Baby

"Just expose independent work as it comes, and let the scheduler do the rest."

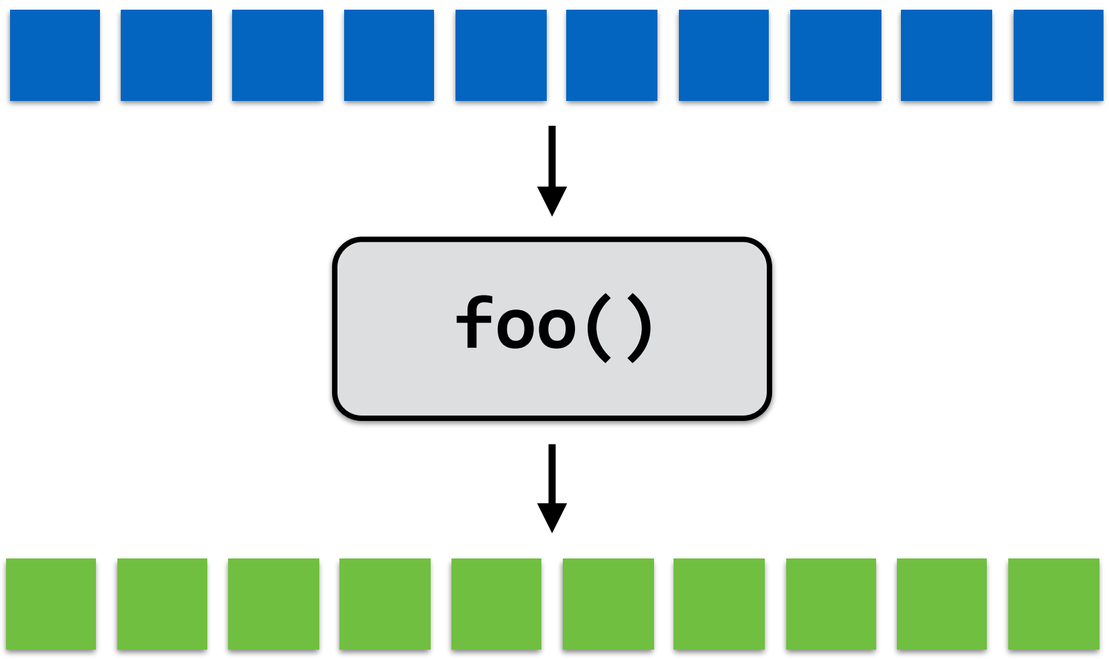
- Robinella

Common parallel programming patterns

Data Parallelism:

Perform same sequence of operations on many data elements

```
//openMP parallel for
#pragma omp parallel for
for (int i=0; i<N; i++) {
   B[i] = foo(A[i]);
}

// CUDA bulk launch
foo<<<numBlocks, threadsPerBlock>>>(A, B);

// ISPC foreach
foreach (i=0 ... N) {
   B[i] = foo(A[i]);
}

// ISPC bulk task launch
launch[numTasks] myFooTask(A, B);

// using higher-order function ‘map’
map(foo, A, B);
```



Common parallel programming patterns

Explicit management of parallelism with threads:

Create one thread per execution unit (or per amount of desired concurrency)

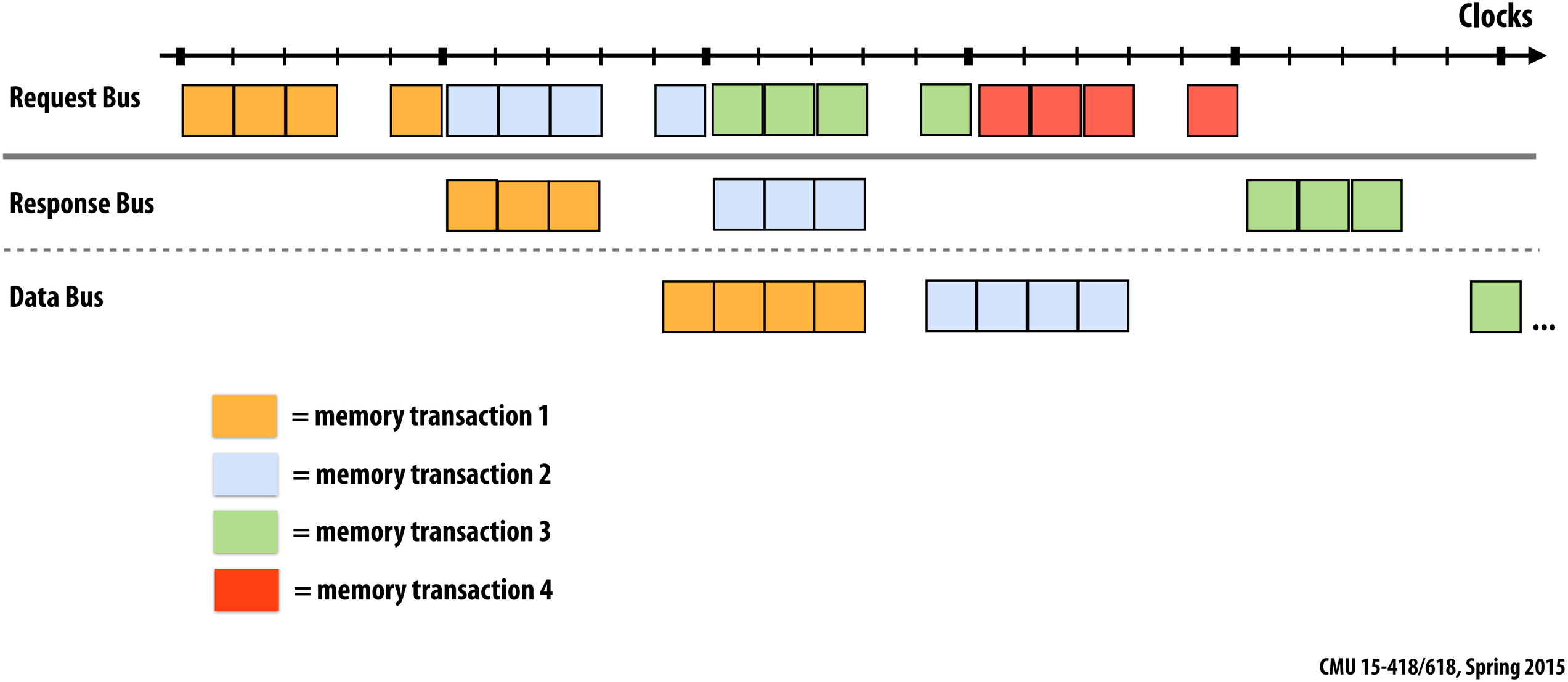
- **Example below: C code with pthreads**
- **Other examples: mpirun -np 4**

```
struct thread_args {
  float* A;
  float* B;
};

int thread_id[MAX_THREADS];

thread_args args;
args.A = A;
args.B = B;

for (int i=0; i<num_cores; i++) {
  pthread_create(&thread_id[i], NULL, myFunctionFoo, &args);
}

for (int i=0; i<num_cores; i++) {
  pthread_join(&thread_id[i]);
}
```

Common parallel programming patterns

Pipeline Parallelism:
Each unit/worker is responsible for one stage of computation on a data element.

Below: three units used by bus transaction: request bus, response bus, data bus
Other examples: processor instruction pipeline, pipelined network transmission, . . .



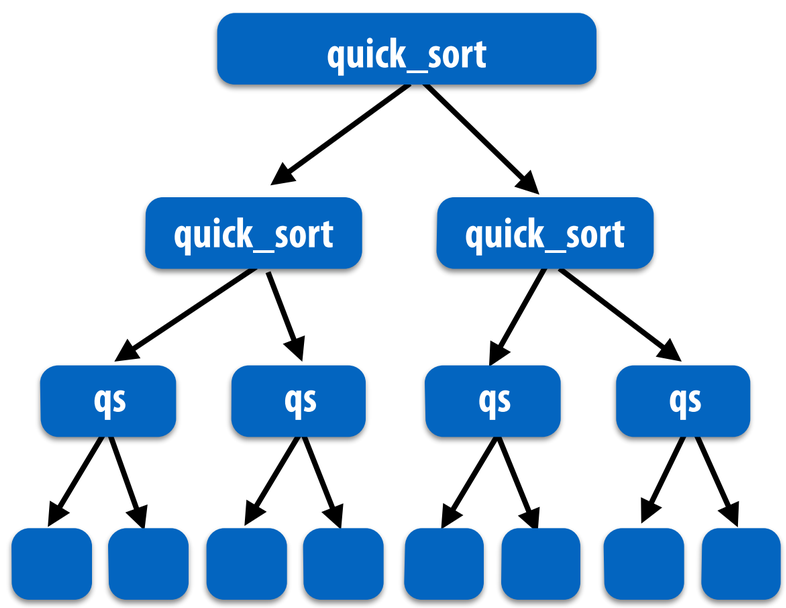
Consider divide-and-conquer algorithms

Quick sort:

```
// sort elements from begin up to (but not including) end
void quick_sort(int* begin, int* end) {

  if (begin >= end-1)
    return;

  else {

     // choose partition key and partition elements
     // by key, return position of key as `middle`
     int* middle = partition(begin, end);

     quick_sort(begin, middle);

     quick_sort(middle+1, last);

  }
}
```

independent work!

Dependencies



Fork-join pattern

- Natural way to express independent work inherent in divide-and-conquer algorithms
- Today's code examples will be in Cilk Plus
  - C++ language extension
  - Originally developed at MIT, now adapted as open standard (in GCC, Intel ICC)

```
cilk_spawn foo(args);
```

← "fork" (create new logical thread of control)

Semantics: invoke `foo`, but unlike standard function call, caller may continue executing asynchronously with execution of `foo`.

```
cilk_sync;
```

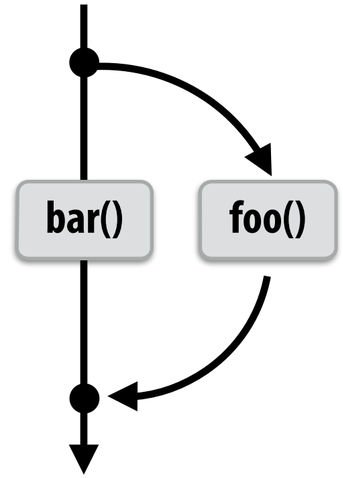
← "join"

Semantics: returns when all calls spawned by current function have completed. ("sync up" with the spawned calls)

Note: there is an implicit `cilk_sync` at the end of every function that contains `cilk_spawn` (implication: when a Cilk function returns, all work associated with that function is complete)

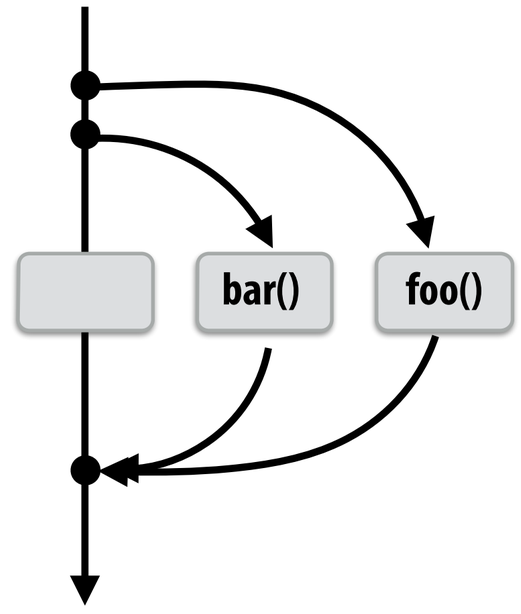
Basic Cilk Plus examples

```
// foo() and bar() may run in parallel
cilk_spawn foo();
bar();
cilk_sync;
```

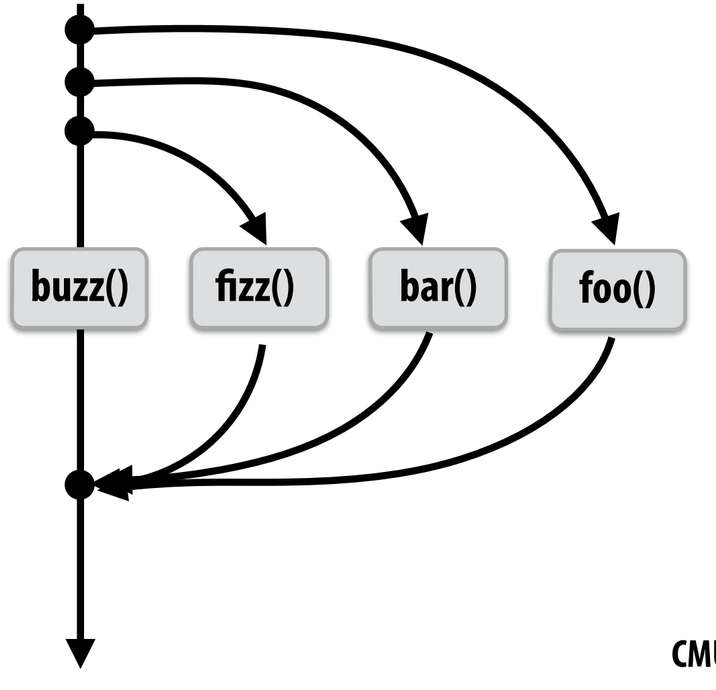



```
// foo() and bar() may run in parallel
cilk_spawn foo();
cilk_spawn bar();
cilk_sync;
```

Same amount of independent work first example, but potentially higher runtime overhead (due to two spawns vs. one)




```
// foo, bar, fizz, buzz, may run in parallel
cilk_spawn foo();
cilk_spawn bar();
cilk_spawn fizz();
buzz();
cilk_sync;
```



Abstraction vs. implementation

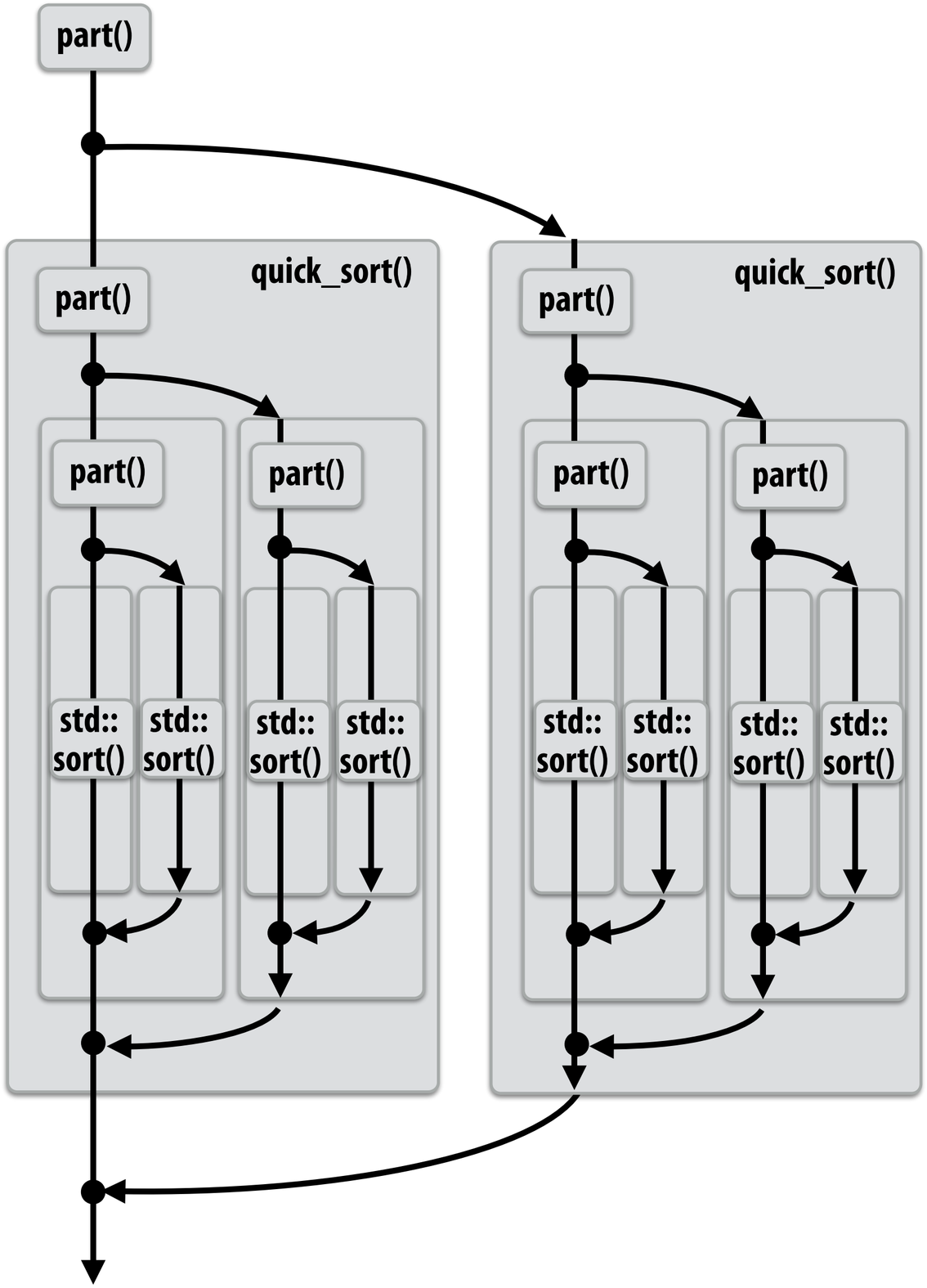
- Notice that the `cilk_spawn` abstraction does not specify how or when spawned calls are scheduled to execute
  - Only that they may be run concurrently with caller (and with all other calls spawned by the caller)

- But `cilk_sync` does serve as a constraint on scheduling
  - All spawned calls must complete before `cilk_sync` returns

Parallel quicksort in Cilk Plus

```
void quick_sort(int* begin, int* end) {
  if (begin >= end - PARALLEL_CUTOFF)
    std::sort(begin, end);
  else {
    int* middle = partition(begin, end);
    cilk_spawn quick_sort(begin, middle);
    quick_sort(middle+1, last);
  }
}
```

Sort sequentially if problem size is sufficiently small (overhead of spawn trumps benefits of potential parallelization)



Writing fork-join programs

- Main idea: expose independent work (potential parallelism) to the system using `cilk_spawn`

- Recall parallel programming rules of thumb
  - Want at least as much work as parallel execution capability (e.g., program should probably spawn at least was much work as there are cores)
  - Want more independent work than execution capability to allow for good workload balance of all the work onto the cores
    - "parallel slack" = ratio of independent work to machine's parallel execution capability (in practice: ~8 is a good ratio)
  - But not too much independent work so that granularity of work is too small (too much slack incurs overhead of managing fine-grained work)

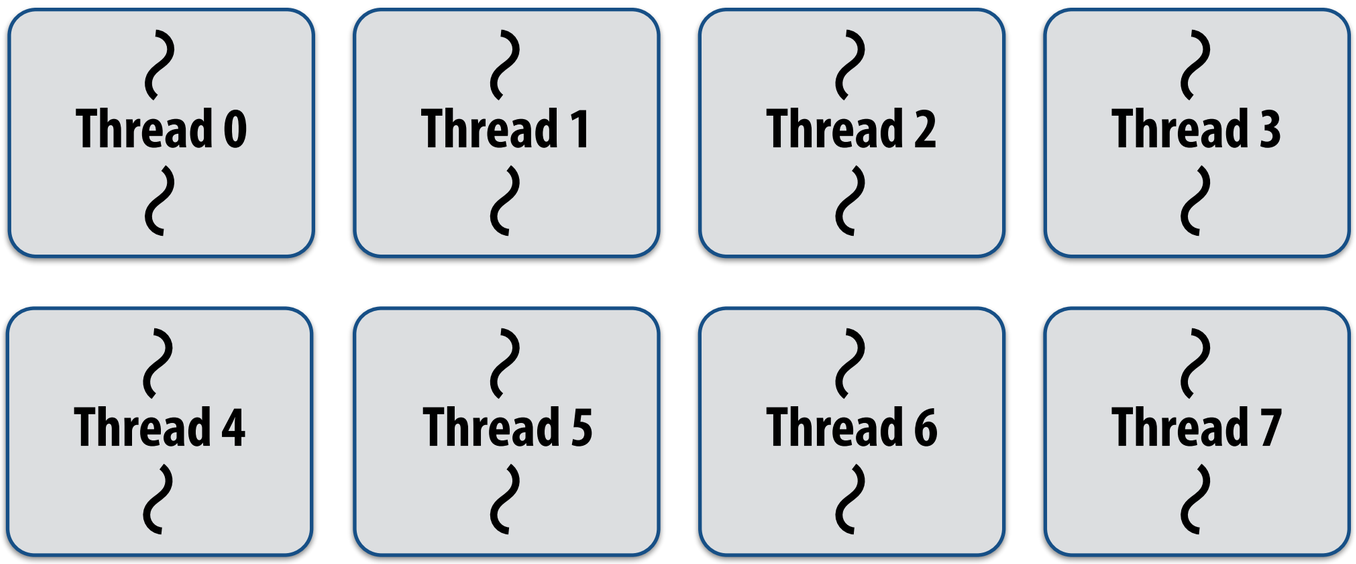
Scheduling fork-join programs

- Consider very simple scheduler:
  - Launch pthread for each `cilk_spawn` using `pthread_create`
  - Translate `cilk_sync` into appropriate `pthread_join` calls

- Potential performance problems?
  - Heavyweight spawn operation
  - Many more concurrently running threads than cores
    - Context switching overhead
    - Larger working set than necessary, less cache locality

Pool of worker threads

- Cilk Plus runtime maintains pool of worker threads
  - Think: all threads created at application launch *
  - Exactly as many worker threads as execution contexts in the machine



Example: Eight thread worker pool for my quad-core laptop with Hyper-Threading

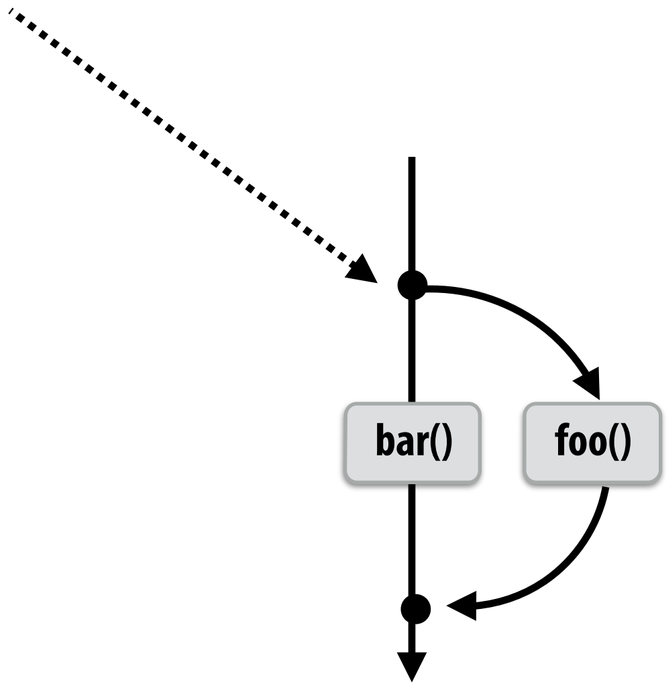
```
while (work_exists()) {
    work = get_new_work();
    work.run();
}
```

* It's perfectly fine to think about it this way, but in reality, runtimes tend to be lazy and initialize worker threads on the first Cilk spawn. (This is a common implementation strategy, ISPC does the same with worker threads that run ISPC tasks.)

Consider execution of the following code

Specifically, consider execution at point of spawn of foo()

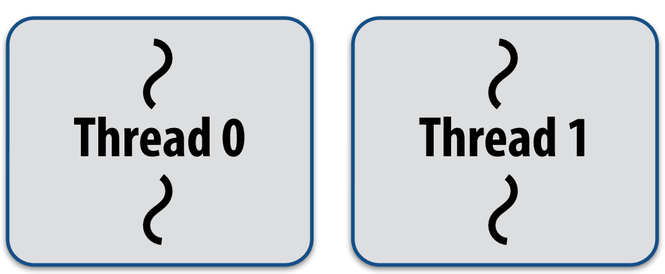
```
cilk_spawn foo();
bar();
cilk_sync;
```



spawned child

continuation (rest of calling function)

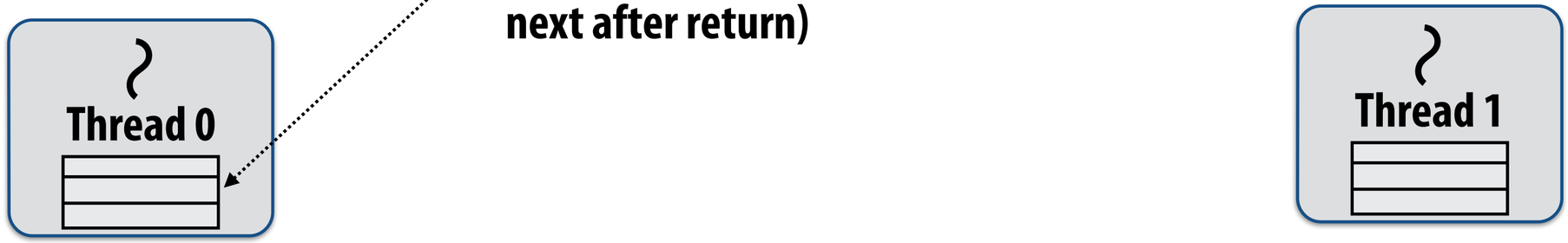
What threads should foo() and bar() be executed by?



Serial implementation

Run child first via function call (continuation is implicit in thread's stack)

- Thread runs foo(), then returns from foo(), then runs bar()

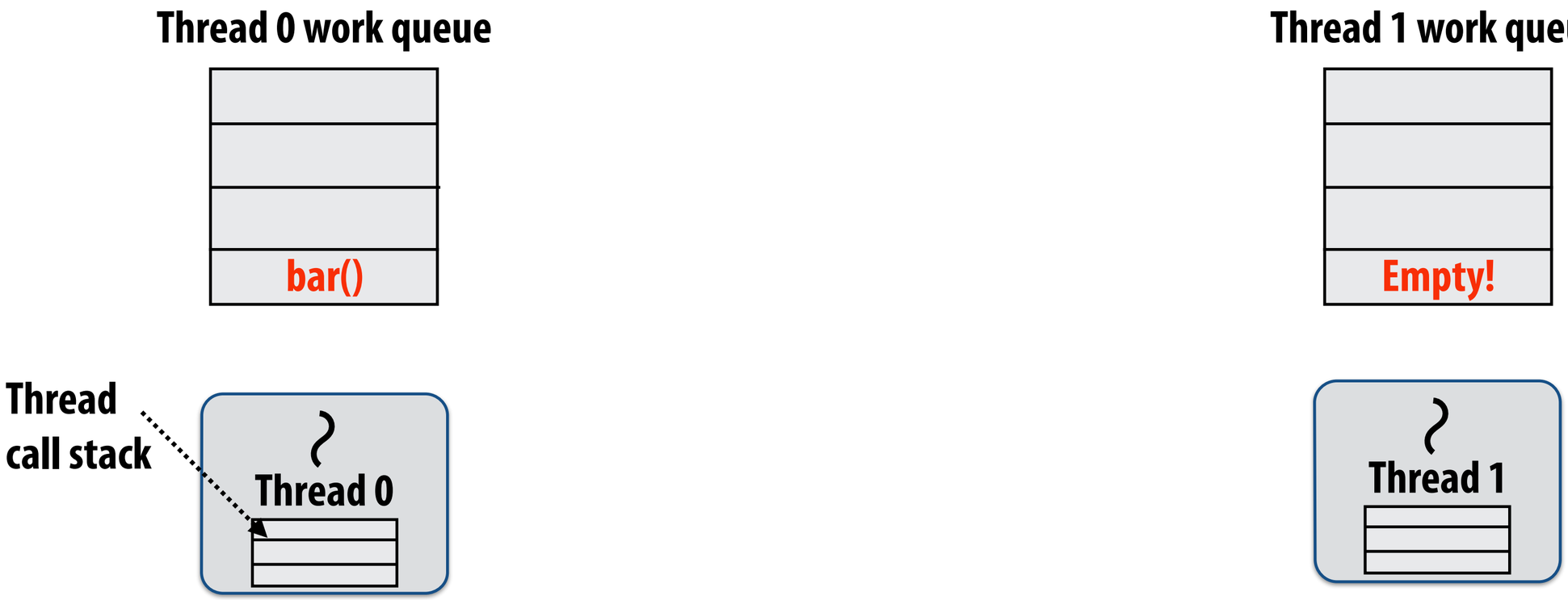


Executing foo()…

Thread 1 goes idle…

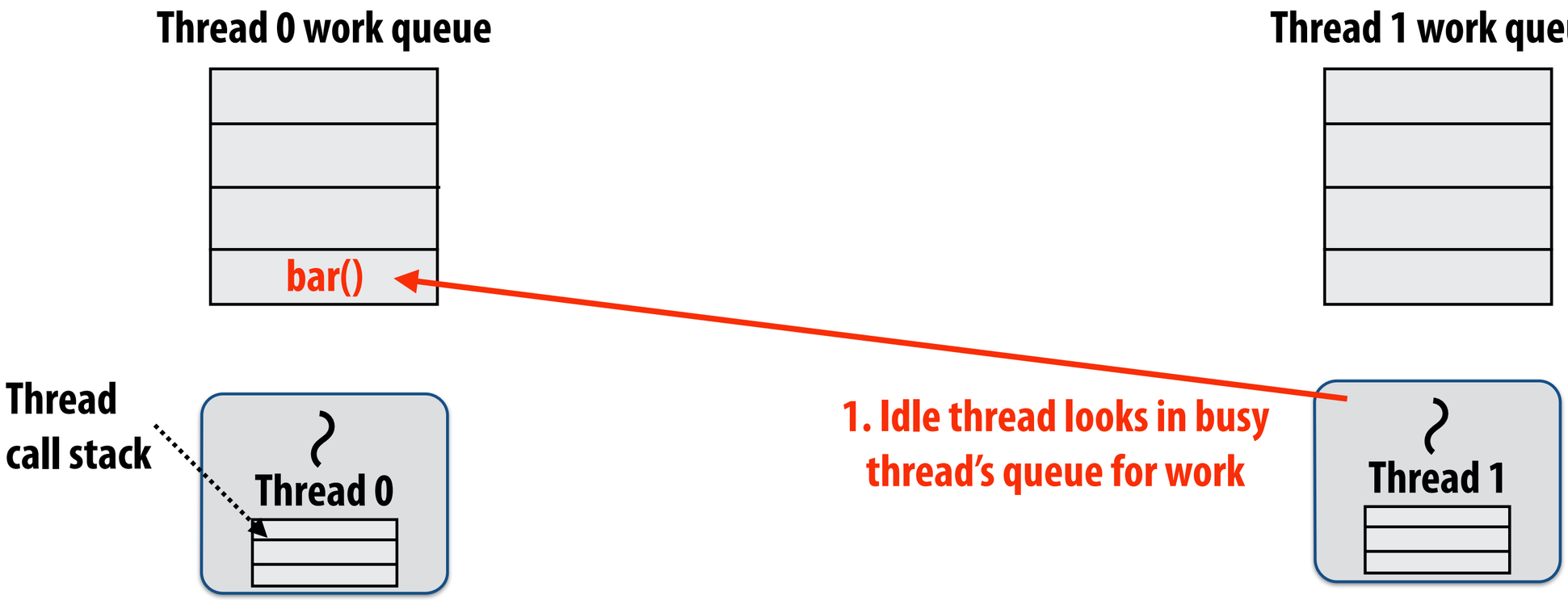
Inefficient: thread 1 could be performing bar() at this time!

Per-thread work queues store “work to do”



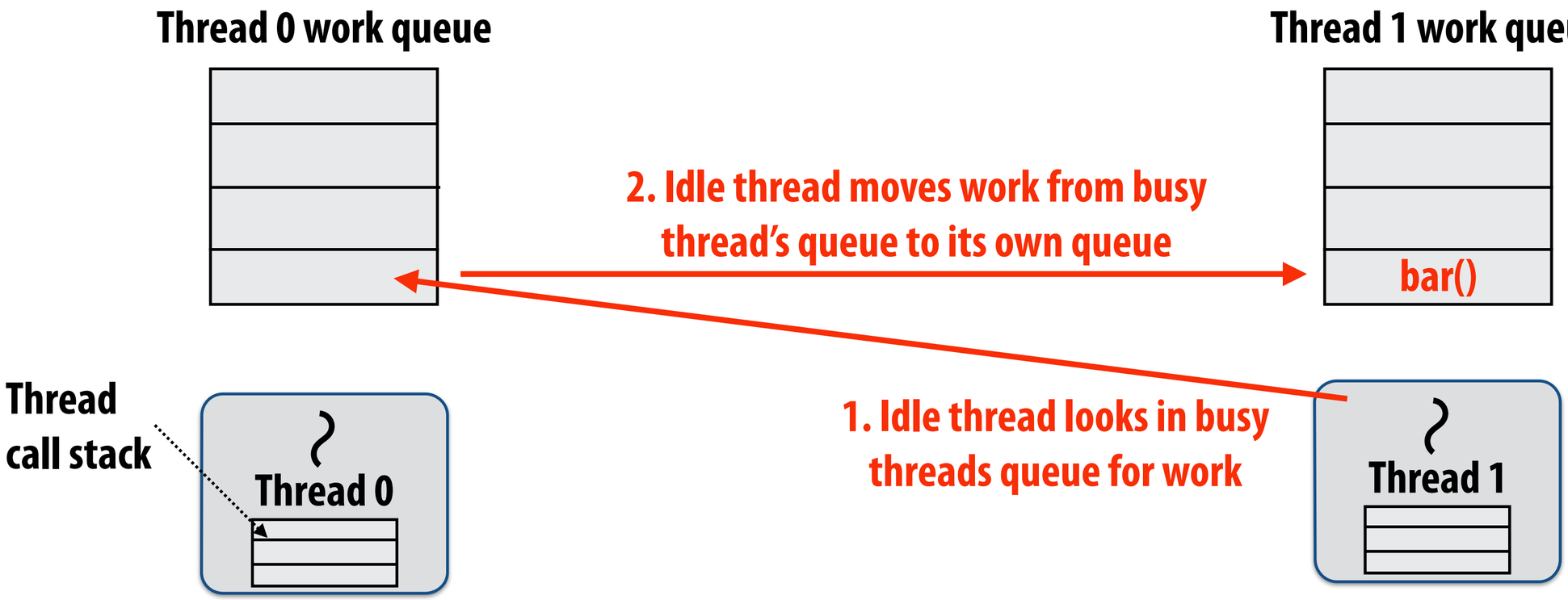
Executing foo()…

Idle threads "steal" work from busy threads



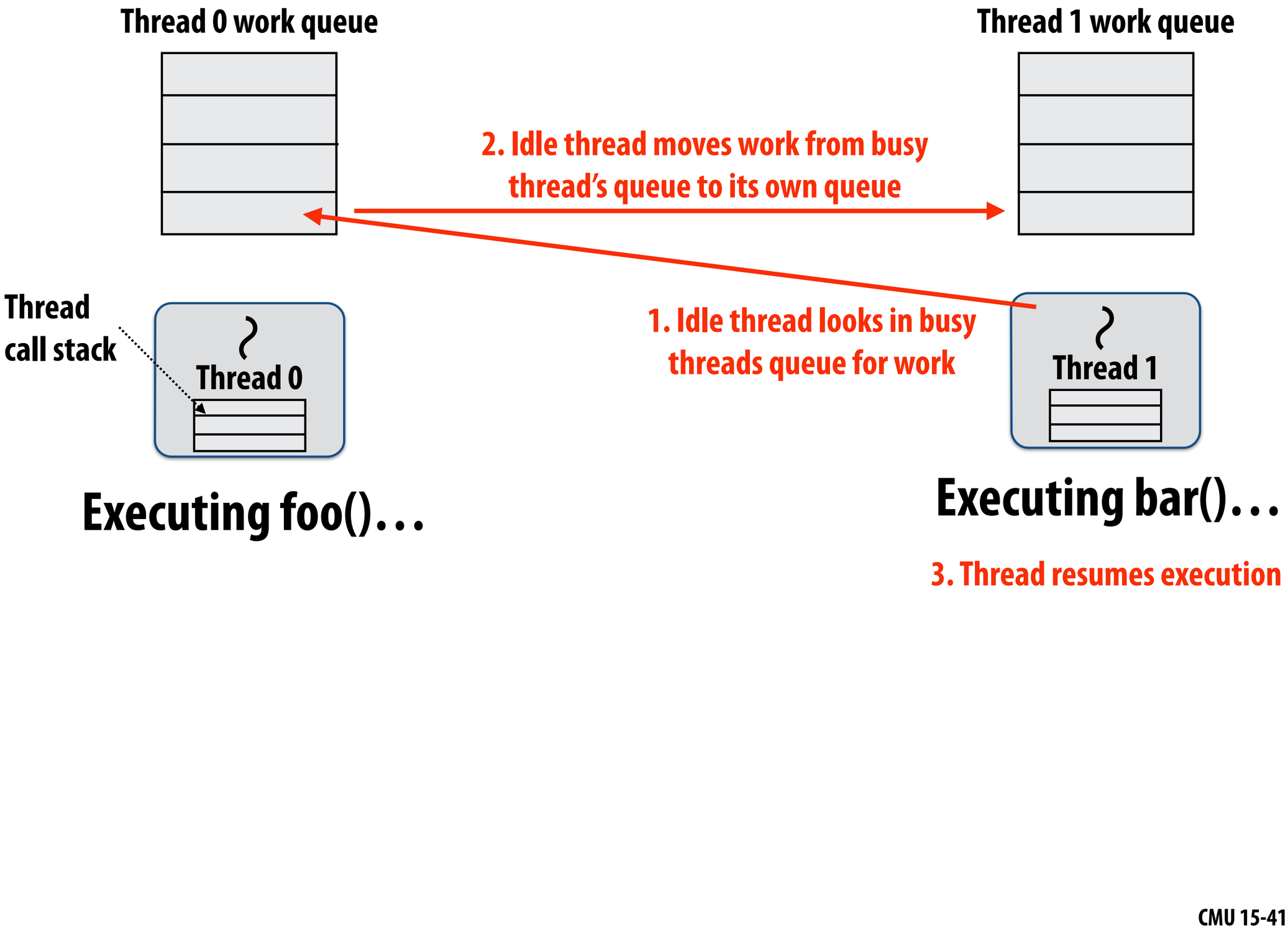
Executing foo()…

Idle threads "steal" work from busy threads



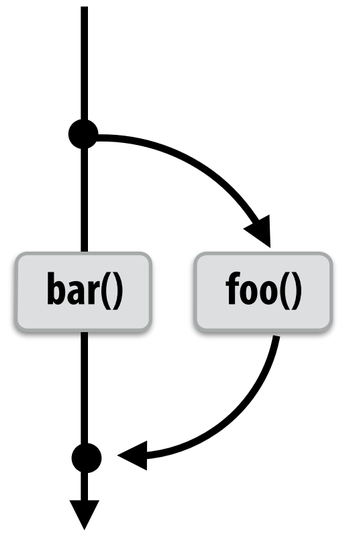
Executing foo()…

Idle threads "steal" work from busy threads



Alternative implementation:
At each spawn, system stores path not executed

```
cilk_spawn foo();

bar();

cilk_sync;
```



spawned child

continuation (rest of calling function)

Run continuation first: record child for later execution

- Child is made available for stealing by other threads ("child stealing")

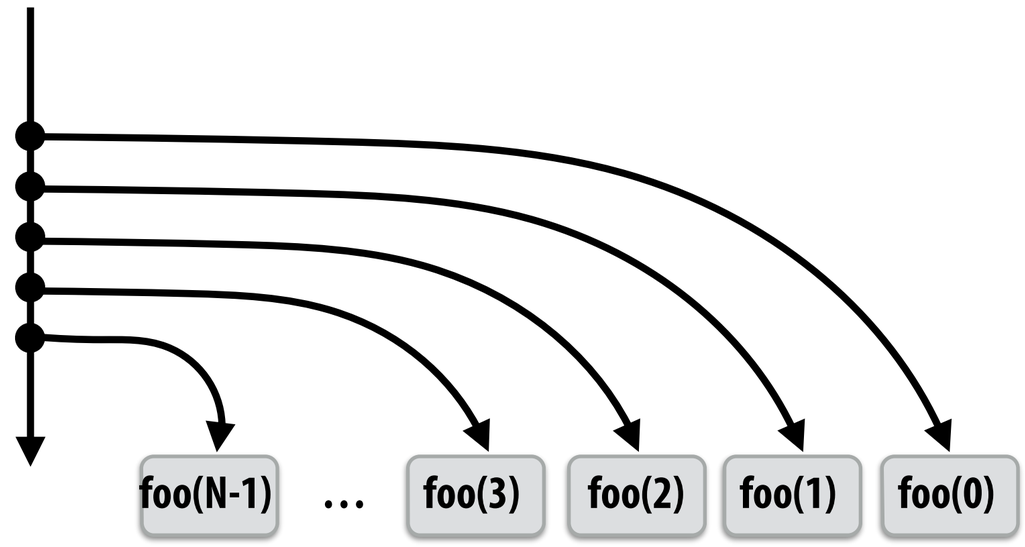
Run child first: record continuation for later execution

- Continuation is made available for stealing by other threads ("continuation stealing")

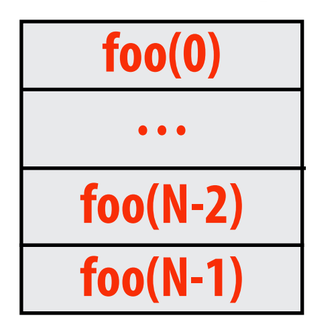
Which implementation do we choose?

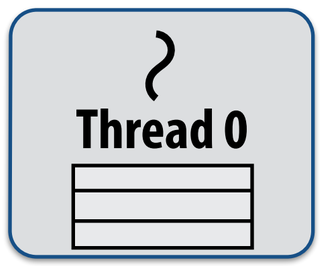
Consider thread executing the following code

```
for (int i=0; i<N; i++) {
  cilk_spawn foo(i);
}
cilk_sync;
```



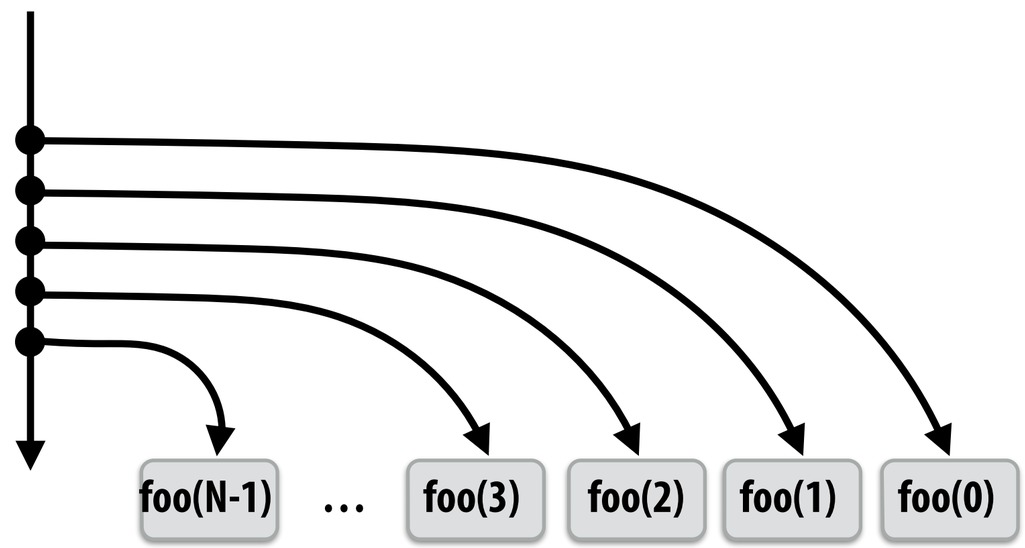
- **Child stealing (run continuation first)**
  - Caller thread spawns work for all iterations before executing any of it
  - Think: breadth-first traversal of call graph. O(N) space for spawned work (maximum space)
  - If no stealing, execution order is very different than that of program with `cilk_spawn` removed





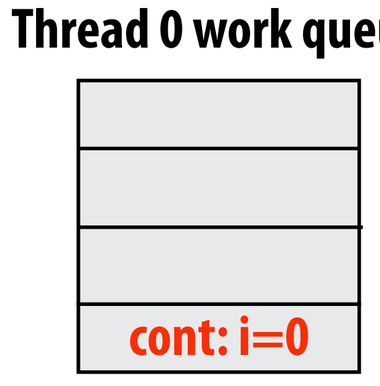
Consider thread executing the following code

```
for (int i=0; i<N; i++) {
  cilk_spawn foo(i);
}
cilk_sync;
```



■ Continuation stealing (run child first)

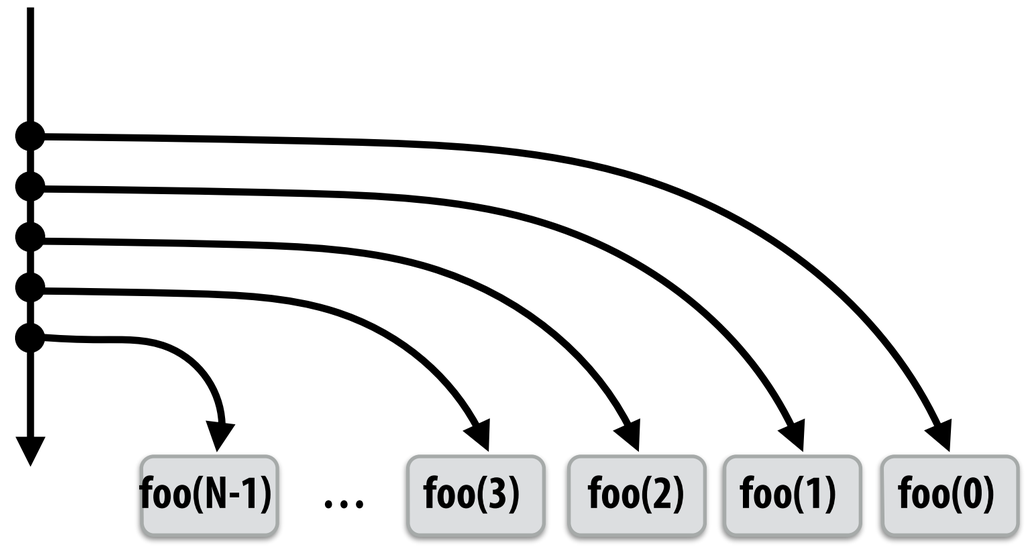
- Caller thread only creates one item to steal (continuation that represents all remaining iterations)
- If no stealing occurs, thread continually pops continuation from work queue, enqueues new continuation (with updated value of `i`)
- Order of execution is the same as for program with spawn removed.
- Think: depth-first traversal of call graph



Executing foo(0)…

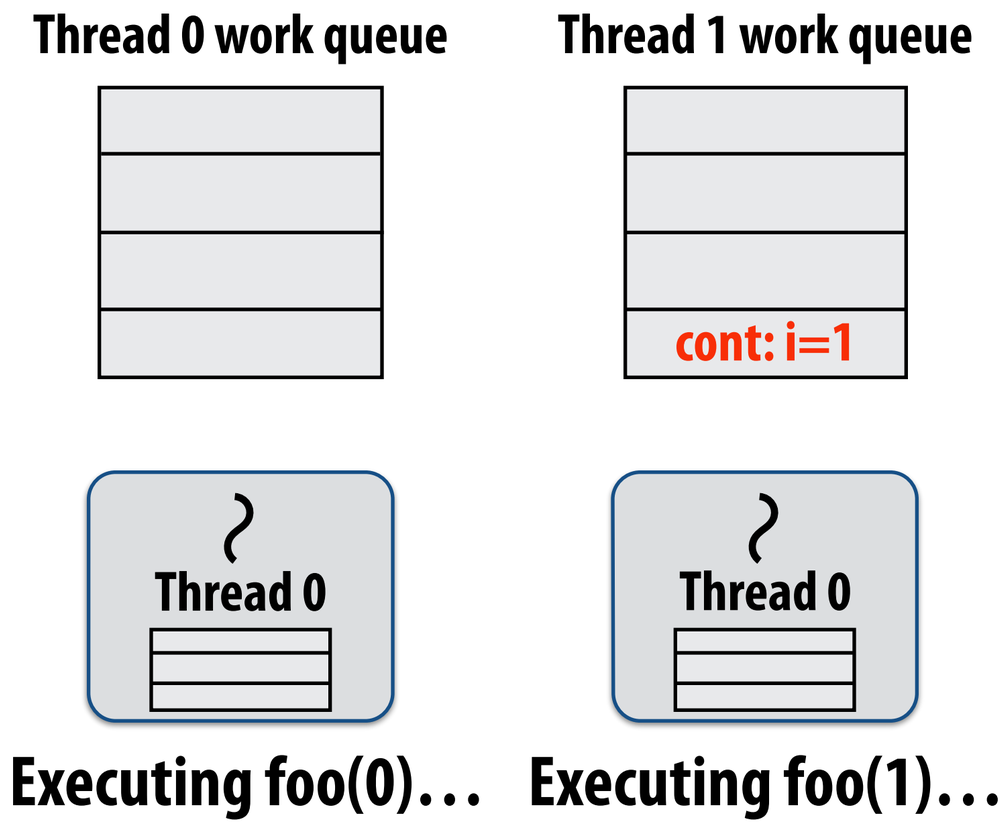
Consider thread executing the following code

```
for (int i=0; i<N; i++) {
  cilk_spawn foo(i);
}
cilk_sync;
```



■ Continuation stealing (run child first)

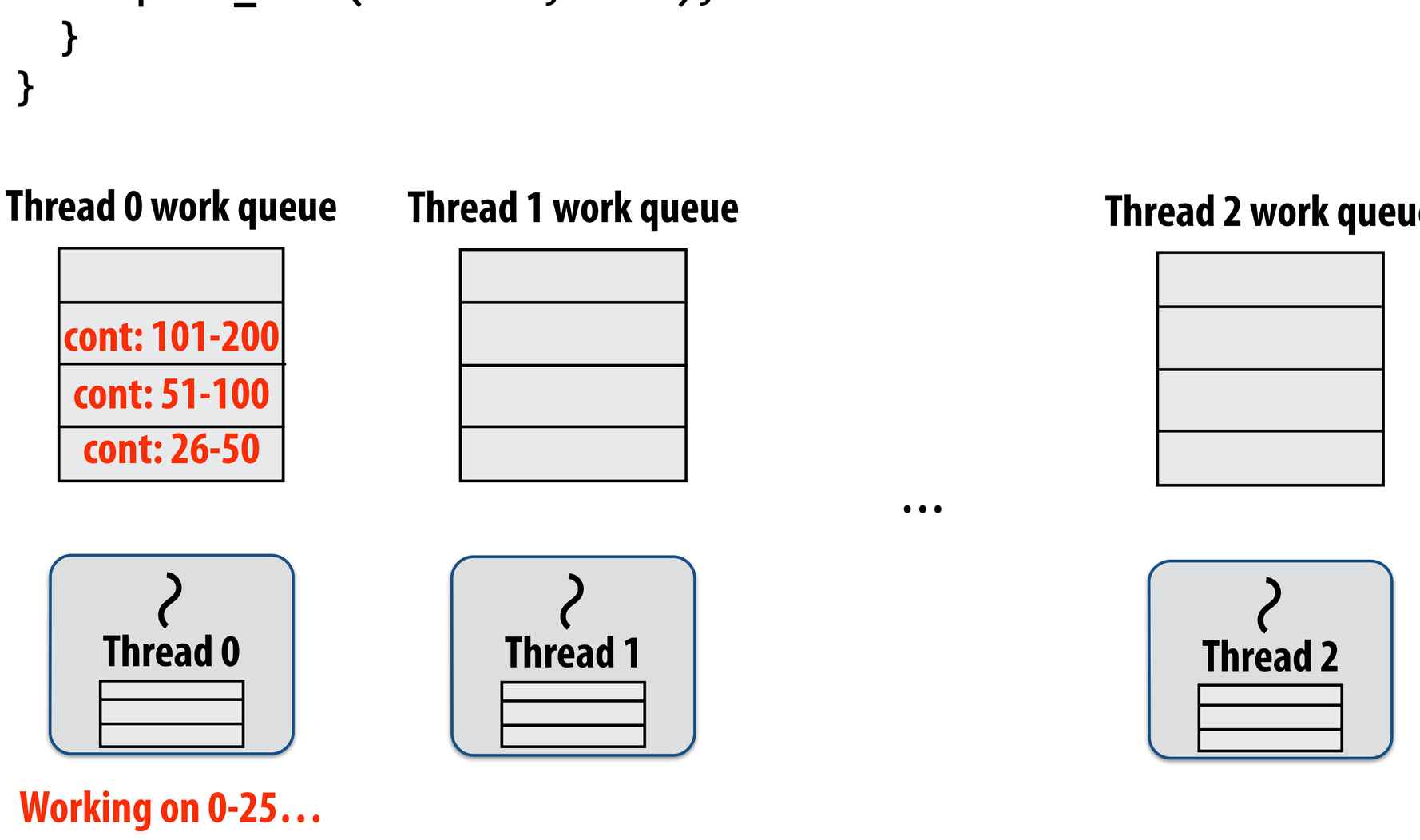
- If continuation is stolen, stealing thread spawns and executes next iteration
- Enqueues continuation with `i` advanced by 1
- Can prove that work queue storage for system with T threads is no more than T times that of stack storage for single threaded execution



Scheduling quicksort: assume 200 elements

```
void quick_sort(int* begin, int* end) {
  if (begin >= end - PARALLEL_CUTOFF)
    std::sort(begin, end);
  else {
     int* middle = partition(begin, end);
     cilk_spawn quick_sort(begin, middle);
     quick_sort(middle+1, last);
  }
}
```

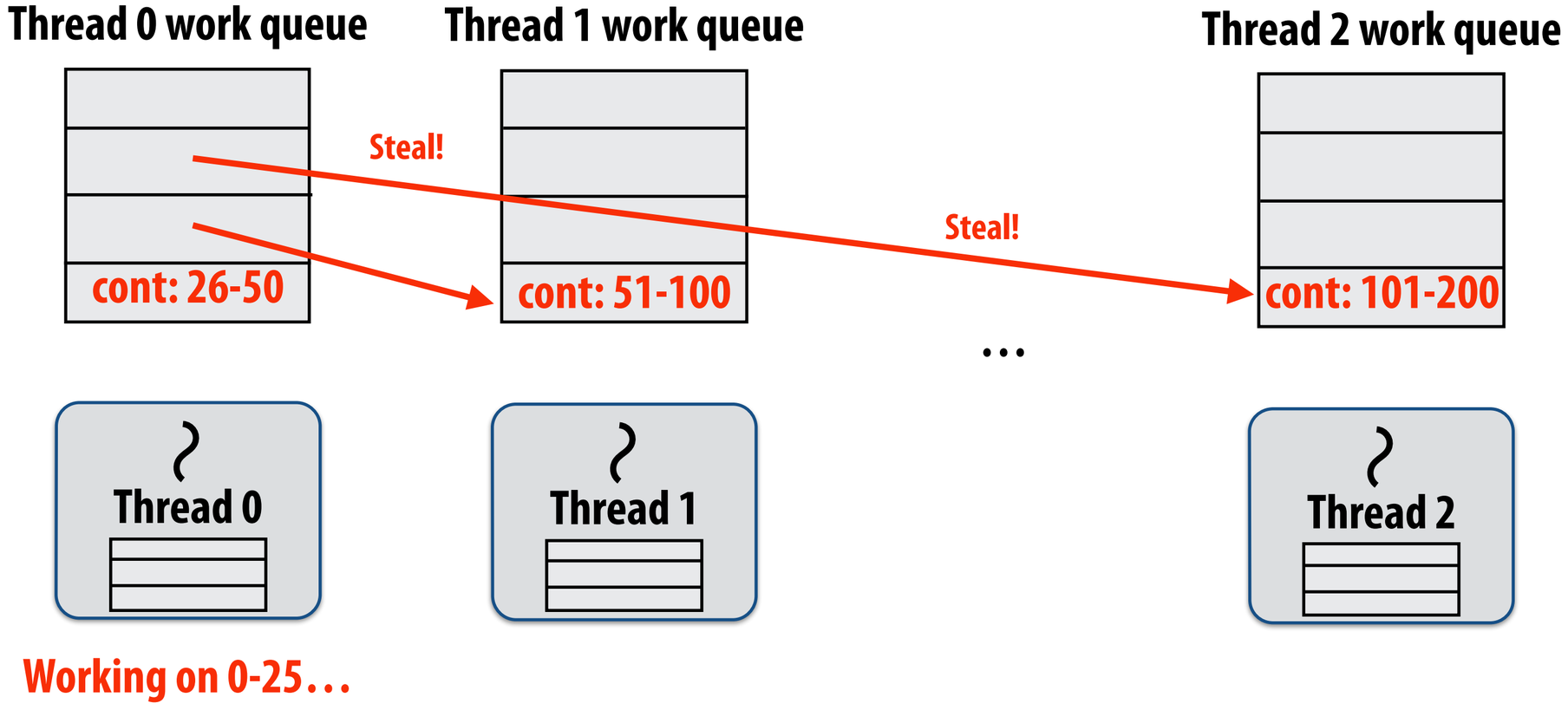
What work in the queue should other threads steal?



Implementing work stealing: dequeue per worker

Work queue implemented as a dequeue (double ended queue)

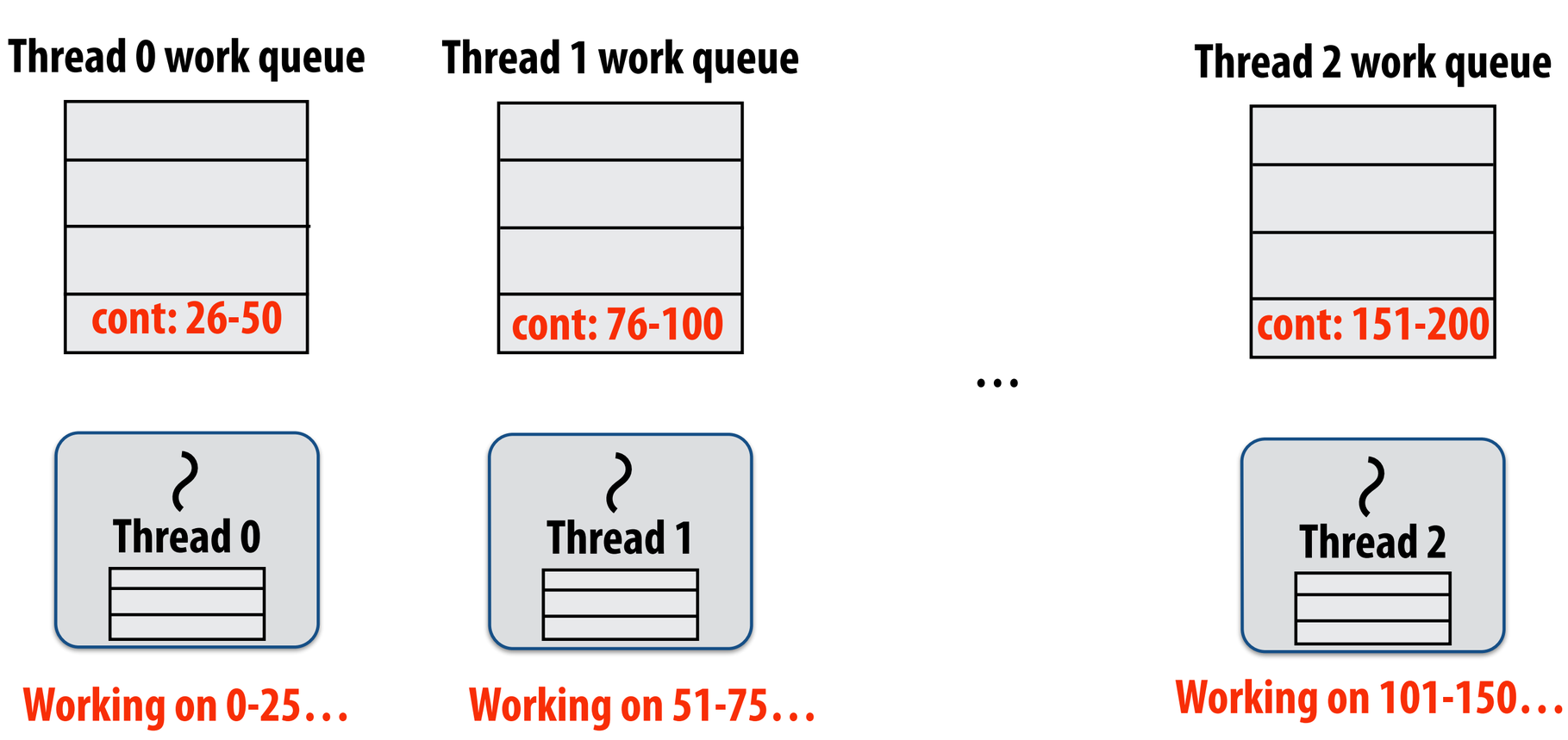
- Local thread pushes/pops from the "tail" (bottom)
- Remote threads steal from "head" (top)
- Efficient lock-free dequeue implementations exist



Implementing work stealing: dequeue per worker

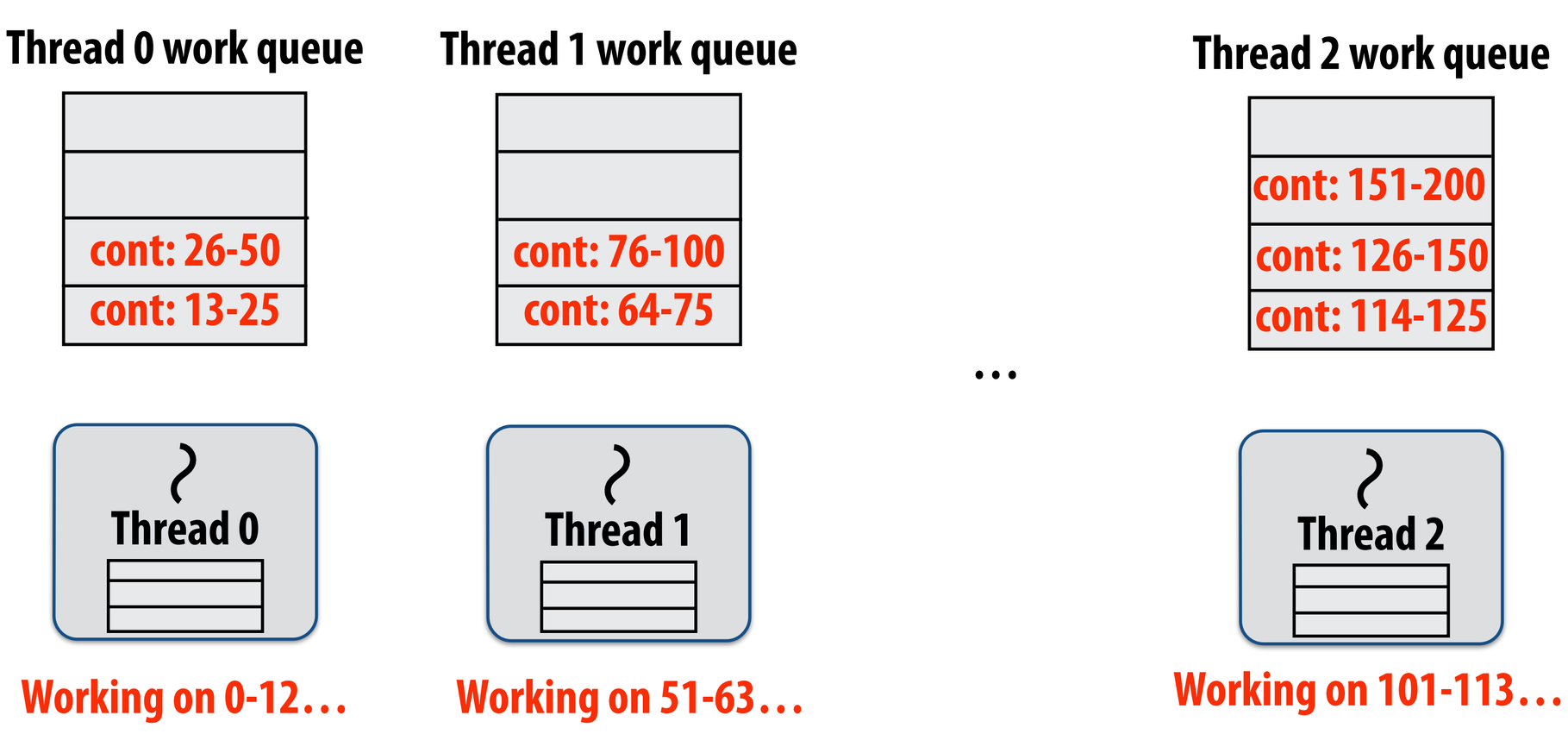
Work queue implemented as a dequeue (double ended queue)

- Local thread pushes/pops from the "tail" (bottom)
- Remote threads steal from "head" (top)
- Efficient lock-free dequeue implementations exist



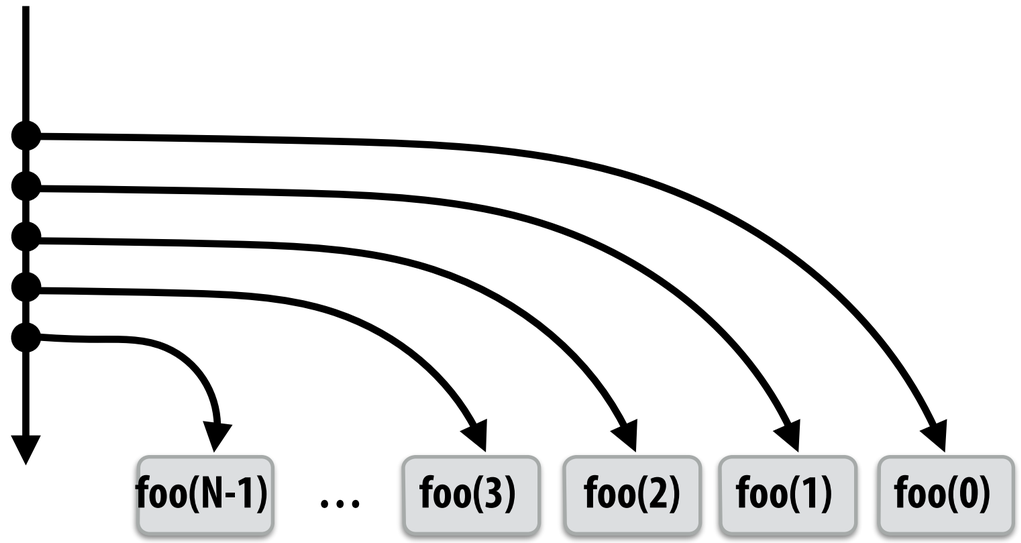
Implementing work stealing: random choice of victim

- Idle threads randomly choose a thread to attempt to steal from
- Stealing from top of dequeue…
  - Reduces contention with local thread: local thread is not accessing same part of dequeue that stealing threads do!
  - Steals work at beginning of call tree: this is a "larger" piece of work, so the cost of performing a steal is amortized over long future computation
  - Maximizes locality: (in conjunction with run-child-first policy) local thread works on local part of call tree



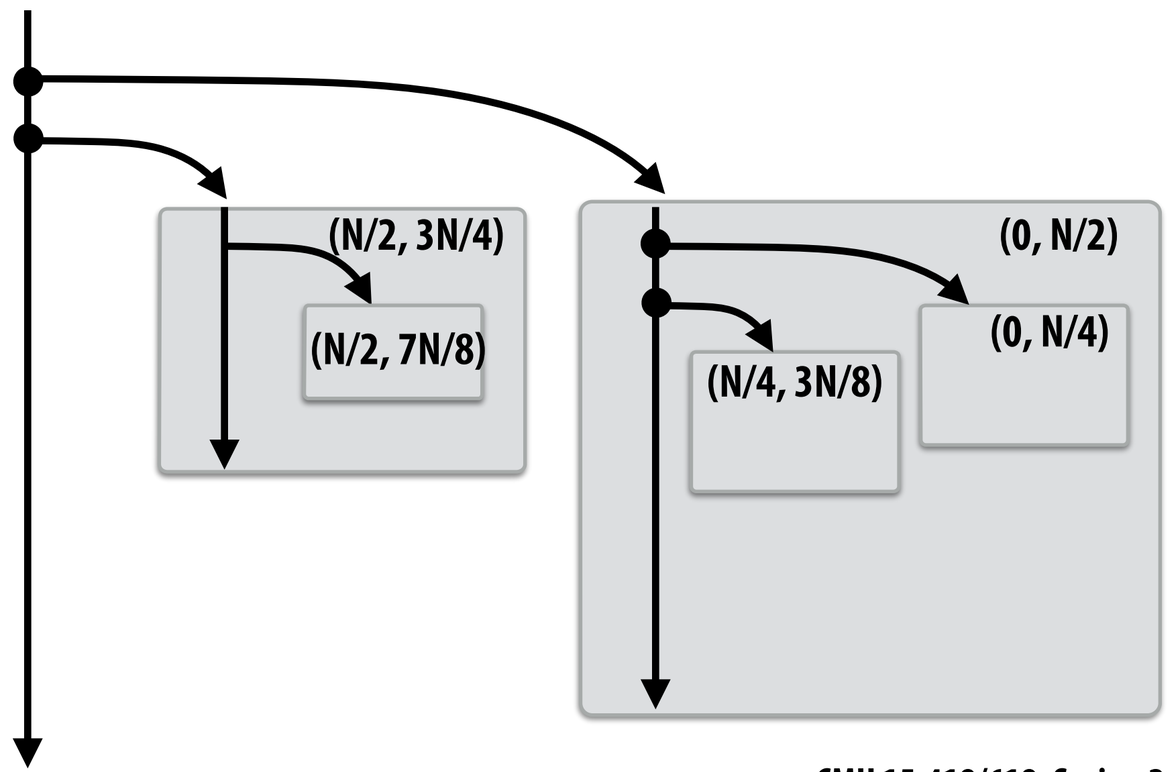
Child-first work stealing scheduler anticipates divide-and-conquer parallelism

```
for (int i=0; i<N; i++) {

  cilk_spawn foo(i);

}

cilk_sync;
```



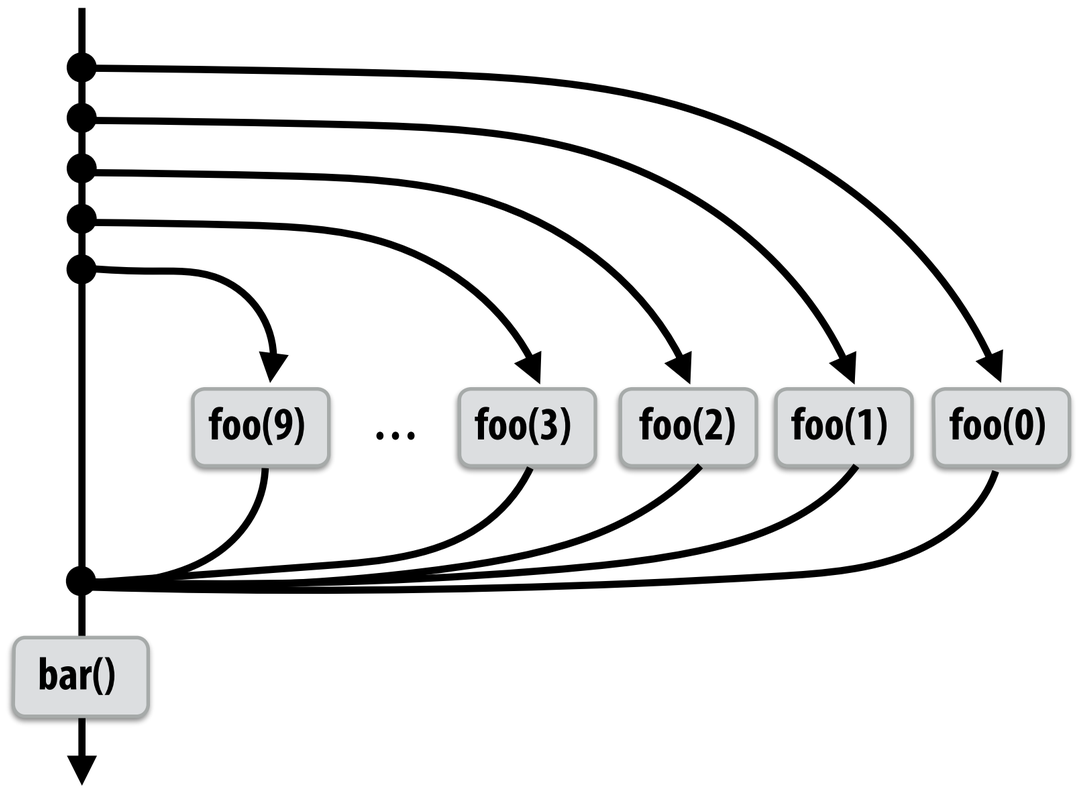
Code at right generates work in parallel: more quickly fills machine

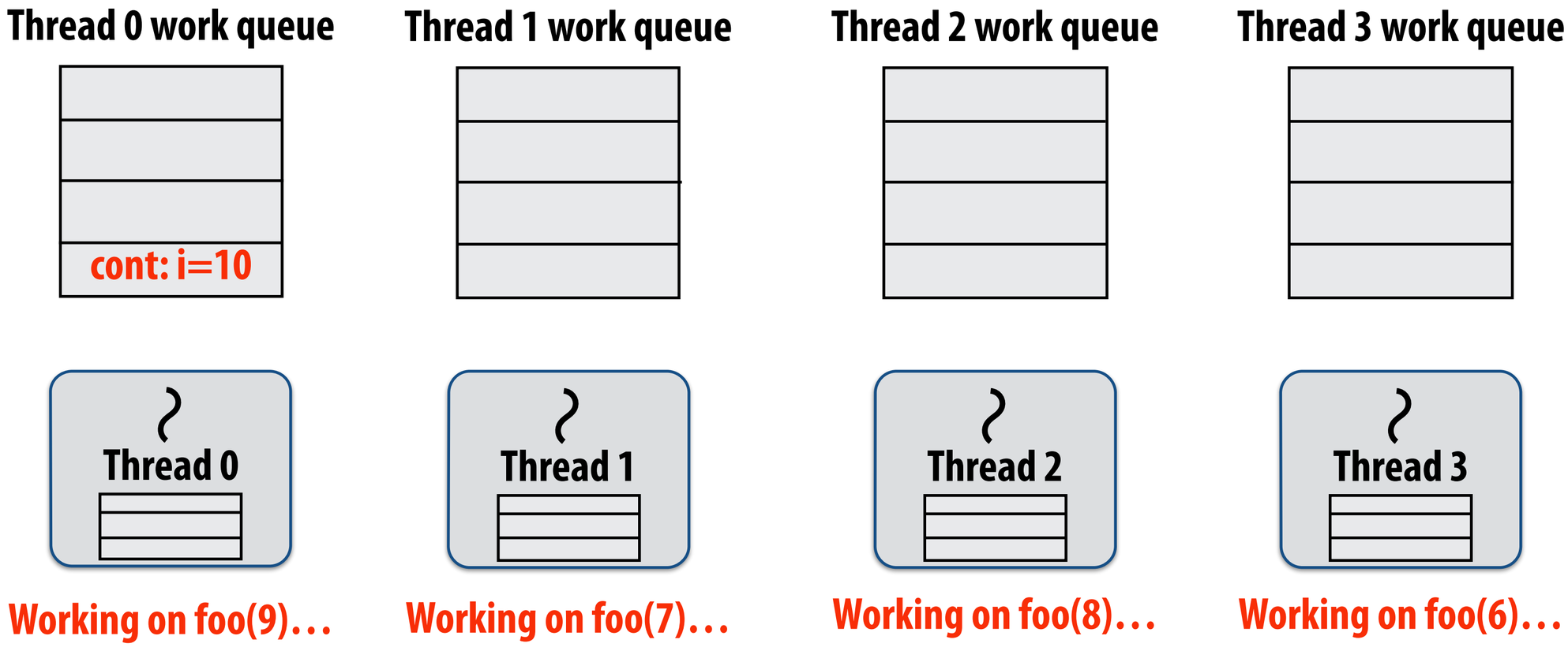
```
void recursive_for(int start, int end) {

  while (start <= end - GRANULARITY) {
    int mid = (end - start) / 2;
    cilk_spawn recursive_for(start, mid);
    start = mid;
  }

  for (int i=start; i<end; i++)
     foo(i);
}

recursive_for(0, N);
```



Implementing sync

```
for (int i=0; i<10; i++) {
  cilk_spawn foo(i);
}
cilk_sync;
bar();
```



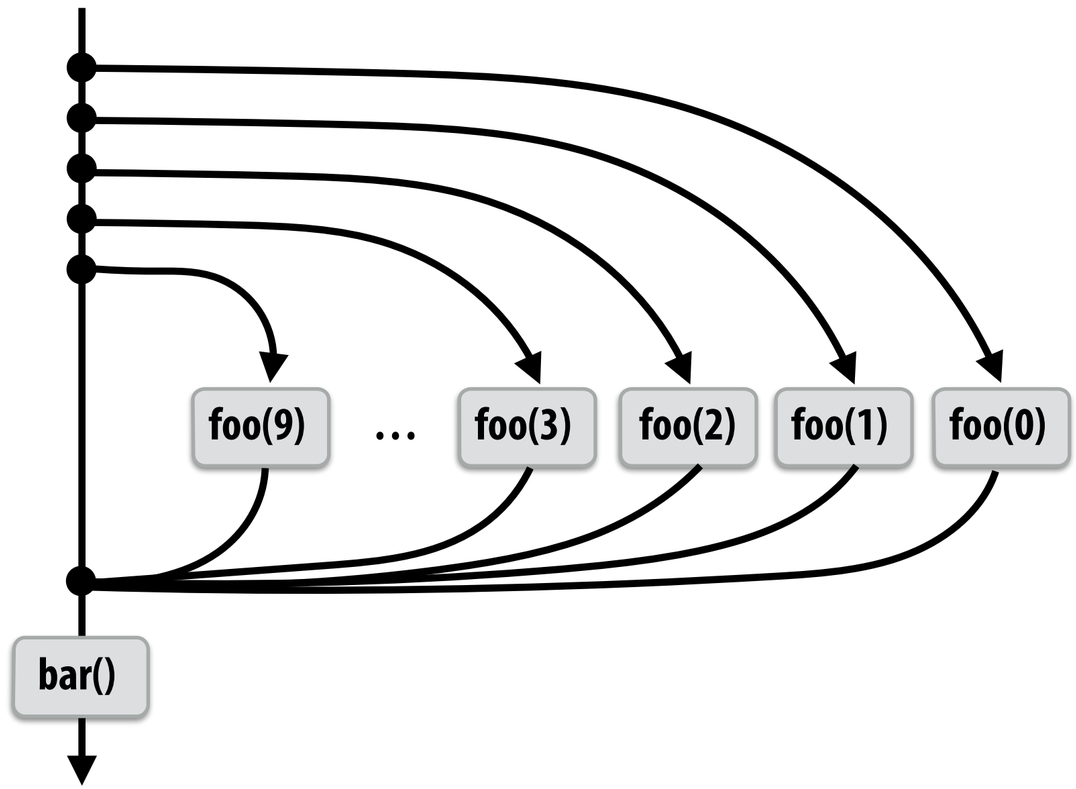


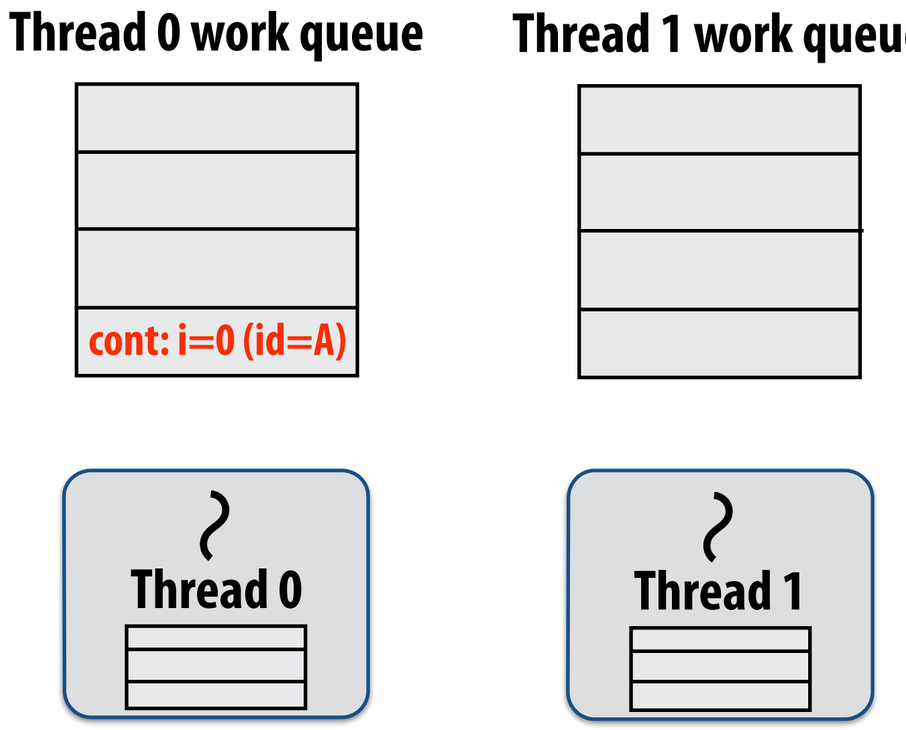
Implementing sync: case 1: no stealing

block (id: A)

```
for (int i=0; i<10; i++) {
  cilk_spawn foo(i);
}
cilk_sync;
bar();
```

Sync for all calls spawned in block A





Working on foo(0), id=A…

If no work has been stolen, then thread behaves like a serial program.

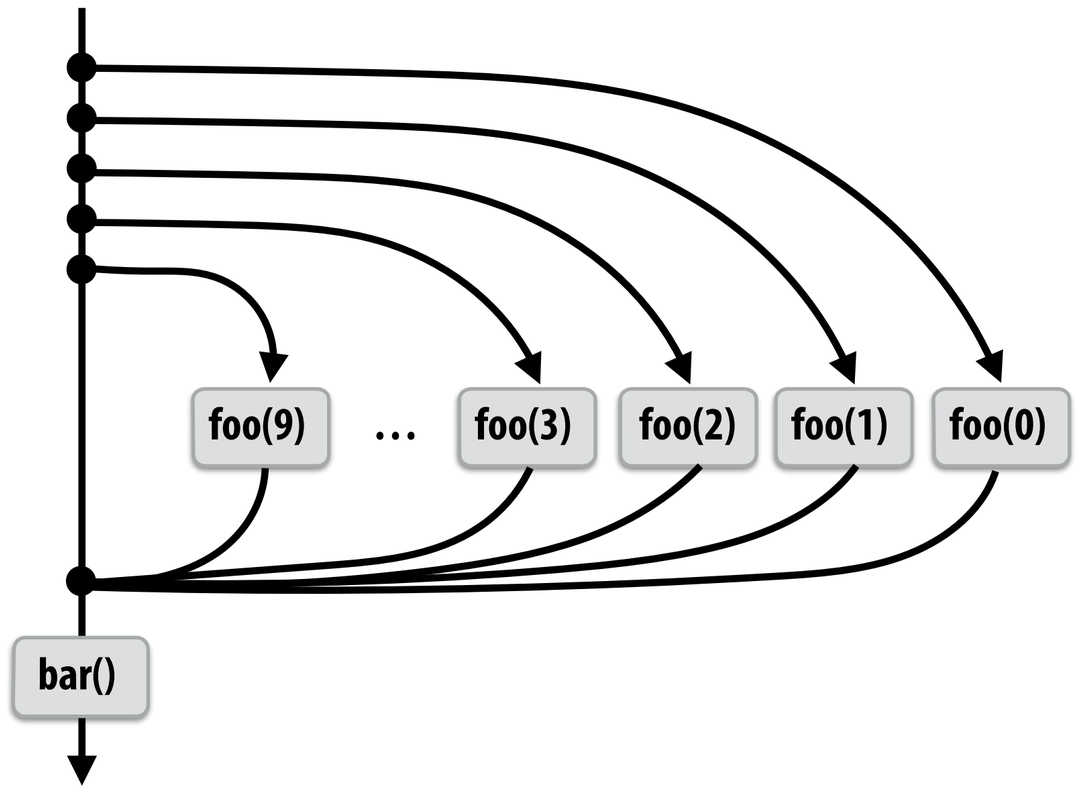
`cilk_sync` is a noop.

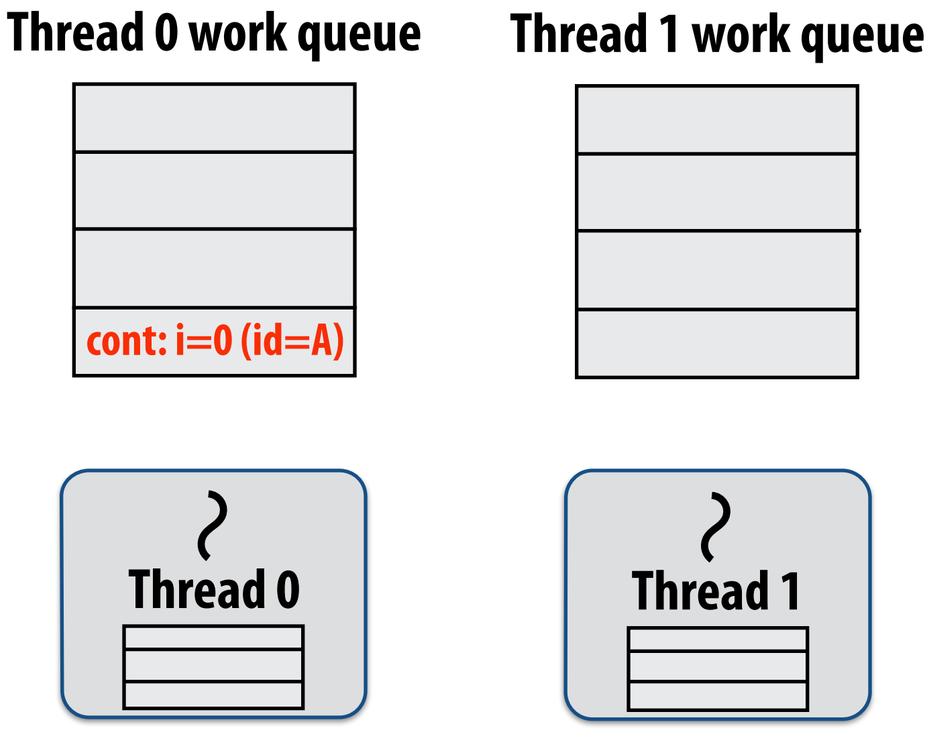
Implementing sync: case 2: stealing

block (id: A)

```
for (int i=0; i<10; i++) {
  cilk_spawn foo(i);
}
cilk_sync;
bar();
```

Sync for all calls spawned in block A





Working on foo(0), id=A…

Example 1: "stalling" join policy

Thread that initiates the fork must perform the sync.

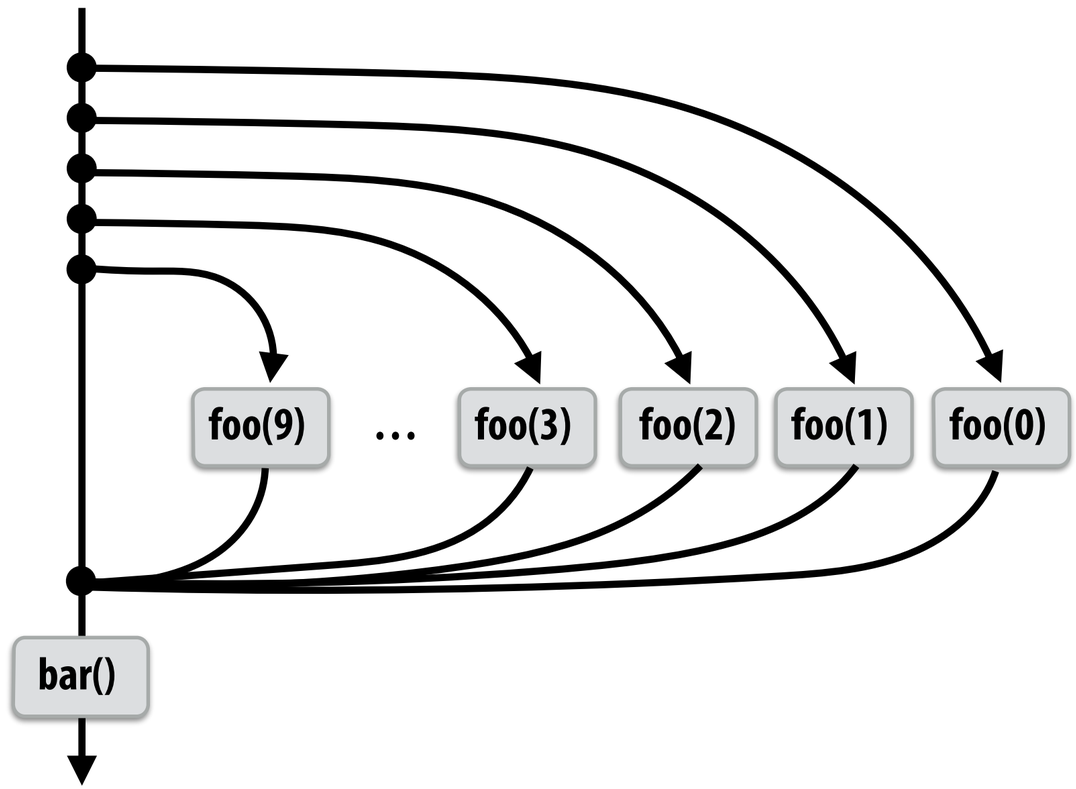
Therefore it waits for all spawned work to be complete.
In this case, thread 0 is the thread initiating the fork

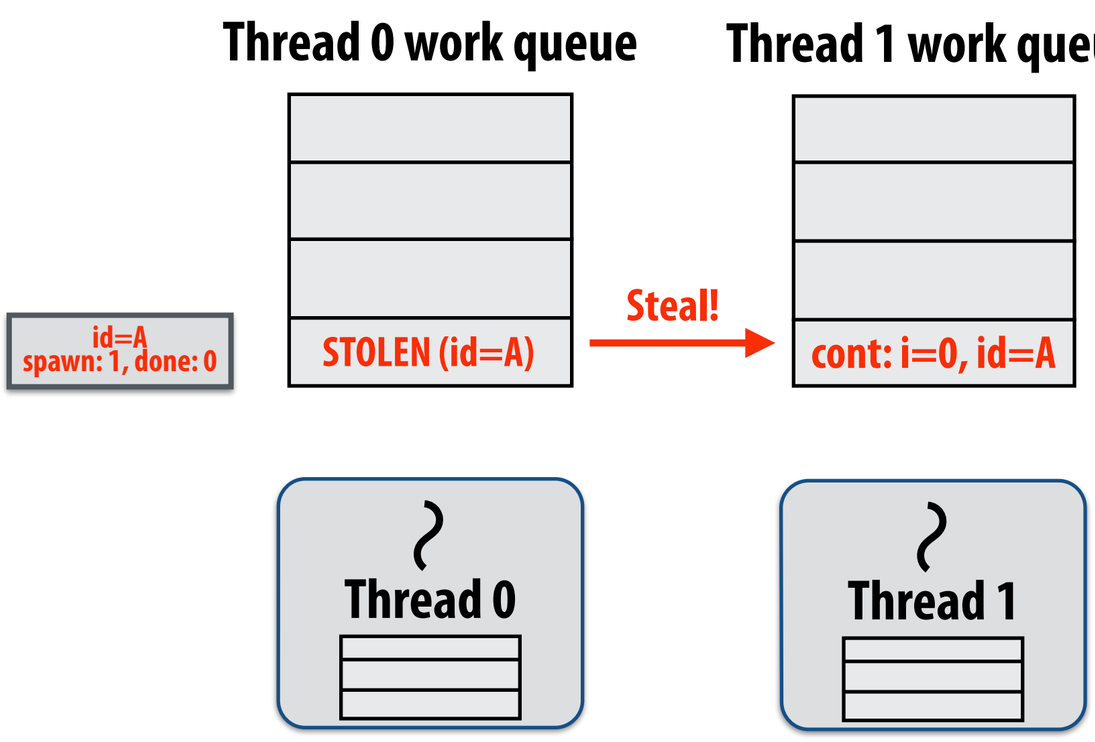
Implementing sync: case 2: stealing

block (id: A)

```
for (int i=0; i<10; i++) {
  cilk_spawn foo(i);
}
cilk_sync;
bar();
```

Sync for all calls spawned in block A





Idle thread 1 steals from busy thread 0

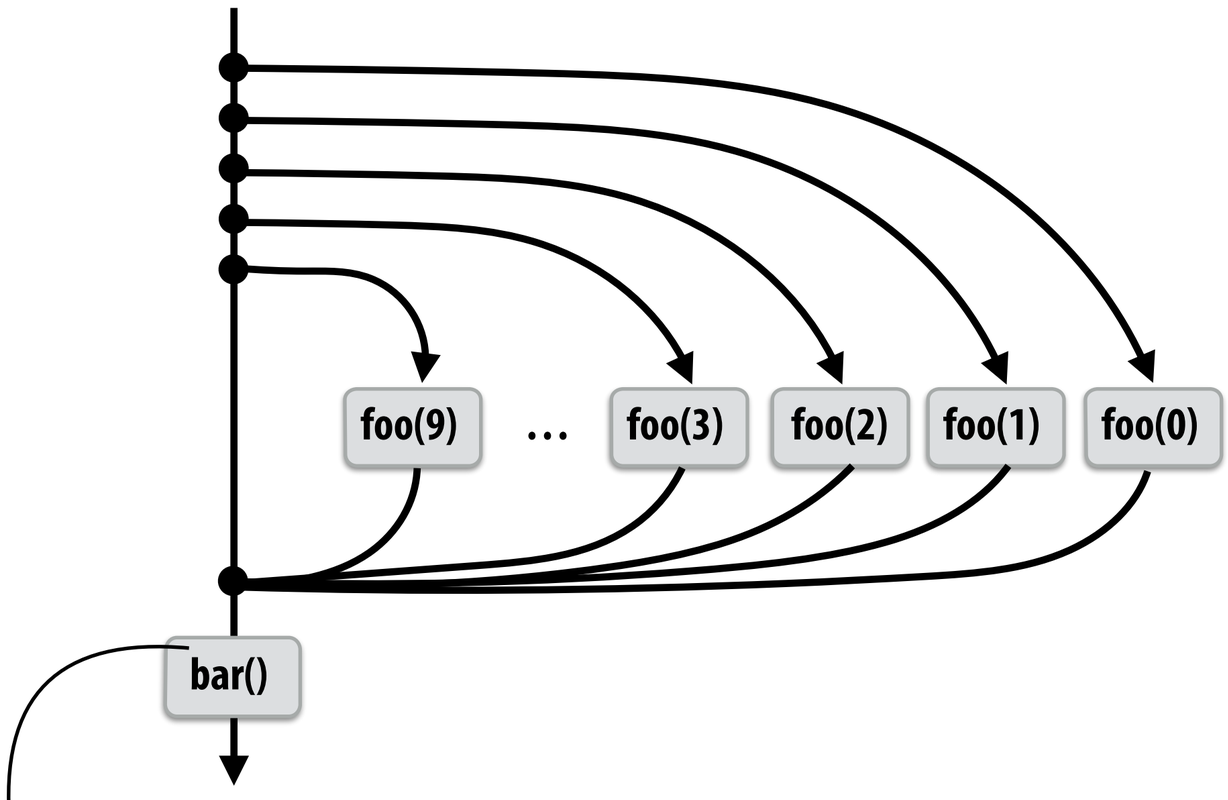
Note: descriptor for block A created

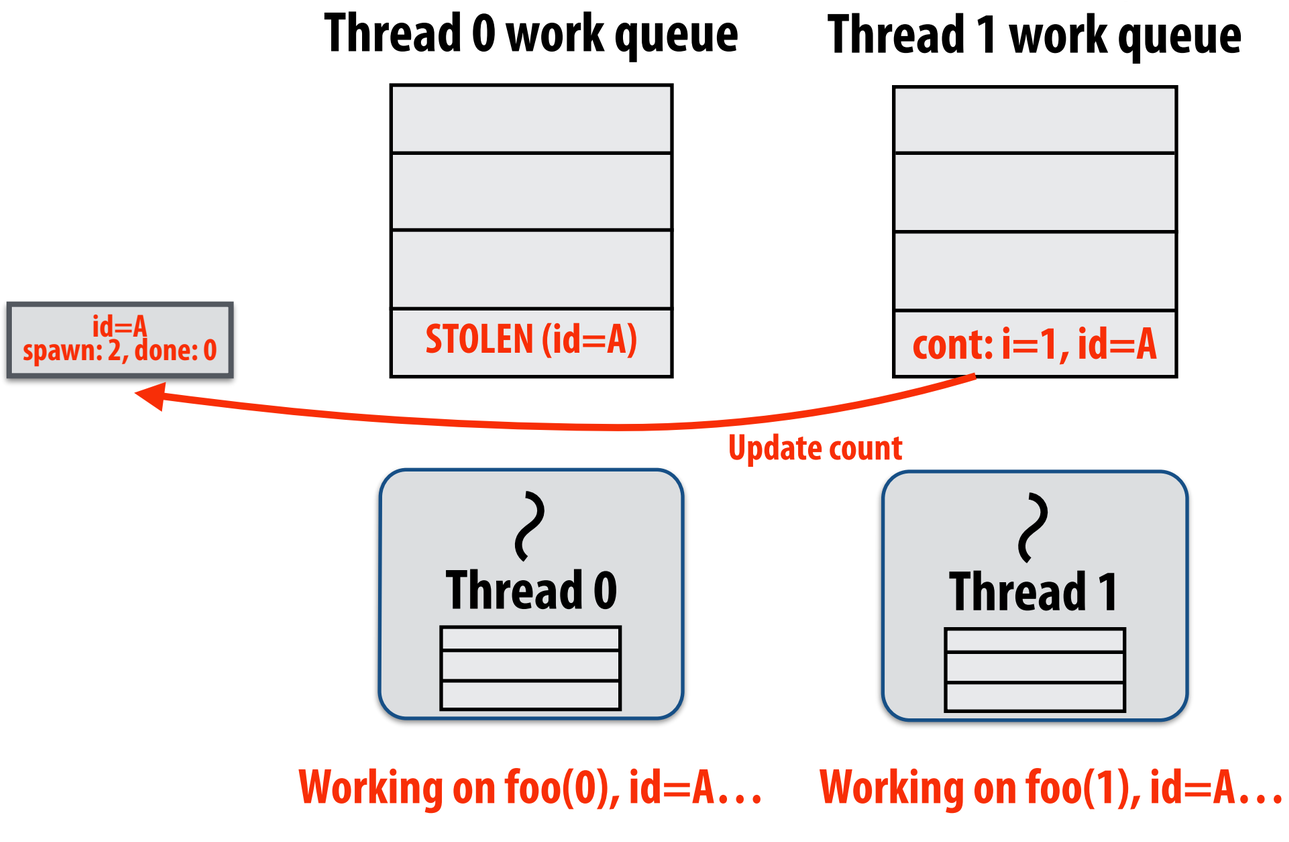
Implementing sync: case 2: stealing

block (id: A)

```
for (int i=0; i<10; i++) {
  cilk_spawn foo(i);
}
cilk_sync;
bar();
```

Sync for all calls spawned in block A





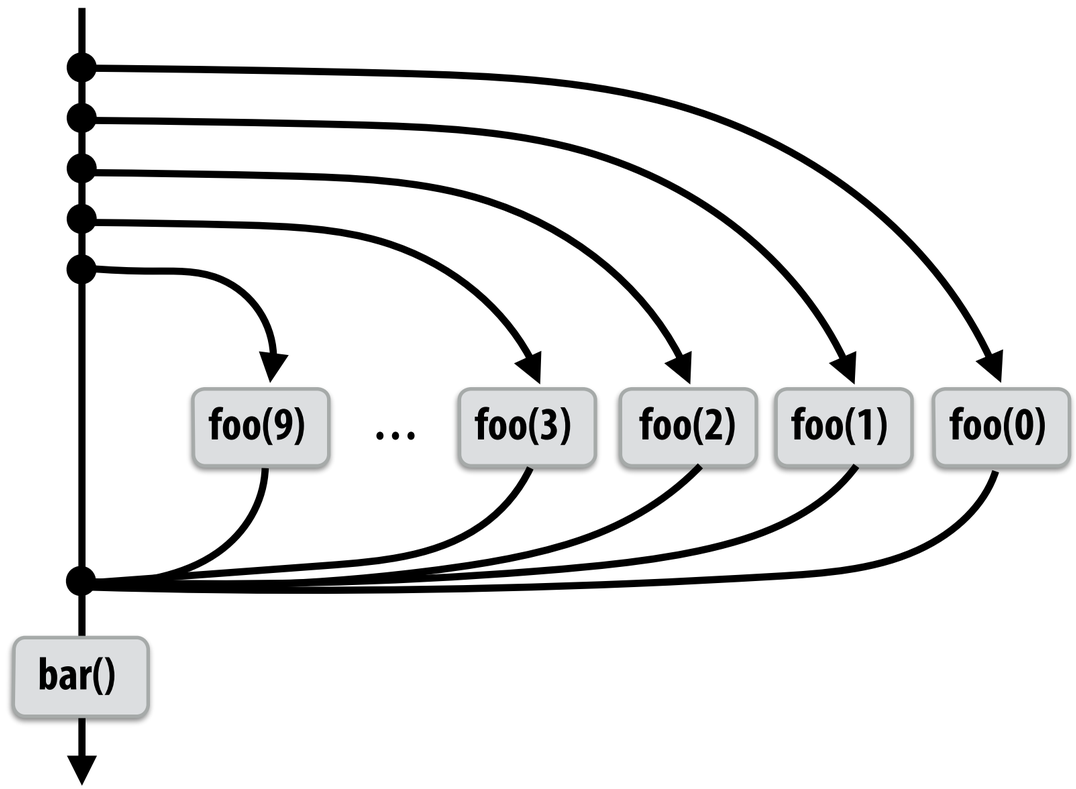
Thread 1 now running foo(1)

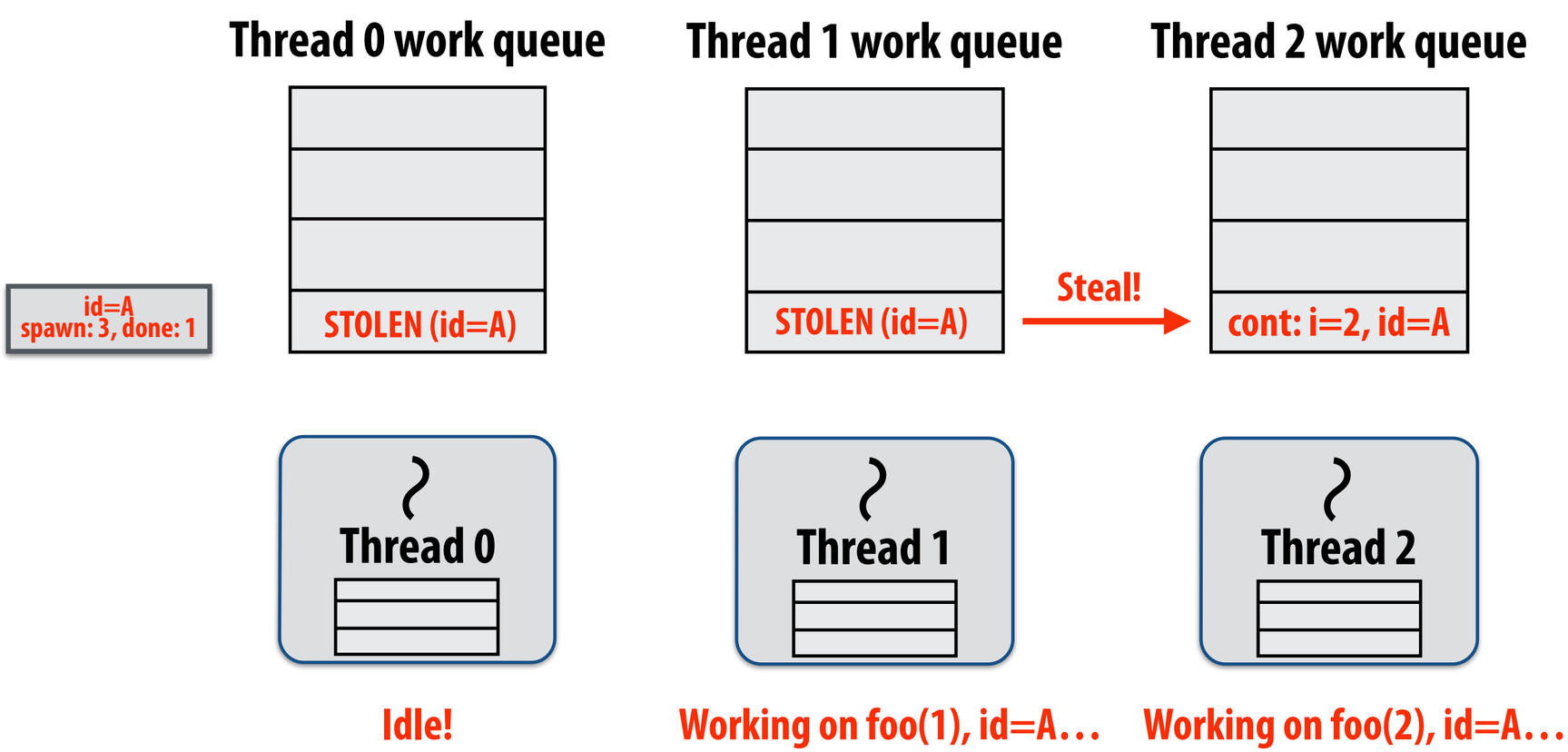
Implementing sync: case 2: stealing

block (id: A)

```
for (int i=0; i<10; i++) {
  cilk_spawn foo(i);
}
cilk_sync;
bar();
```

Sync for all calls spawned in block A





Thread 0 completes foo(0) (updates spawn descriptor)

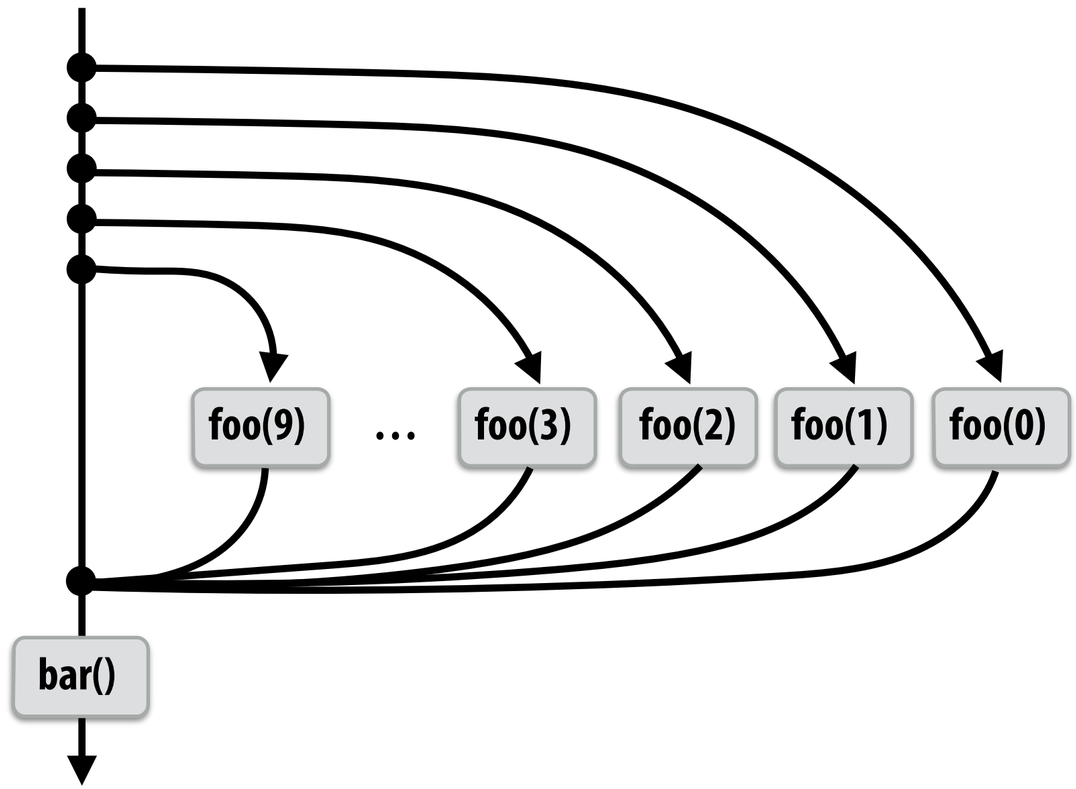
Thread 2 now running foo(2)

Implementing sync: case 2: stealing

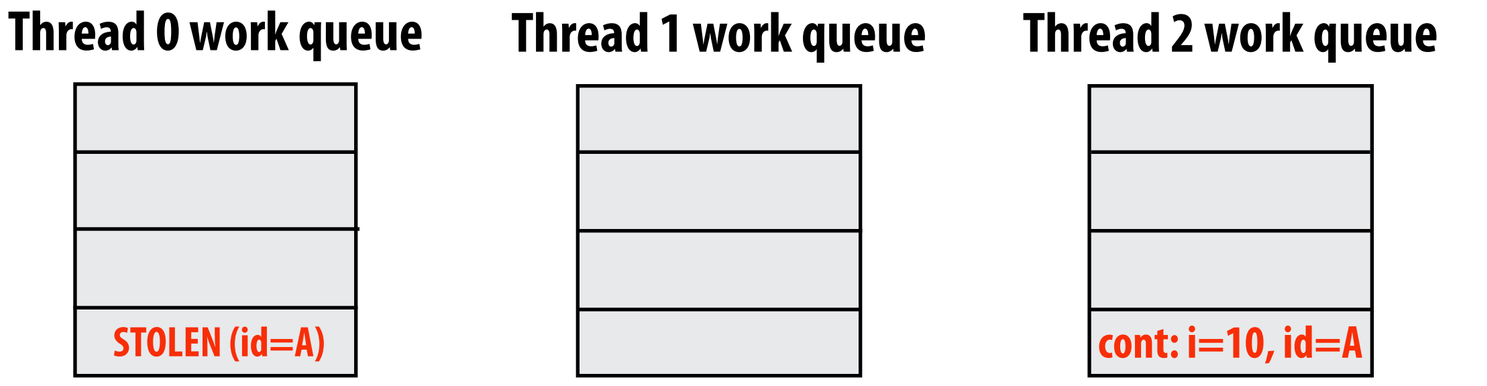
block (id: A)

```
for (int i=0; i<10; i++) {
  cilk_spawn foo(i);
}
cilk_sync;
bar();
```

Sync for all calls spawned in block A



id=A
spawn: 10, done: 9

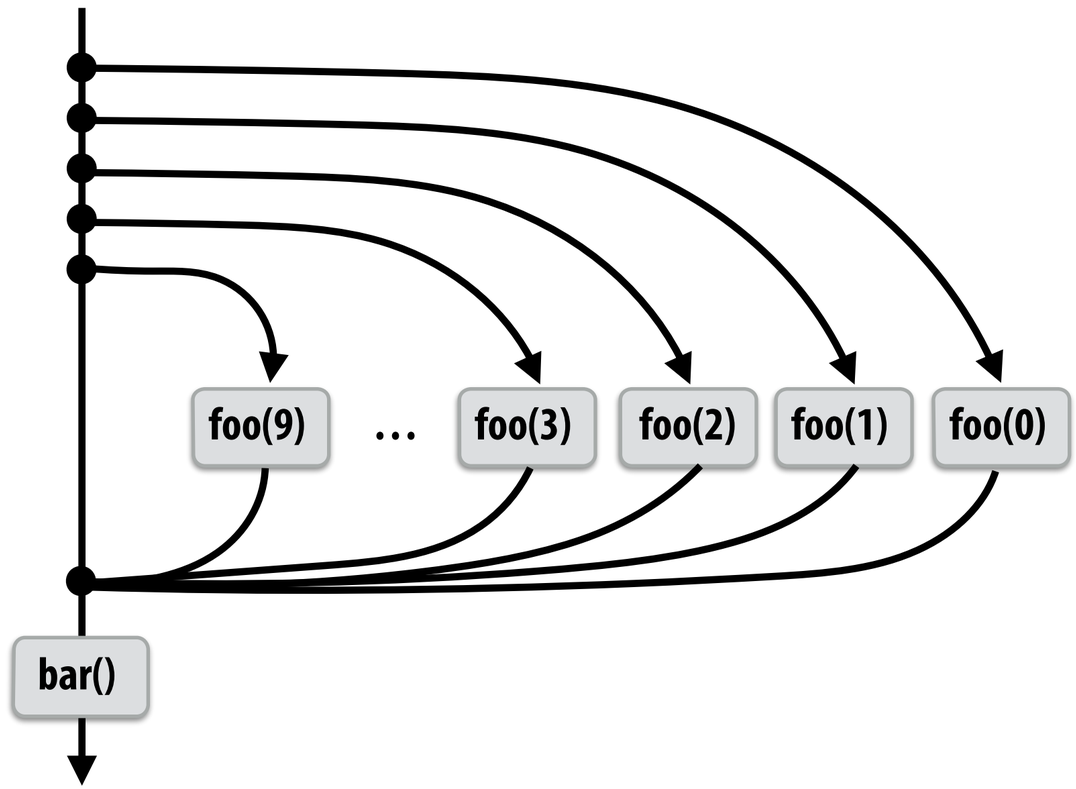


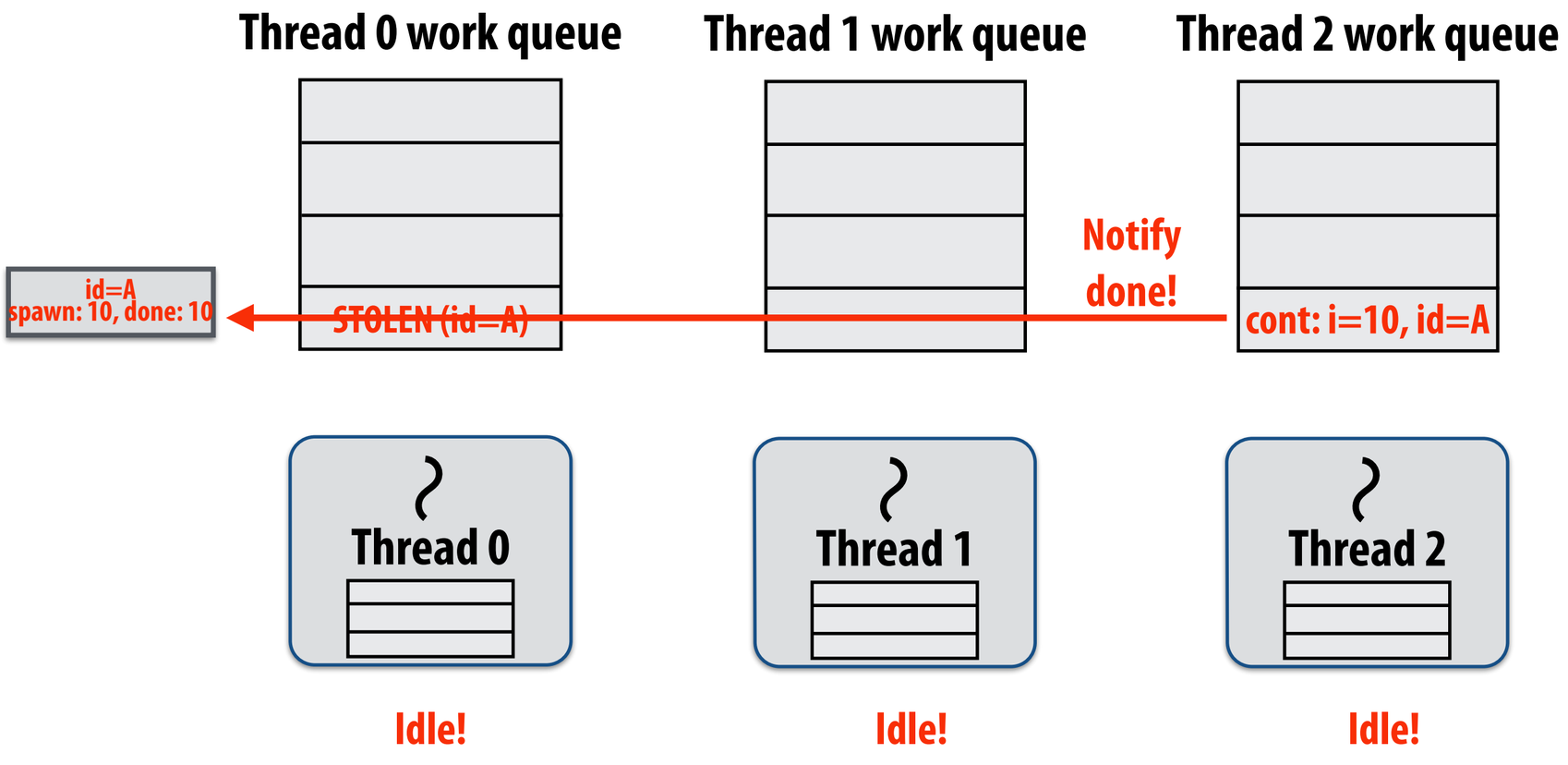
Implementing sync: case 2: stealing

block (id: A)

```
for (int i=0; i<10; i++) {
  cilk_spawn foo(i);
}
cilk_sync;
bar();
```

Sync for all calls spawned in block A





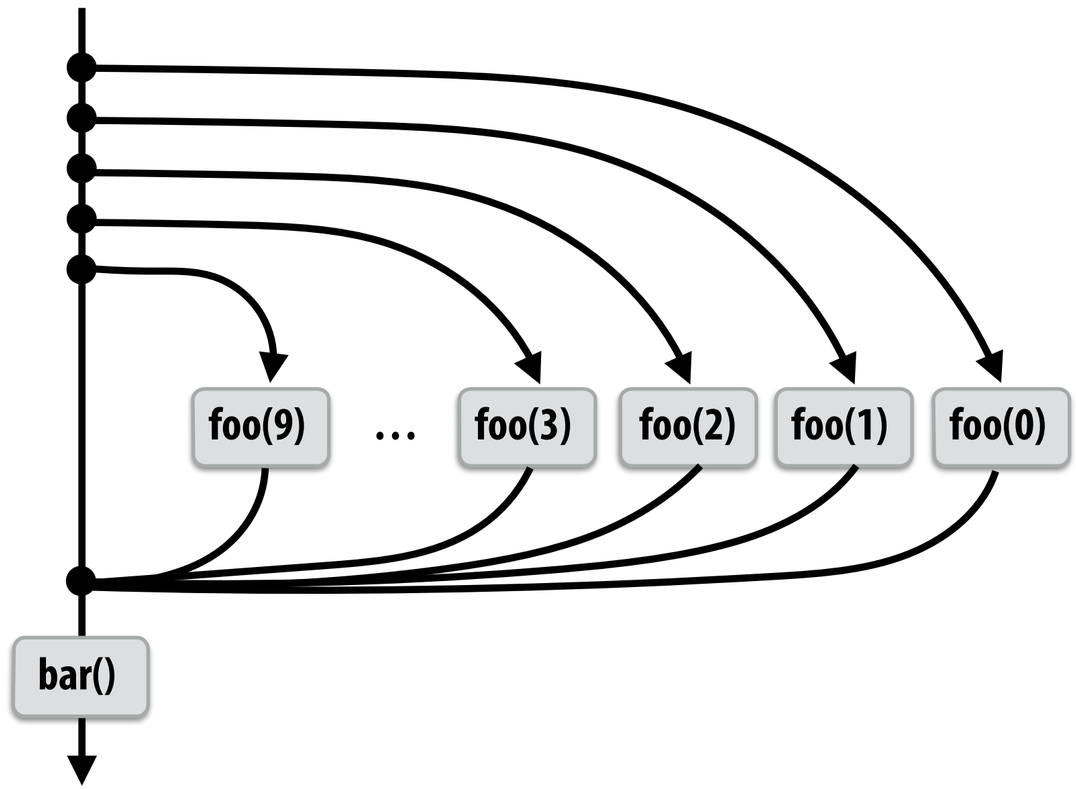
Last spawn completes.

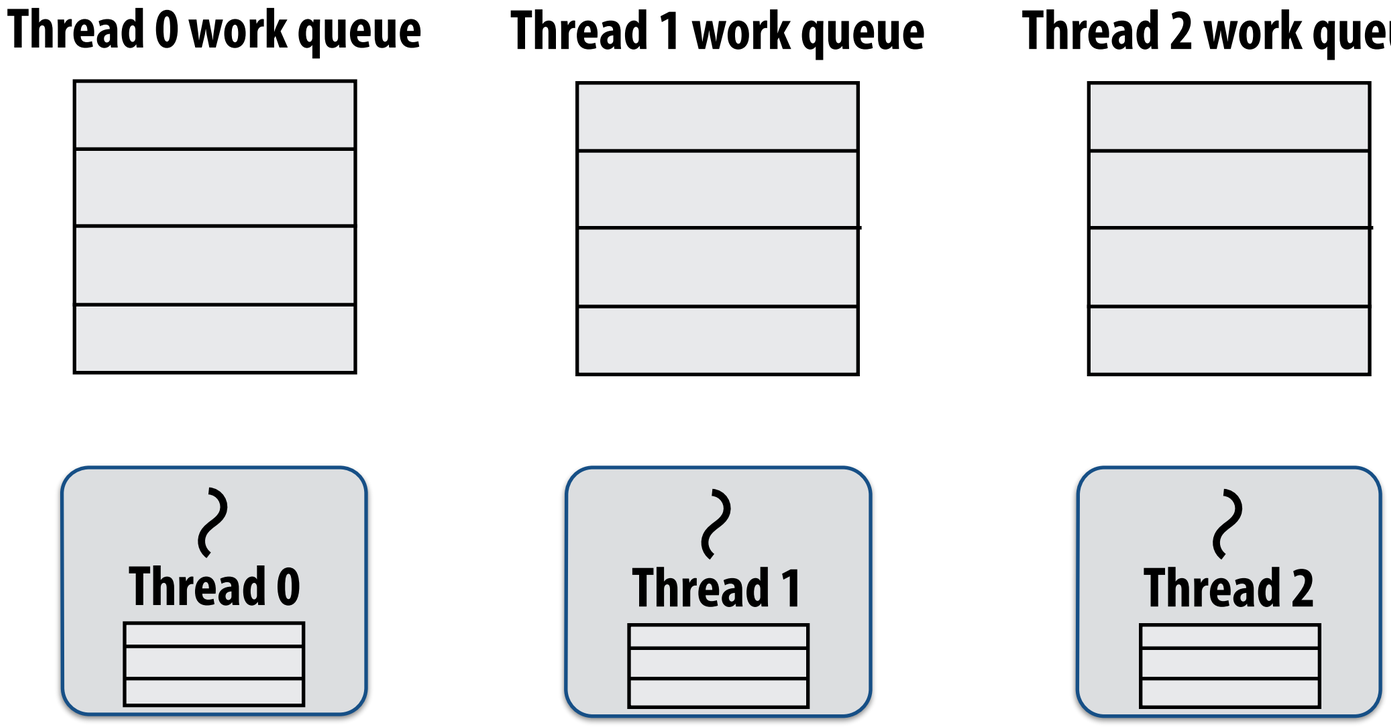
Implementing sync: case 2: stealing

block (id: A)

```
for (int i=0; i<10; i++) {
  cilk_spawn foo(i);
}
cilk_sync;
bar();
```

Sync for all calls spawned in block A





Thread 0 now resumes continuation and executes bar()

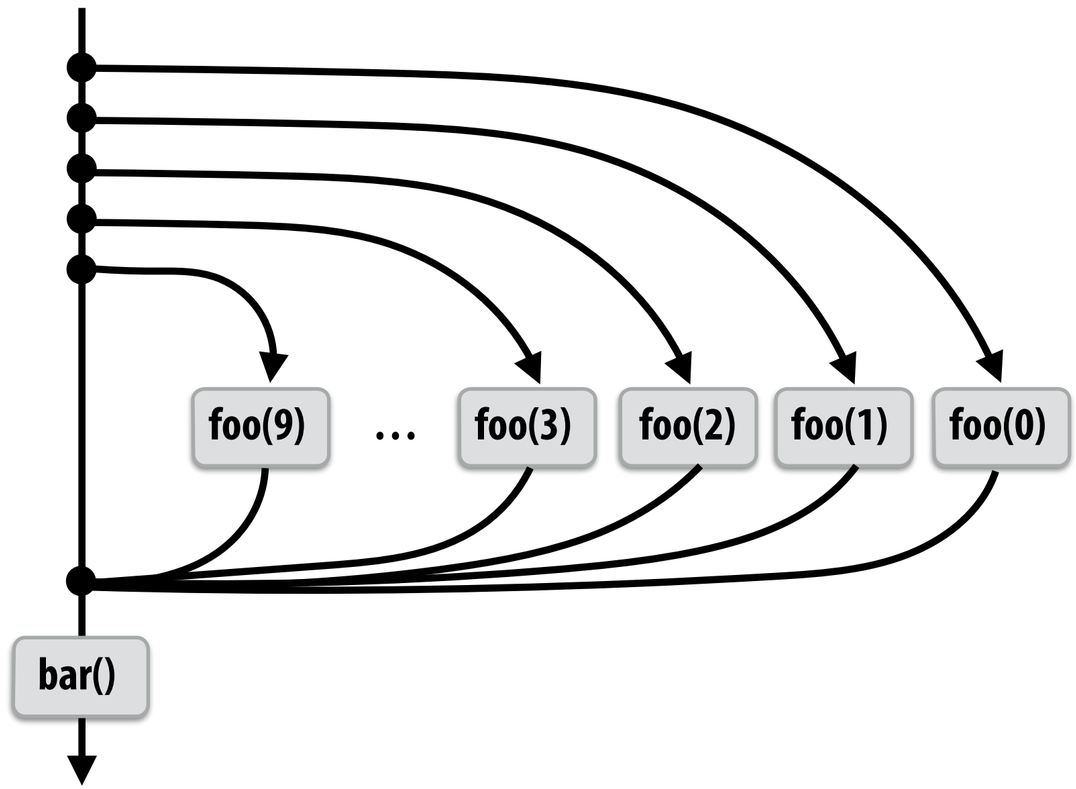
Note block A descriptor is now free.

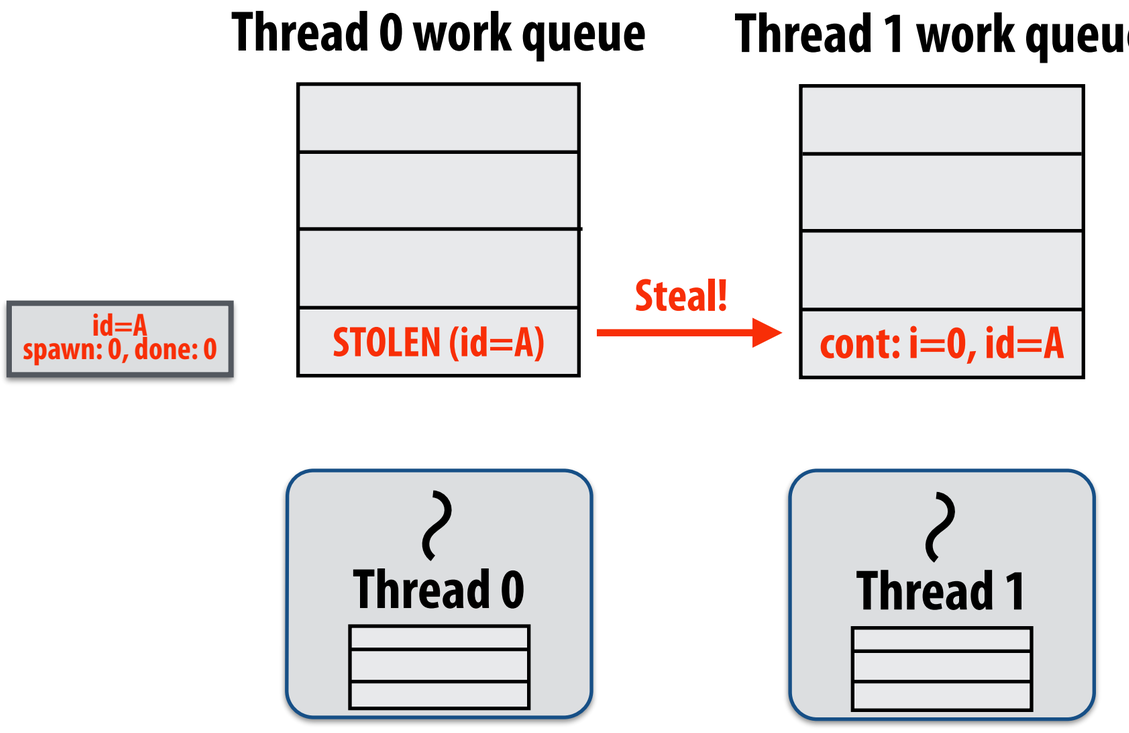
Implementing sync: case 2: stealing

block (id: A)

```
for (int i=0; i<10; i++) {
  cilk_spawn foo(i);
}
cilk_sync;
bar();
```

Sync for all calls spawned in block A





Example 2: "greedy" policy

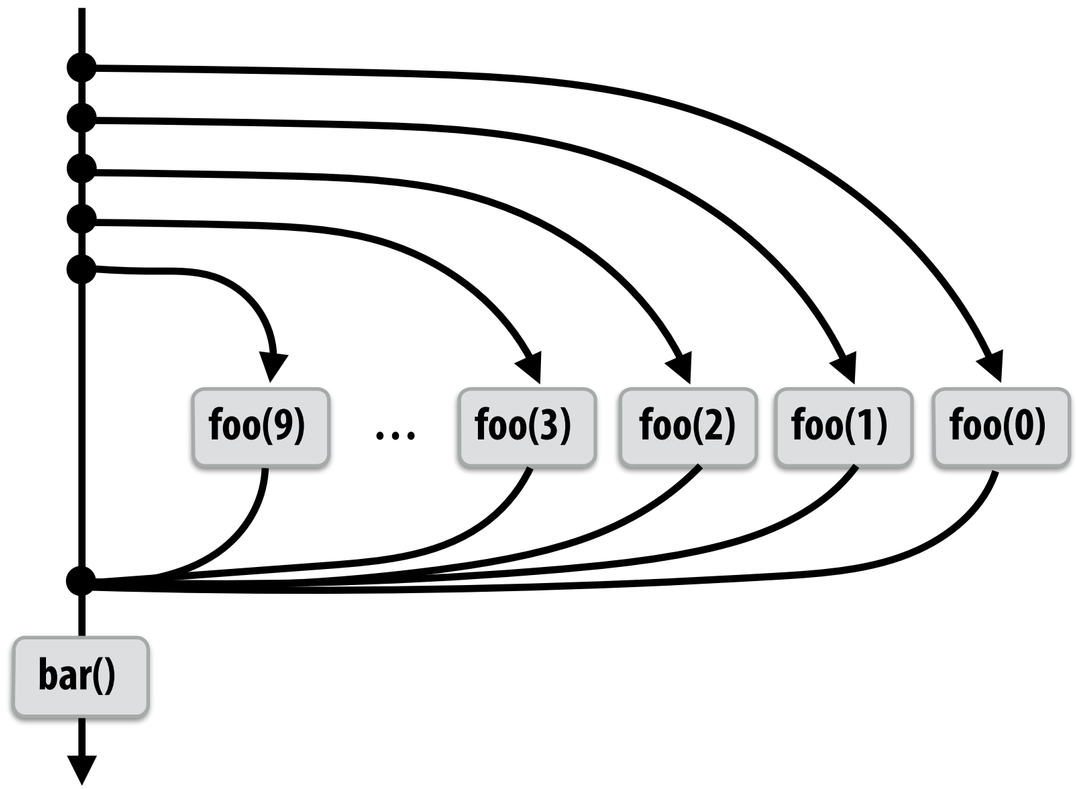
- When thread that initiates the fork goes idle, it looks to steal new work
- Last thread to reach the join point continues execution after sync

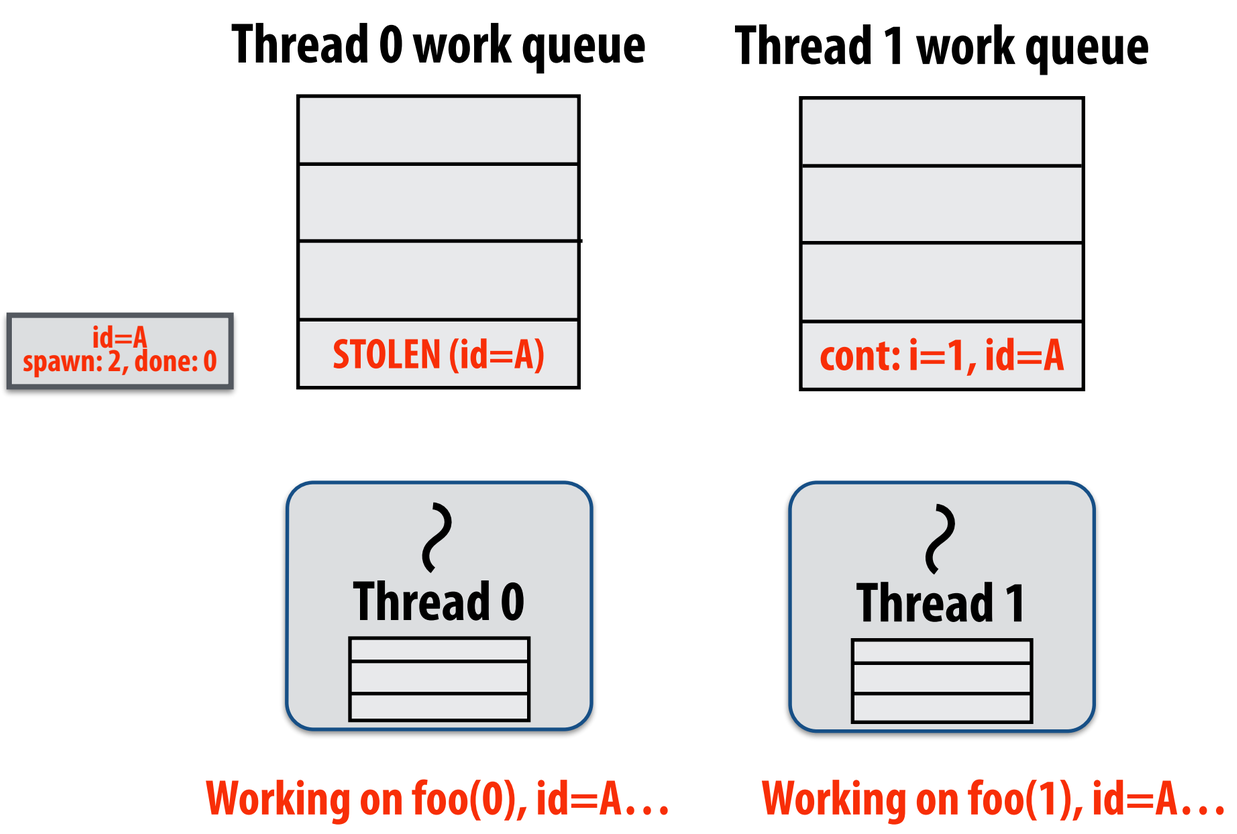
Implementing sync: case 2: stealing

block (id: A)

```
for (int i=0; i<10; i++) {
  cilk_spawn foo(i);
}
cilk_sync;
bar();
```

Sync for all calls spawned in block A





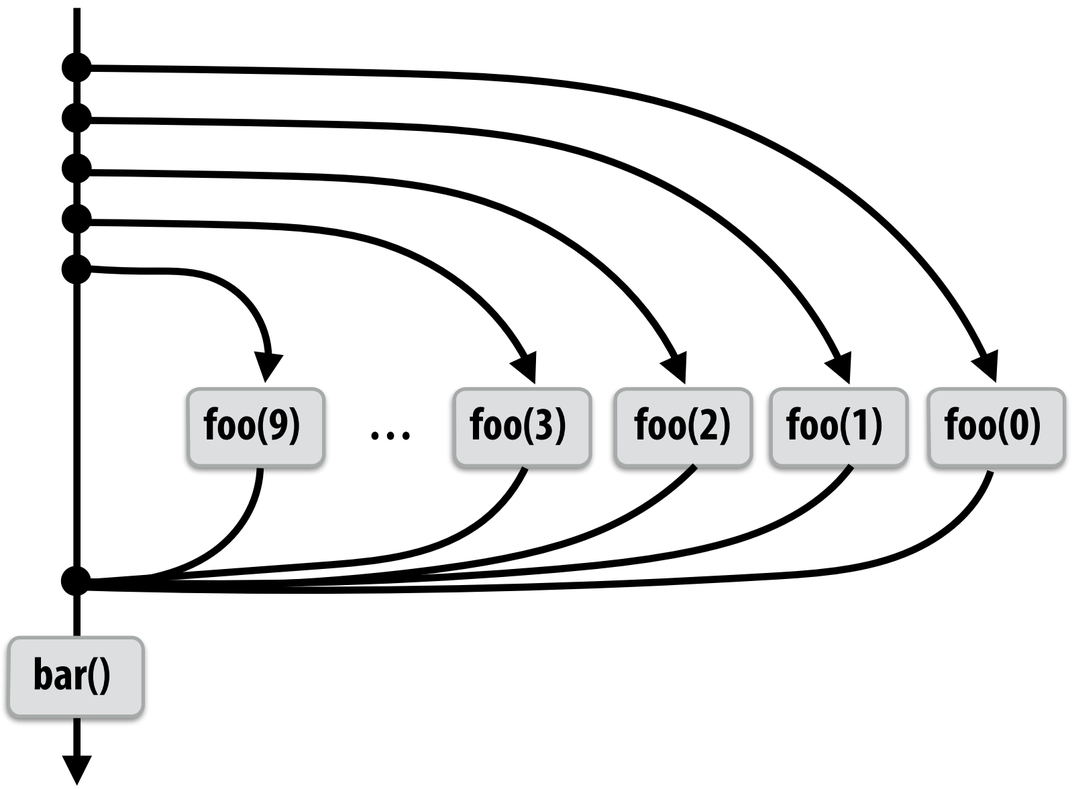
Idle thread 1 steals from busy thread 0 (as in the previous case)

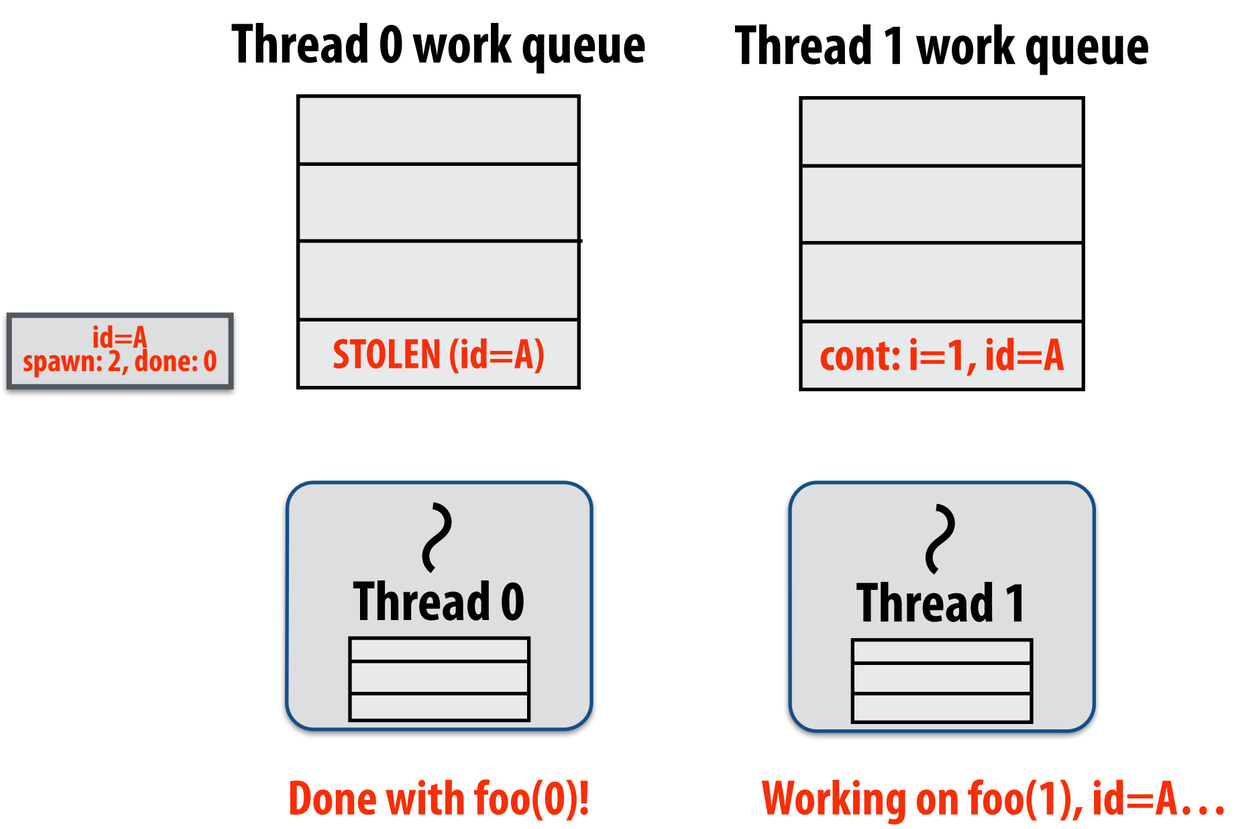
Implementing sync: case 2: stealing

block (id: A)

```
for (int i=0; i<10; i++) {
  cilk_spawn foo(i);
}
cilk_sync;
bar();
```

Sync for all calls spawned in block A





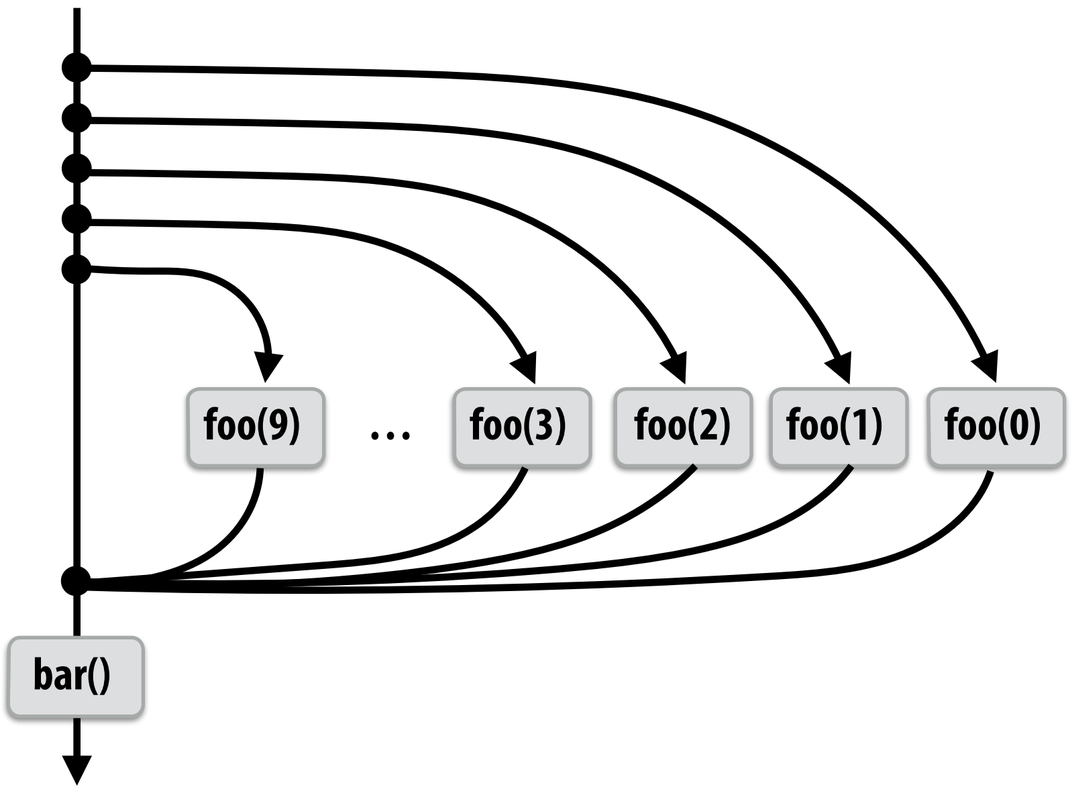
Thread 0 completes foo(0)
No work to do in local dequeue, so thread looks to steal!

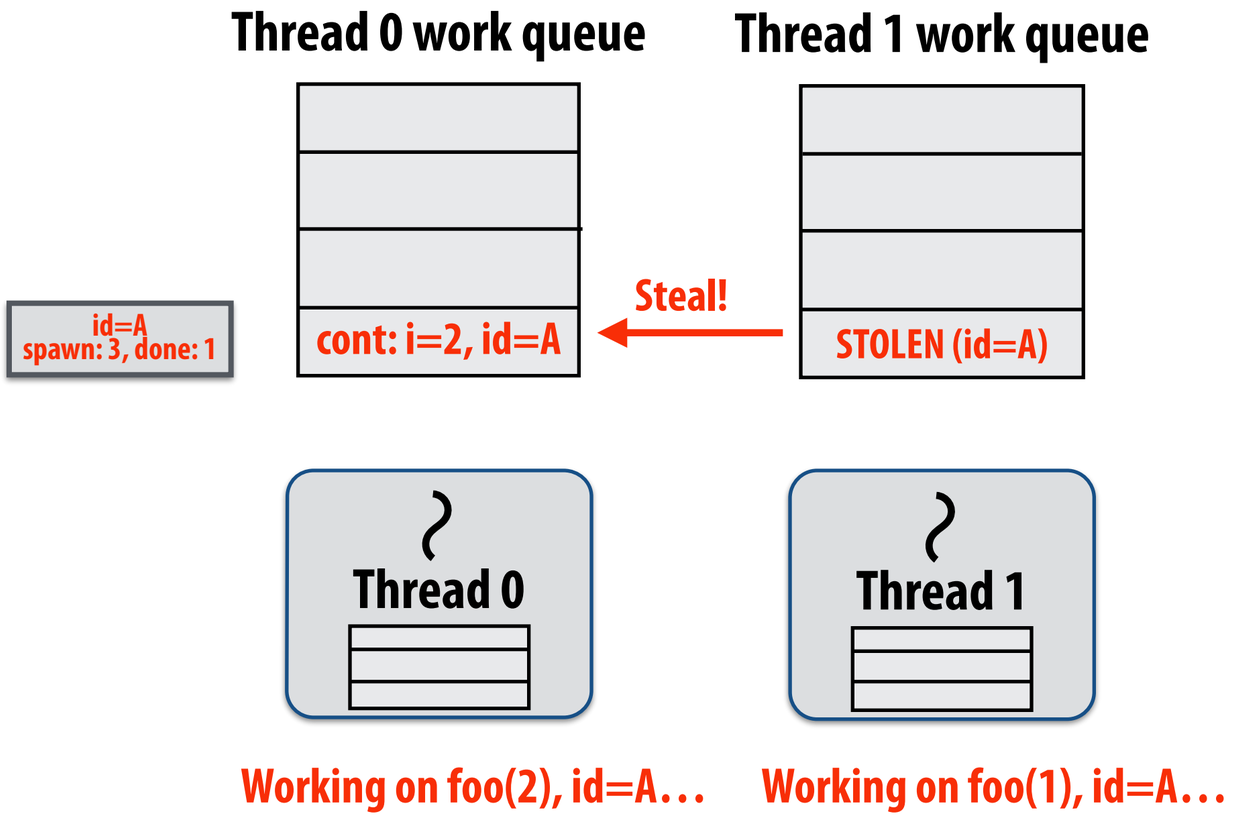
Implementing sync: case 2: stealing

block (id: A)

```
for (int i=0; i<10; i++) {
  cilk_spawn foo(i);
}
cilk_sync;
bar();
```

Sync for all calls spawned in block A





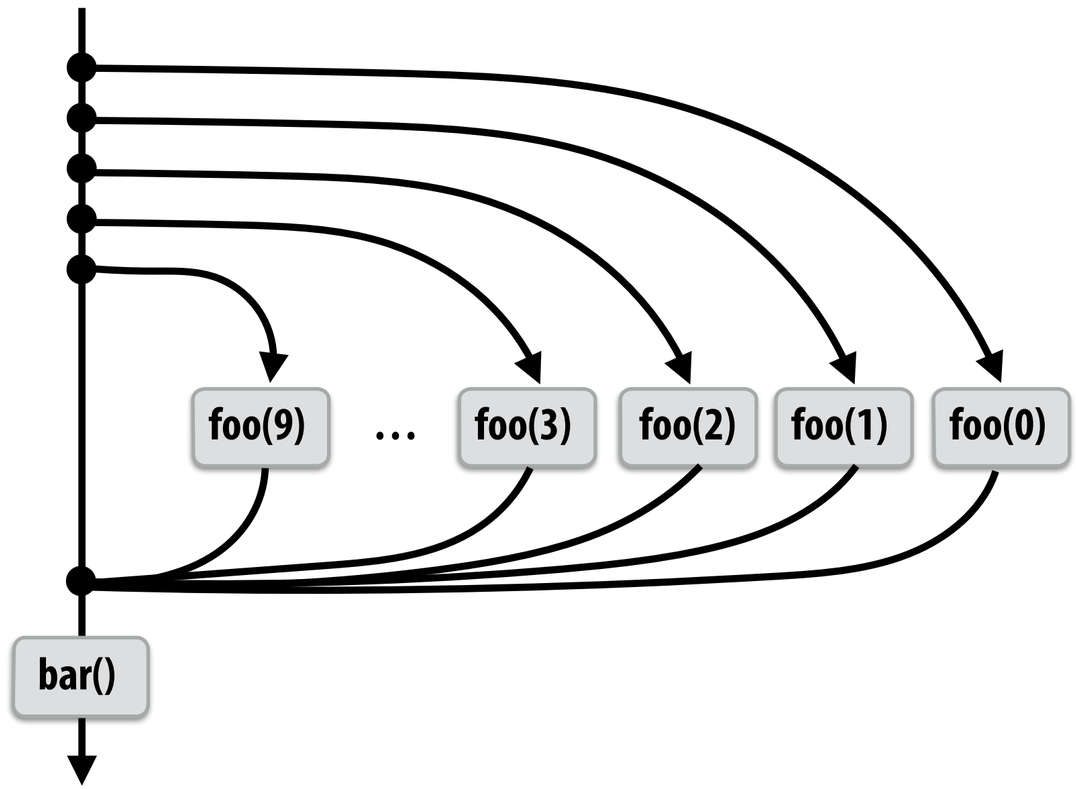
Thread 0 now working on foo(2)

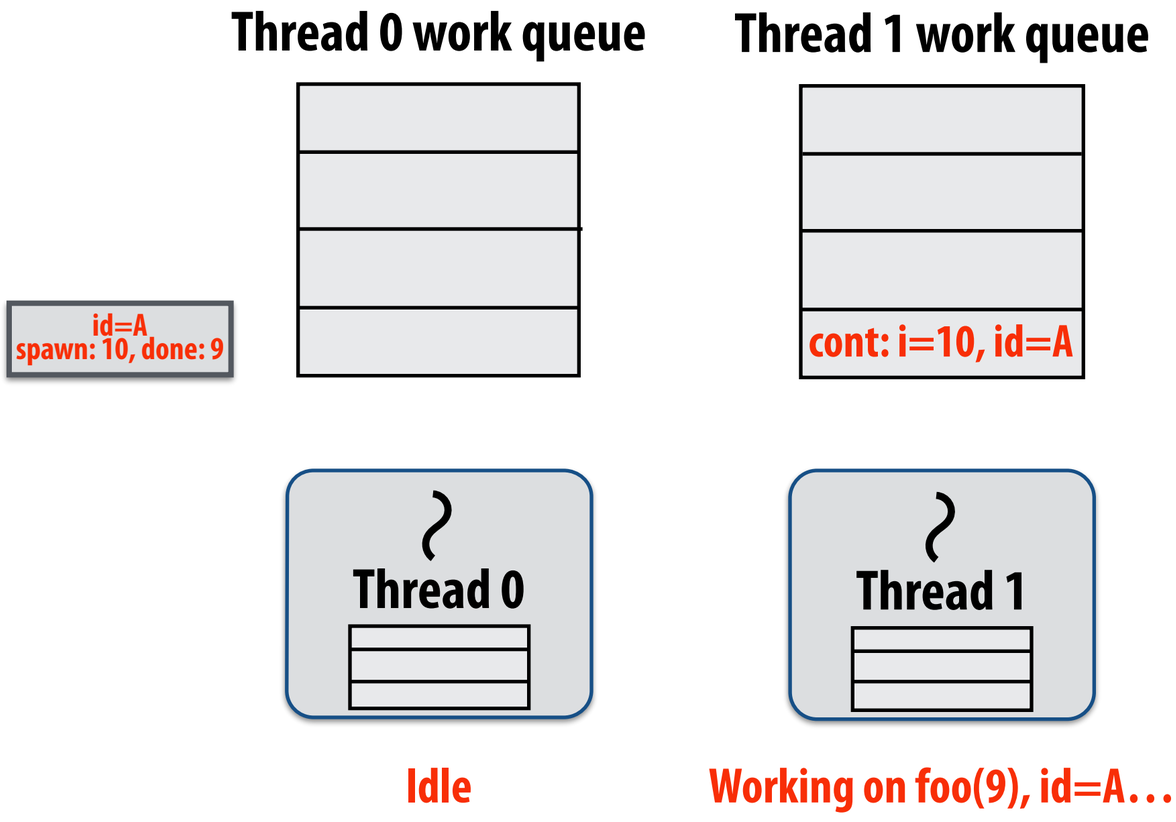
Implementing sync: case 2: stealing

block (id: A)

```
for (int i=0; i<10; i++) {
  cilk_spawn foo(i);
}
cilk_sync;
bar();
```

Sync for all calls spawned in block A





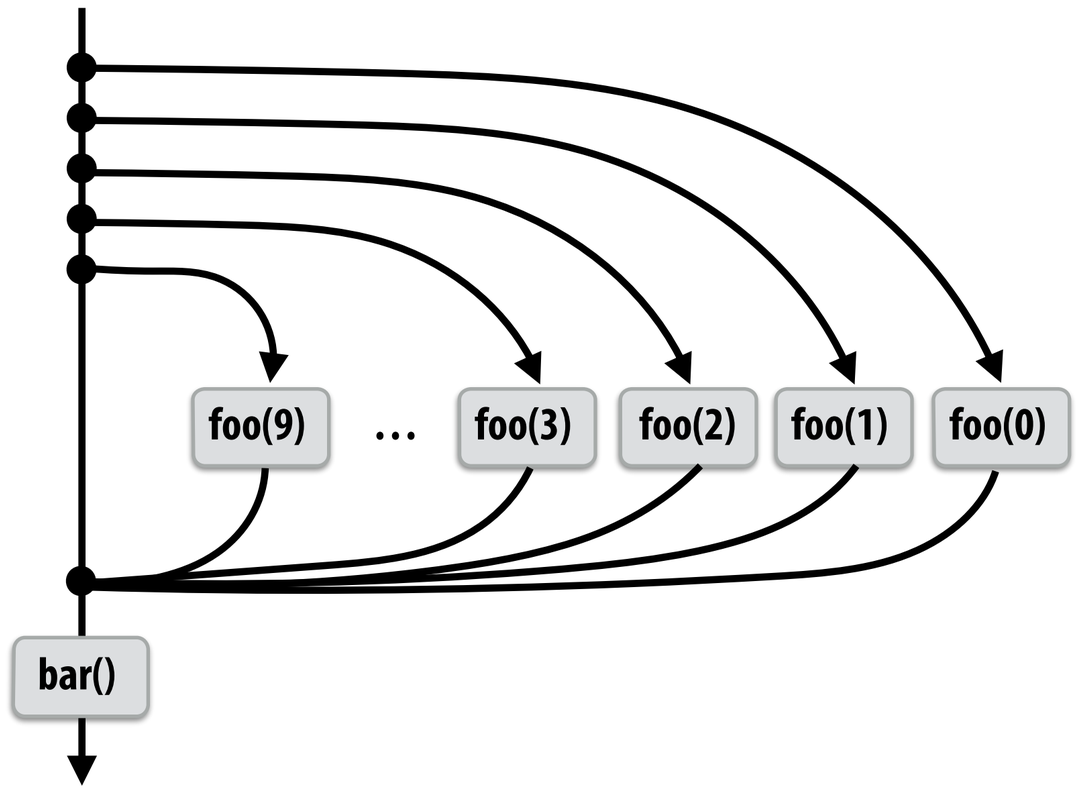
Thread 1 is last to finish spawned calls for block A.

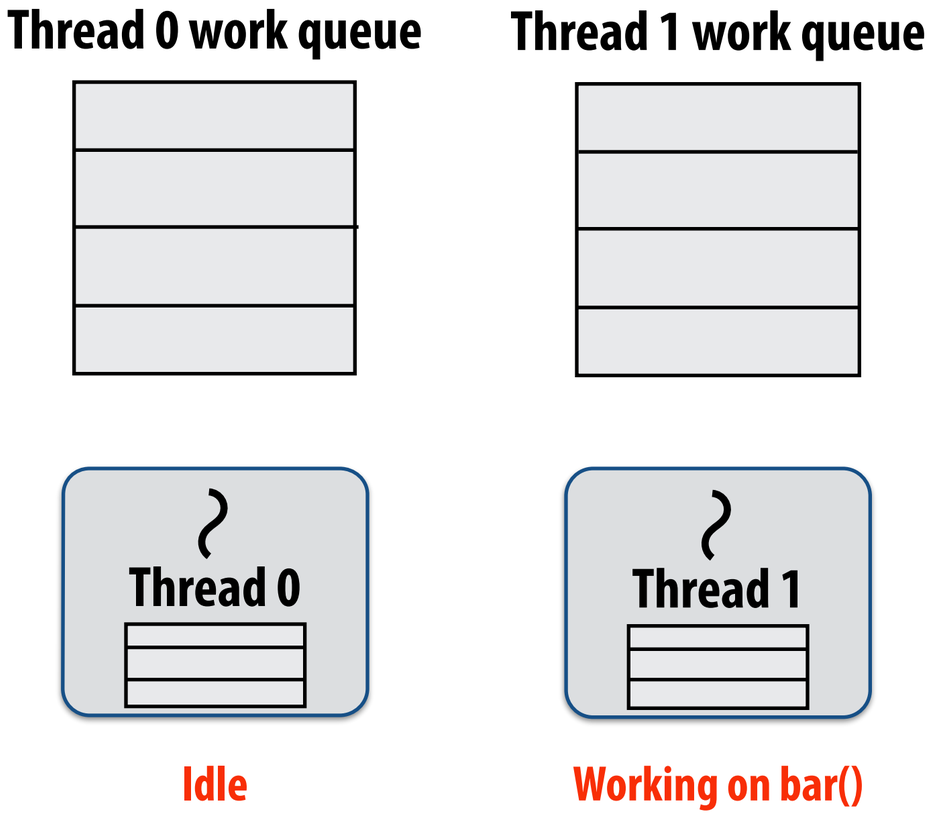
Implementing sync: case 2: stealing

block (id: A)

```
for (int i=0; i<10; i++) {
  cilk_spawn foo(i);
}
cilk_sync;
bar();
```

Sync for all calls spawned in block A





Thread 1 continues on to run bar()
Note block A descriptor is now free.

Cilk uses greedy join scheduling

- **Greedy join scheduling policy**
  - All threads always attempt to steal if there is nothing to do (thread only goes idle if no work to steal is present in system)
  - Worker thread that initiated spawn may not be thread that executes logic after cilk_sync
- **Remember:**
  - Overhead of bookkeeping steals and managing sync points only occurs when steals occur
  - If large pieces of work are stolen, this should occur infrequently
    - Most of the time, threads are pushing/popping local work from their local dequeue

Summary

- Fork-join parallelism: a natural way to express divide-and-conquer algorithms
  - Discussed Cilk Plus, but OpenMP also has fork/join primitives

- Cilk Plus runtime implements spawn/sync abstraction with locality-aware work stealing scheduler
  - Always run spawned child (continuation stealing)
  - Greedy behavior at join (threads do not wait at join, immediately look for other work to steal)