L04-1
并行编程基础
—负载平衡

《并行处理》

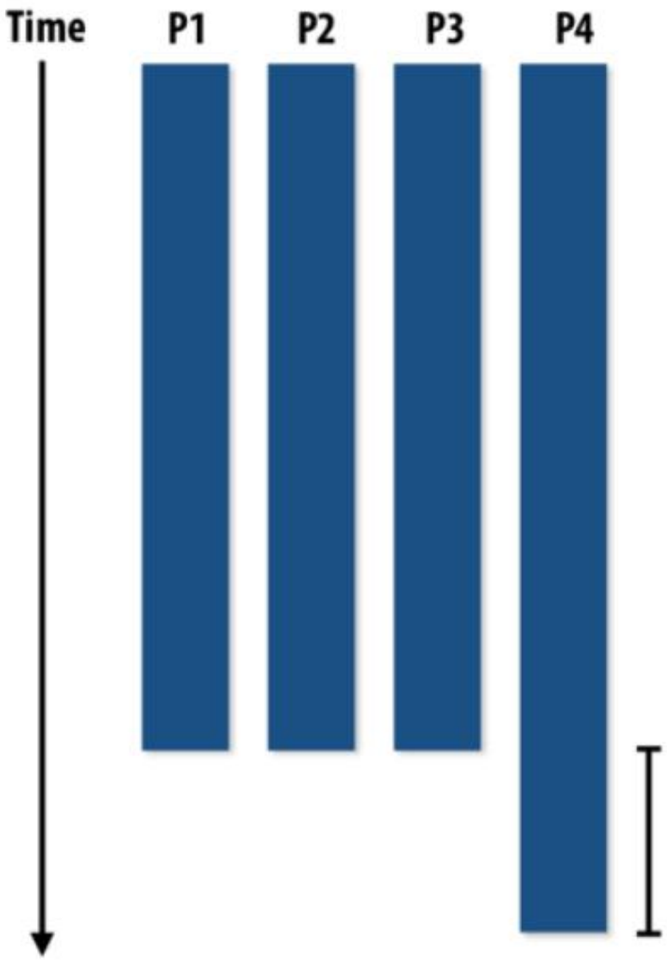
邵恩
高性能计算机研究中心

面向高性能计算的编程模型

- 优化并行程序的性能是一个**反复迭代**且复杂的代码改造过程（问题分解、任务分配和资源编排）
- 并行程序优化的三个关键目标（**相互矛盾的**）
  - 保证计算任务在可用的计算资源上的负载平衡(Load-balance)
  - 减少通信 (避免计算停顿)
  - 减少为提高并行性、资源分配、减少通信等而带来的额外开销

负载平衡（Load-balance）

Ideally：所有的处理器在程序执行过程中一直在计算（它们同时计算，它们同时完成自己的部分工作）



回忆阿姆达尔定律：
即使只有少量的负载不平衡，就可以显着限制最大加速比（并行加速比）

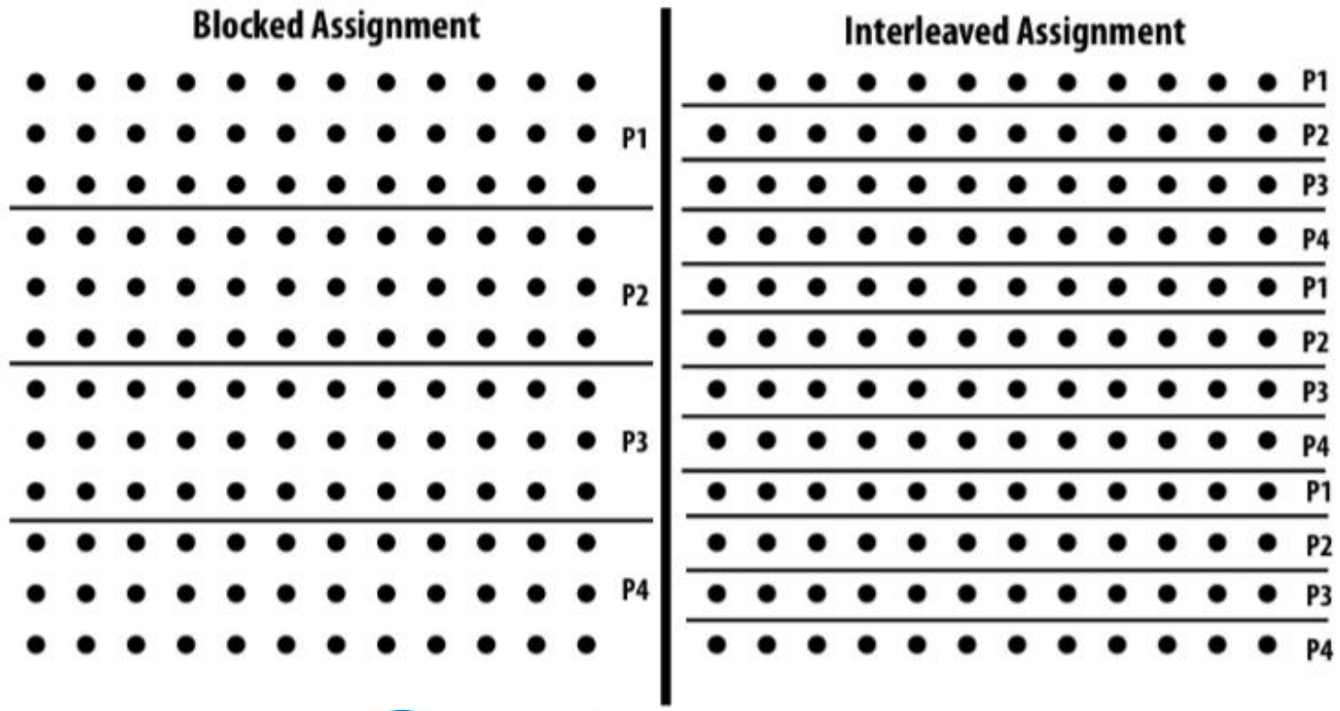
P4 多做 20% 的工作 -> P4 需要额外 20% 的时间才能完成
-> 并行程序运行时间的 20% 是串行执行

(此处无法并行的计算任务，约占整个程序工作量的5%)

静态分配(Static assignment)

- 线程的工作分配是预先确定的
  - 不一定在编译时确定（赋值算法可能取决于运行时参数，如输入数据大小、线程数等）
- 为每个线程（工作者）分配相等数量的网格单元（工作）
  - 我们讨论了两种静态分配: 阻塞（blocked ）and 交错（interleaved）
- 静态赋值的良好特性：简单，基本上为零运行时开销

阻塞式静态分配（blocked ）　交错式静态分配（interleaved）



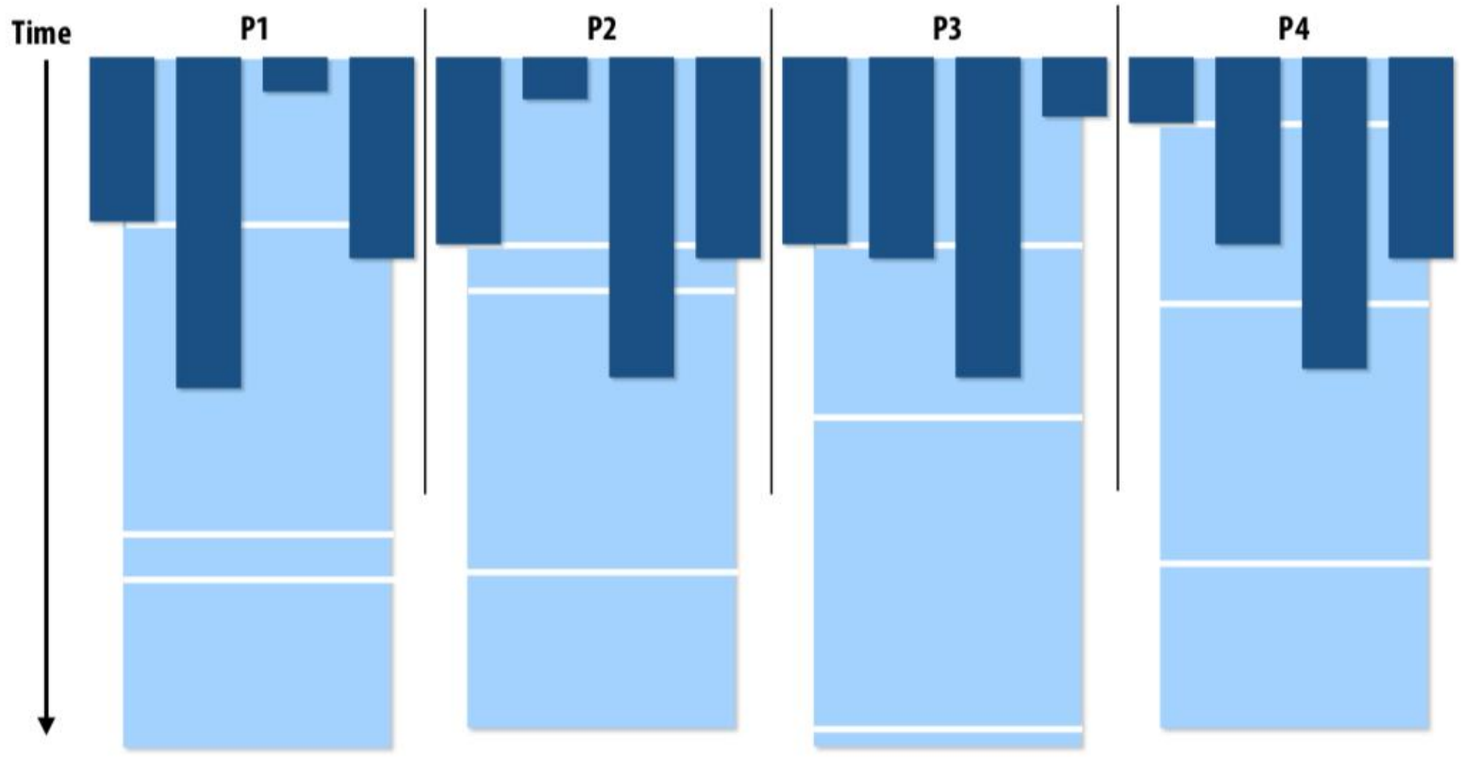
静态赋值什么时候适用?

- 当工作的**成本（执行时间）和工作量**是可预测的（这样程序员就可以提前制定好分配）
- 最简单的例子：预先知道所有子计算任务的时间耗时成本相同

在上面的例子中：
有 12 个任务，已知每个计算任务的成本相同。
分配方案：静态分配三个任务给四个处理器

静态赋值什么时候适用?

- 当工作是可预测的，但并非所有工作的成本都相同时
- 当有关执行时间的统计数据已知时（例如，平均成本相同）

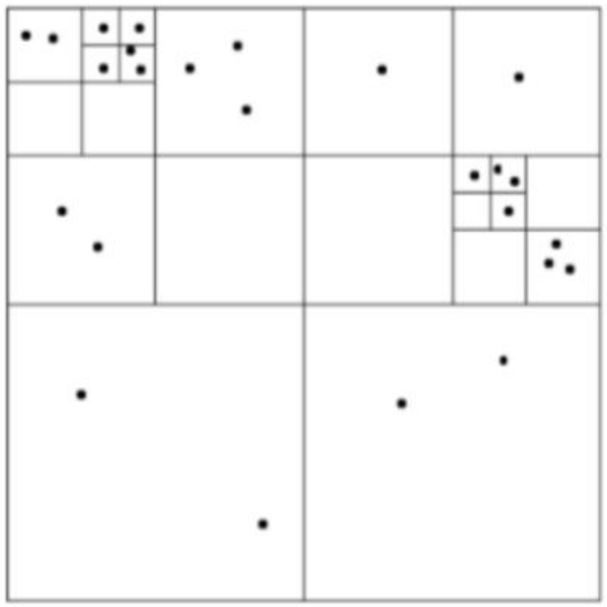


作业具有不平等但已知的成本：分配给处理器以确保整体良好的负载平衡四个处理，每个处理器串行执行四个计算任务（因为是串行执行的，所以每个处理器的总计算时间T是相同的）

“半静态” “Semi-static” 分配

- 短期内的工作成本是可预测的
  - Idea: 用近期完成刚刚完成任务的执行时间，预测邻近的短期任务的执行时间
- 应用程序定期进行自我分析并重新调整分配
  - 重新调整之间的间隔分配是静态的

N 体模拟



N-Body simulation:

在模拟过程中移动时重新分配处理器（如果运动缓慢，则不需要经常发生重新分配）

自适应网格

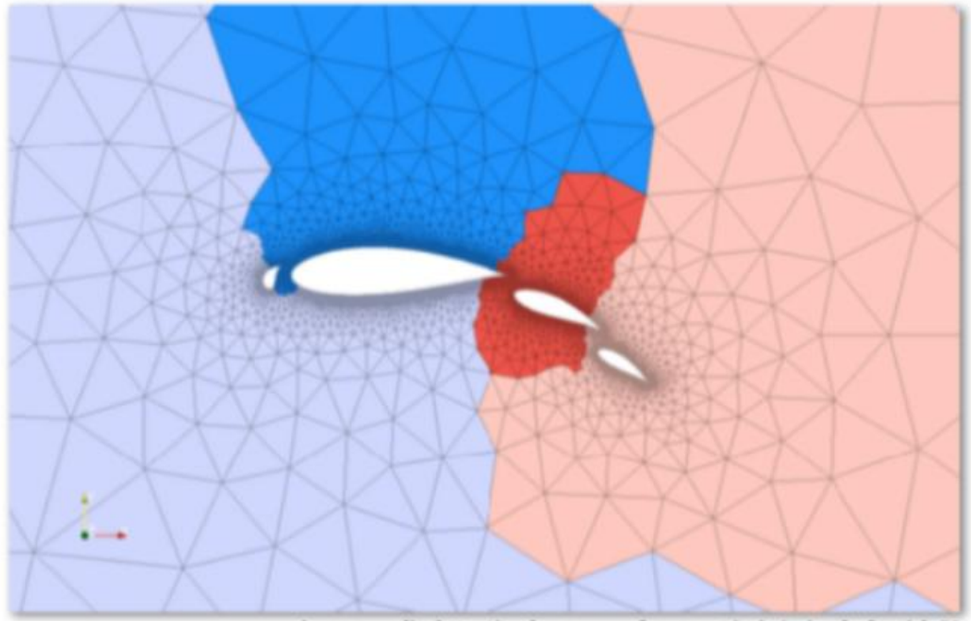


Image credit: http://typhon.sourceforge.net/spip/spip.php?article22

Adaptive mesh:

网格随着对象移动或流过对象变化而变化，但变化发生得很慢（颜色表示将部分网格分配给不同的处理器）

动态分配

- 随程序的运行和执行时，动态确定分配处理器，以确保负载平衡。（任务的执行时间，或者说任务总数，是不可预测的）

顺序程序（独立循环迭代）

```
int N = 1024;
int* x = new int[N];
bool* prime = new bool[N];

// initialize elements of x here

for (int i=0; i<N; i++)
{
    // unknown execution time
    is_prime[i] = test_primality(x[i]);
}
```

不可预测

并行程序
(SPMD通过多线程共享地址空间模型执行)

```
int N = 1024;
// assume allocations are only executed by 1 thread
int* x = new int[N];
bool* is_prime = new bool[N];

// initialize elements of x here

LOCK counter_lock;
int counter = 0;    // shared variable

while (1) {
  int i;
  lock(counter_lock);
  i = counter++;
  unlock(counter_lock);
  if (i >= N)
     break;
  is_prime[i] = test_primality(x[i]);
}
```

不可预测

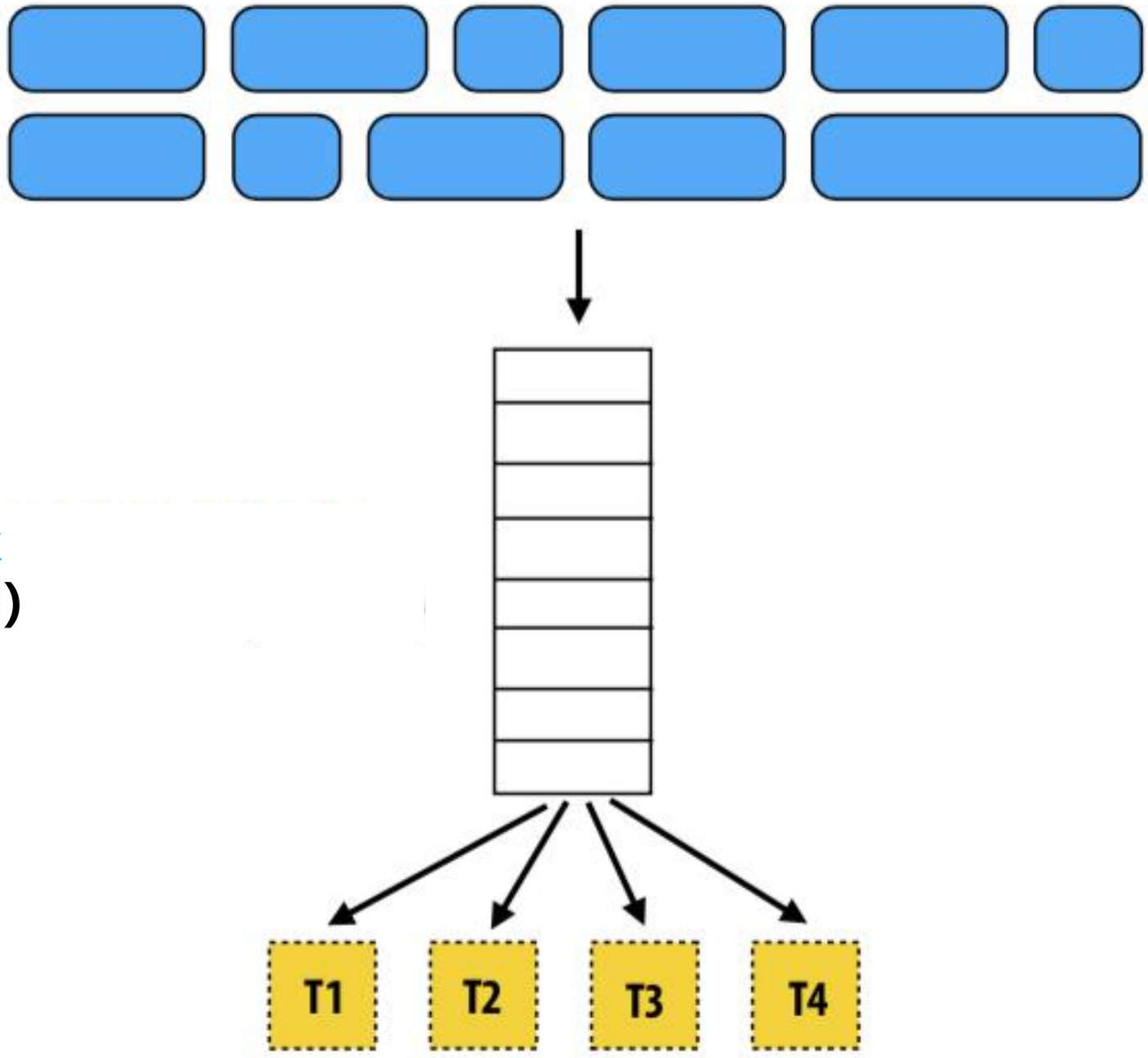
atomic_incr(counter);

使用队列(Queue)进行动态分配

问题分解（分解子任务）

Sub-problems

(a.k.a. "tasks", "work")



共享工作队列：一组要做的work

（我们假设每项work都是独立的）

Work线程：

从共享work队列中提取数据

在创建新work时将其推送到队列

每个work由什么构成?

- 这个实现有什么潜在问题?

```
const int N = 1024;
// assume allocations are only executed by 1 thread
float* x = new float[N];
bool* prime = new bool[N];

// initialize elements of x here

LOCK counter_lock;
int counter = 0;

while (1) {
  int i;
  lock(counter_lock);
  i = counter++;
  unlock(counter_lock);
  if (i >= N)
     break;
  is_prime[i] = test_primality(x[i]);
}
```

- **这是串行程序中不存在的开销**
- **而且..它是串行执行（回忆阿姆达尔定律）**

细粒度划分：1个work对应处理1个数据元素

可能良好的负载平衡（许多小任务，small tasks）
潜在的高同步成本
（在临界区要通过序列化，串行执行，降低同步开销）

提高任务粒度

```
const int N = 1024;
const int GRANULARITY = 10;
// assume allocations are only executed by 1 thread

float* x = new float[N];
bool* prime = new bool[N];

// initialize elements of x here

LOCK counter_lock;
int counter = 0;

while (1) {
  int i;
  lock(counter_lock);
  i = counter;
  counter += GRANULARITY;
  unlock(counter_lock);
  if (i >= N)
     break;
  int end = min(i + GRANULARITY, N);
  for (int j=i; j<end; j++)
     is_prime[j] = test_primality(x[j]);
}
```

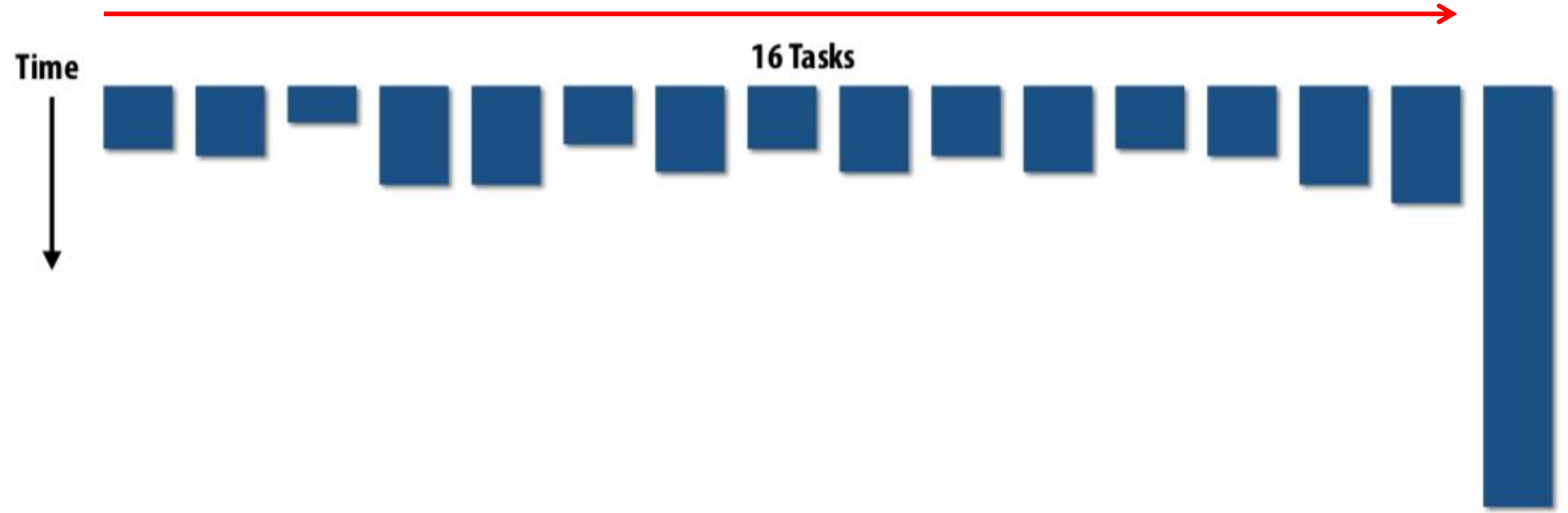
- 粗粒度划分：1个任务处理10个元素
- 降低同步成本（进入临界区的次数减少 10 次）

选取task的大小

- 比处理器有更多的任务很有用（许多小任务通过动态分配实现良好的工作负载平衡
  - 细粒度划分子任务（每个任务处理的数据量少，任务数量更多)
- 但是想要减少任务数量，最小化管理分配的开销（如：同步)
  - 粗粒度划分子任务（每个任务处理的数据量多，任务数量更少)
- 理想的粒度取决于许多因素（共同主题：必须了解您的工作负载和您的机器)

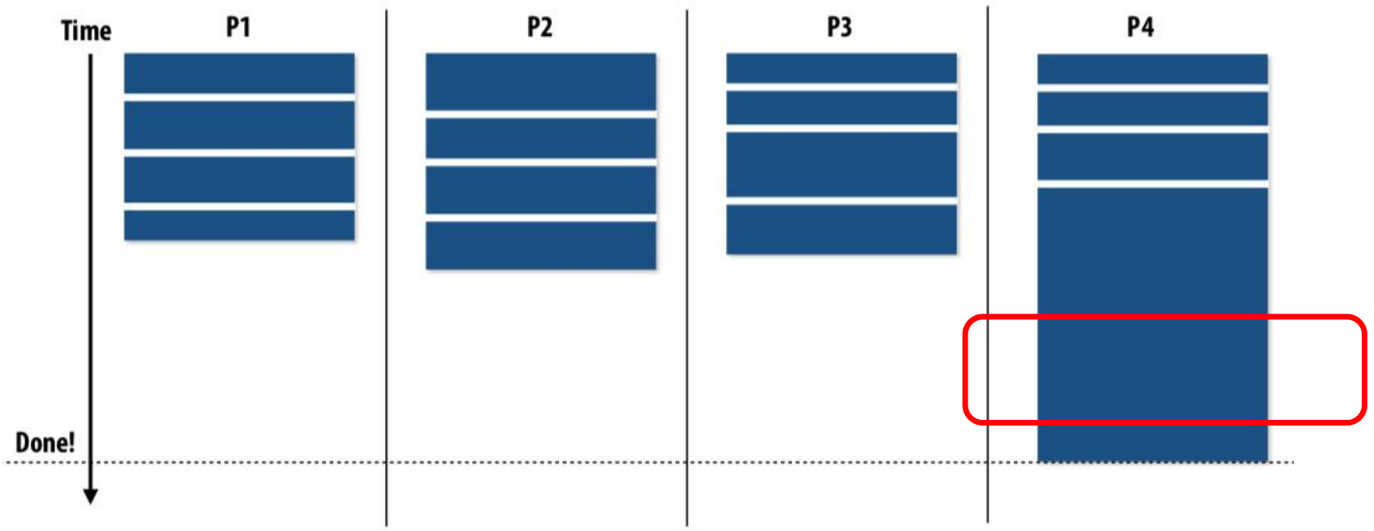
更智能的任务调度

- 通过共享工作队列进行动态调度
- 如果系统按从左到右的顺序将这些任务分配给worker，会发生什么？



更智能的任务调度

- 如果调度程序最后运行长任务会怎样?
  - 会出现潜在的负载不平衡！ (load imbalance!)

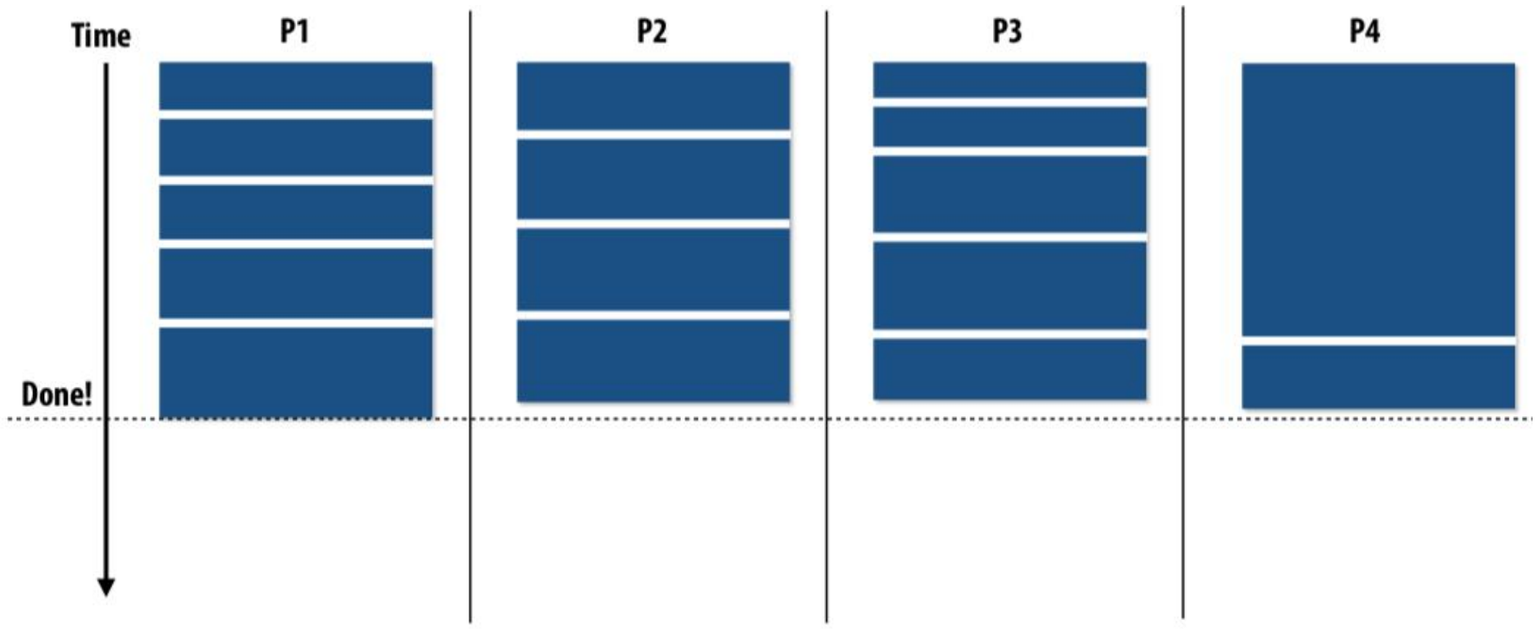


“长杆task”
(long pole task)

- 不平衡问题的一种可能解决方案
- 将工作分成更多的小任务
  - 相对于整体执行时间，希望“长杆task”的执行时间变短
  - 可能会增加同步开销（用不同处理器执行）
  - 也可能不增加开销，因为把长杆task切小后，无法并行到不同处理器上，还是在相同的处理器上串行执行 (长杆任务基本上是顺序的)

更智能的任务调度

- 安排长杆task优先执行，以减少计算结束时的“溢出”



- 另一个解决方案：更智能的调度
- 安排执行时间长的任务优先执行
  - 长时间执行的线程数量较少，但工作量与其他线程的计算量大致相同
  - 需要对各个task的执行时间具有可预测性，评估各个task的执行时间

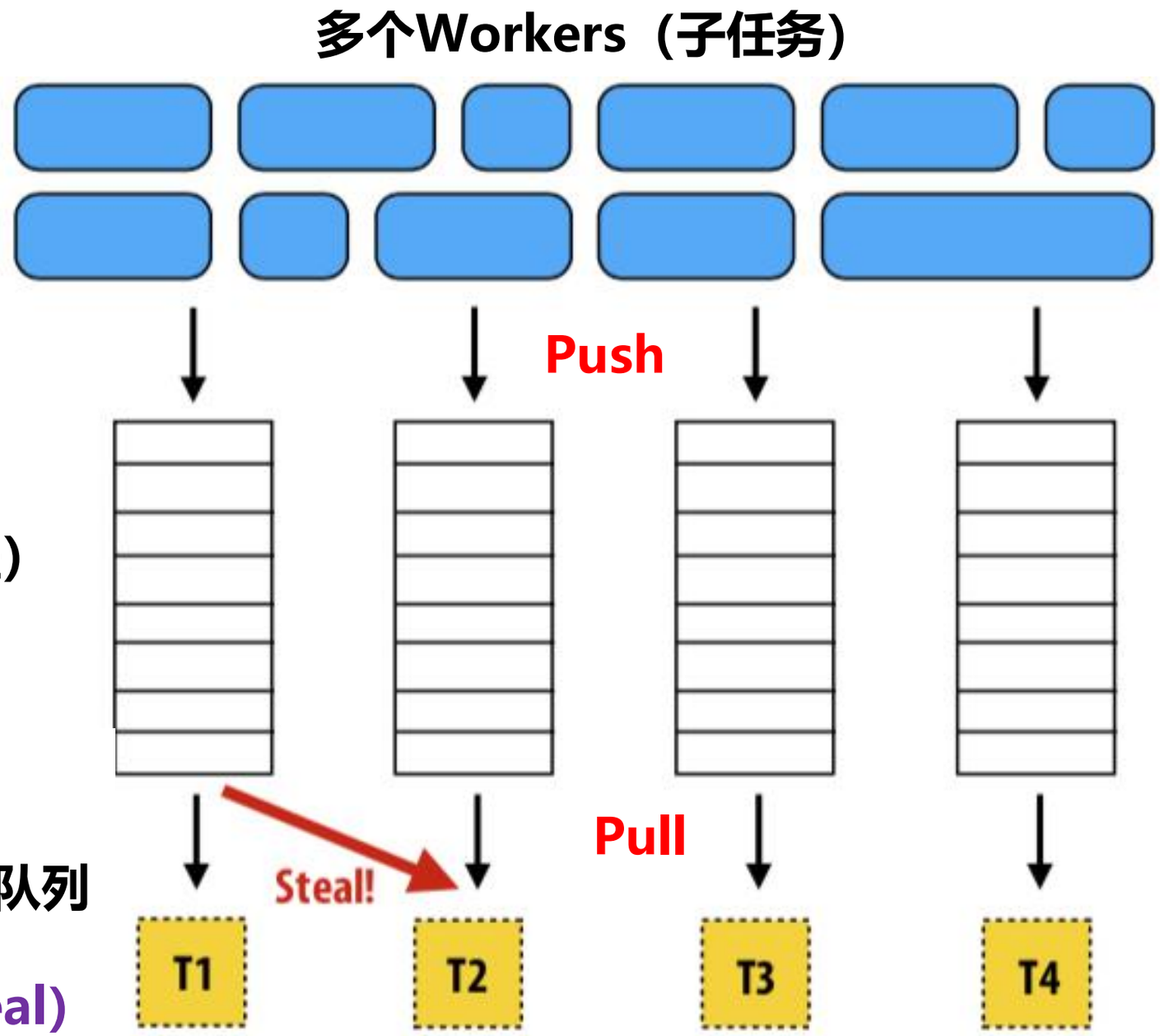
减少同步开销

- 避免在单个工作队列上的各个worker间进行同步，所以考虑使用多个独立队列

问题分解（分解子任务）

Subproblems

(a.k.a. "tasks", "work to do")



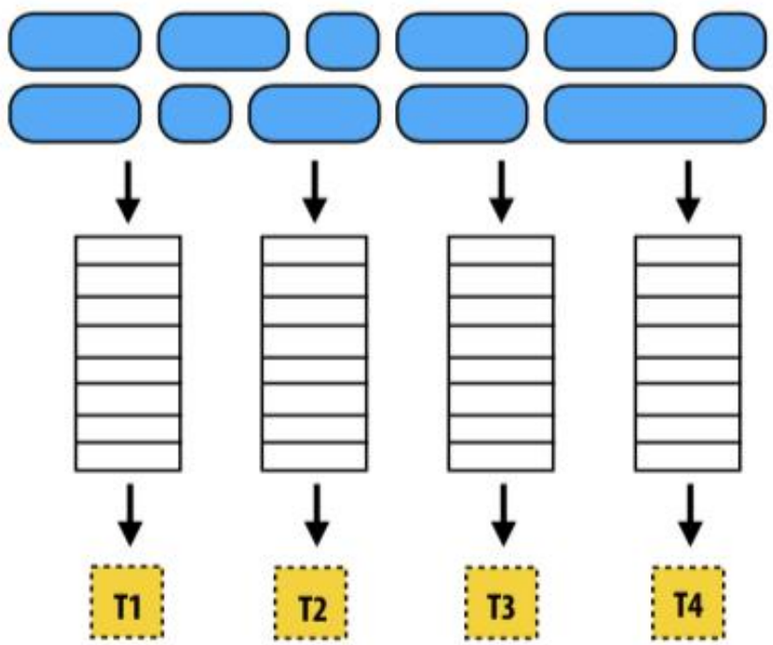
一组工作队列
(一般来说，一个队列对应一个线程)

工作线程：

- **从工作队列中拉取(Pull)数据**
- **将新工作推送(Push)到 0WN 工作队列**
- **当本地工作队列为空时...**
- **从另一个工作队列中"窃取" (Steal)工作**

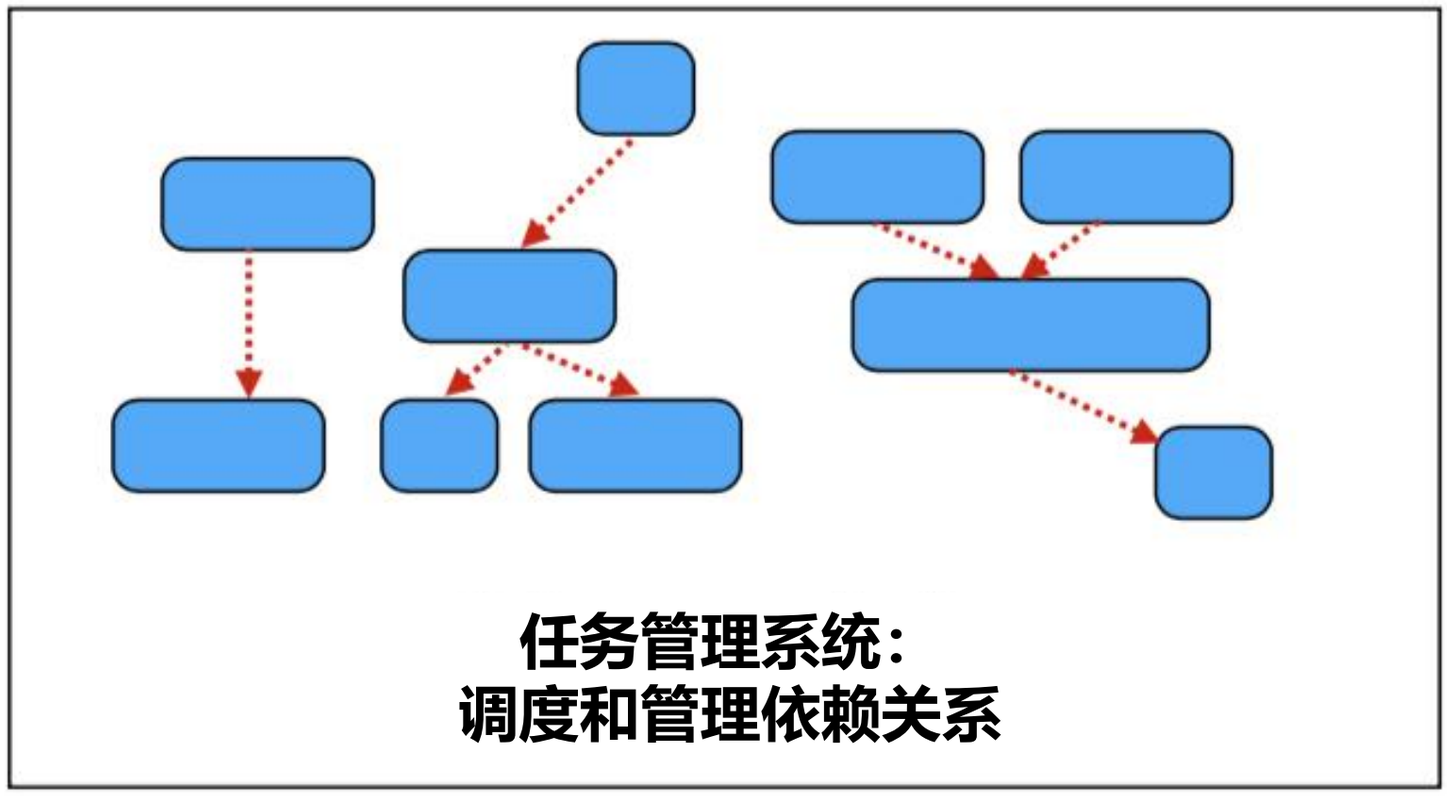
分布式工作队列

- “窃取”期间发生昂贵的同步/通信
  - 但并非每次线程窃取新工作时都会有昂贵的开销
  - 窃取仅在必要时发生以确保良好的负载平衡
- 导致局部性增加
  - 常见情况：线程处理它们自己创建的任务（生产者-消费者局部性）
- 实施挑战
  - 从谁哪里偷？/要偷多少？
  - 如何检测程序终止？
  - 确保本地队列访问速度快（同时保留互斥）



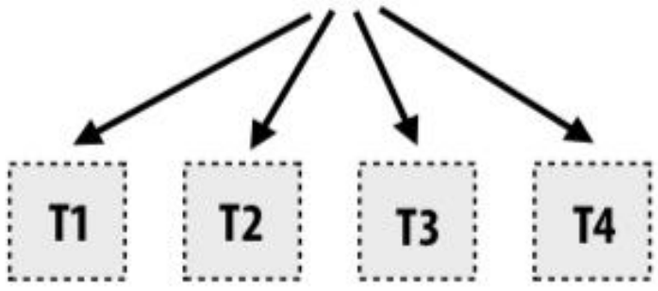
任务队列中的工作彼此并不独立（有相互依赖）

= **依赖关系**



任务管理系统：
调度和管理依赖关系

在满足所有任务依赖性之前，任务不会从队列中出队并分配给工作线程



worker可以向任务系统提交新任务（带有可选的显式依赖项）

```
foo_handle = enqueue_task(foo);                 // enqueue task foo (independent of all prior tasks)
bar_handle = enqueue_task(bar, foo_handle);     // enqueue task bar, cannot run until foo is complete
```

总结

- 挑战：如何实现良好的工作负载平衡
- 希望所有处理器一直工作（否则资源空闲）
- 但想要实现这种平衡的低开销（ low-cost ）解决方案
    - 最小化计算任务间的管理开销（例如，调度/分配逻辑）
    - 最小化同步成本
- 静态分配与动态分配
    - 这不是一个非此即彼的决定，而是需要经过一系列思考的多次选择
    - 尽可能使用有关工作负载的先验知识，以减少负载不平衡和任务管理/同步成本
        - 在极限情况下，如果系统知道一切，就使用完全静态分配
- 今天讨论的问题涵盖问题分解（ decomposition），任务分配（assignment），以及资源调度编排（orchestration）

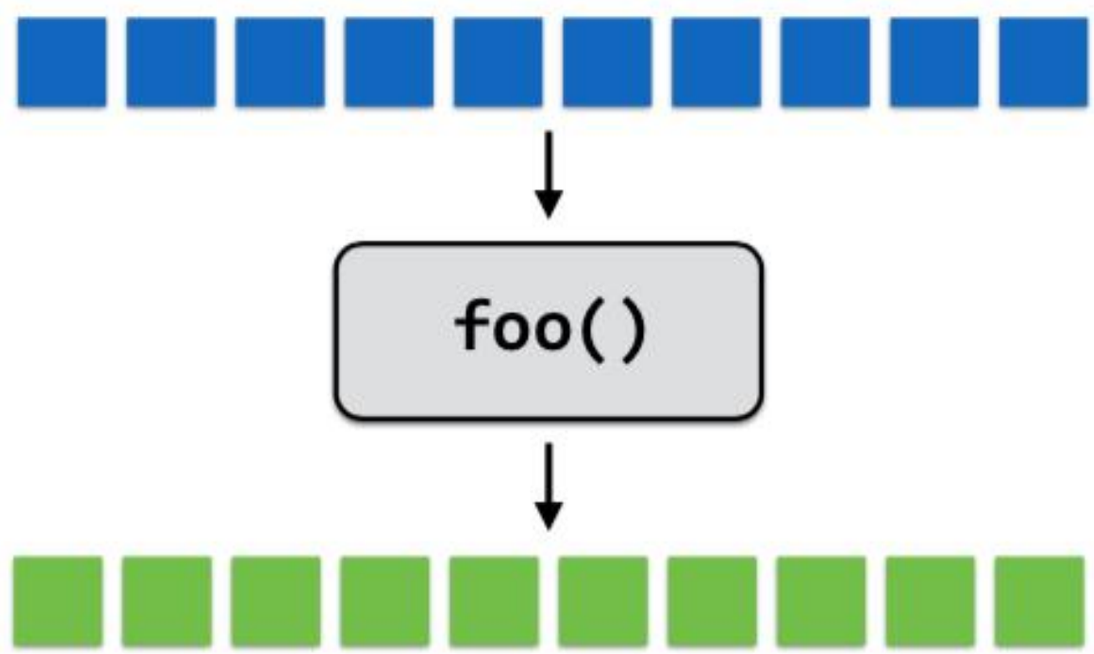
fork-join 的并行性调度

常见的并行编程模式

- 对许多数据元素执行**相同的计算操作**

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

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



常见的并行编程模式

- 显式管理线程并行性

- 每次执行计算任务，都需要**新创建一个线程** (或按所需创建更多的并发线程)

```
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]);
}
```

（创建新的线程控制逻辑）

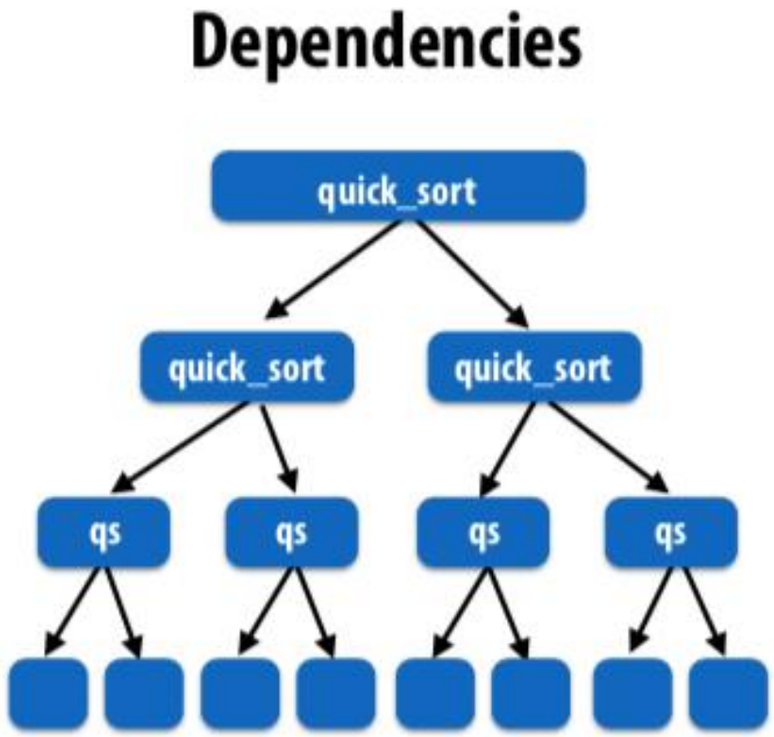
（隐式同步）

考虑分而治之的算法（divide-and-conquer）

以快排为例：

```
// 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);

  }
}
```

彼此独立的计算任务！



Fork-join的编程模型

- 在分治算法中表达独立工作的自然方式
- 本课程的代码示例将使用 Cilk++
  - C++ 语言扩展
  - 最初在麻省理工学院开发，现在改编为开放标准（在 GCC、Intel ICC 中）

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

“fork” **（创建新的线程控制逻辑）**

语义：调用 foo，但与标准函数调用不同，调用者可以异步执行多个 foo 。

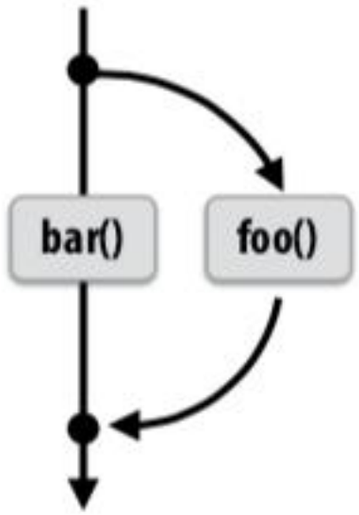
```
cilk_sync;
```

“join” **（隐式同步）**

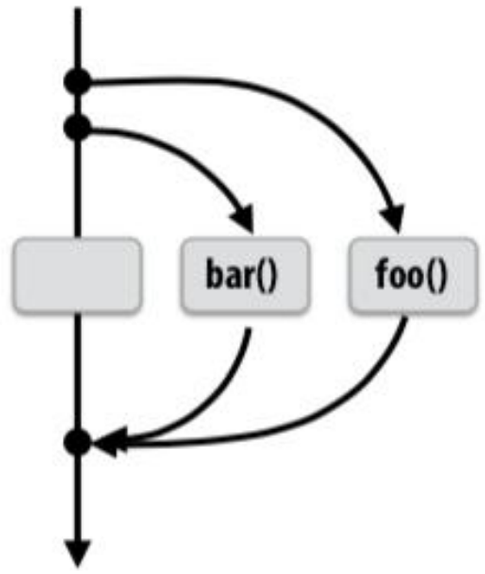
- 语义：当前函数产生的所有调用都完成时返回结果
- 注意：每个包含 **cilk_spawn** 的函数末尾都有一个隐式的 **cilk_sync**（含义：当 Cilk 函数返回时，与该函数相关的所有工作都已完成）

基本的Cilk++ 例子

```
// foo() and bar() may run in parallel
cilk_spawn foo();   开启新线程
bar();
cilk_sync;
```

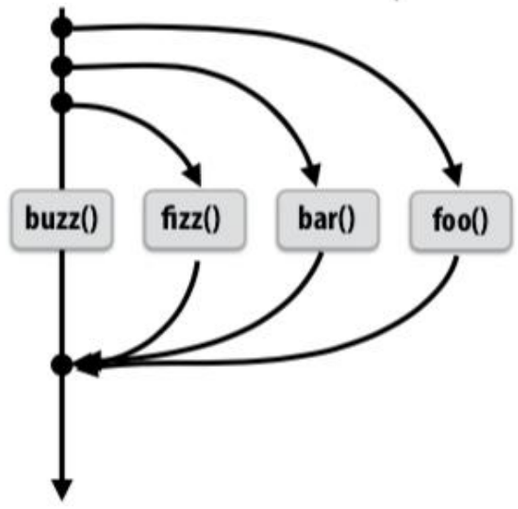



```
// foo() and bar() may run in parallel
cilk_spawn foo();   开启新线程
cilk_spawn bar();   开启新线程
cilk_sync;
```



与第一个示例的计算量相同，但运行时开销可能更高（由于两次spawns）

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



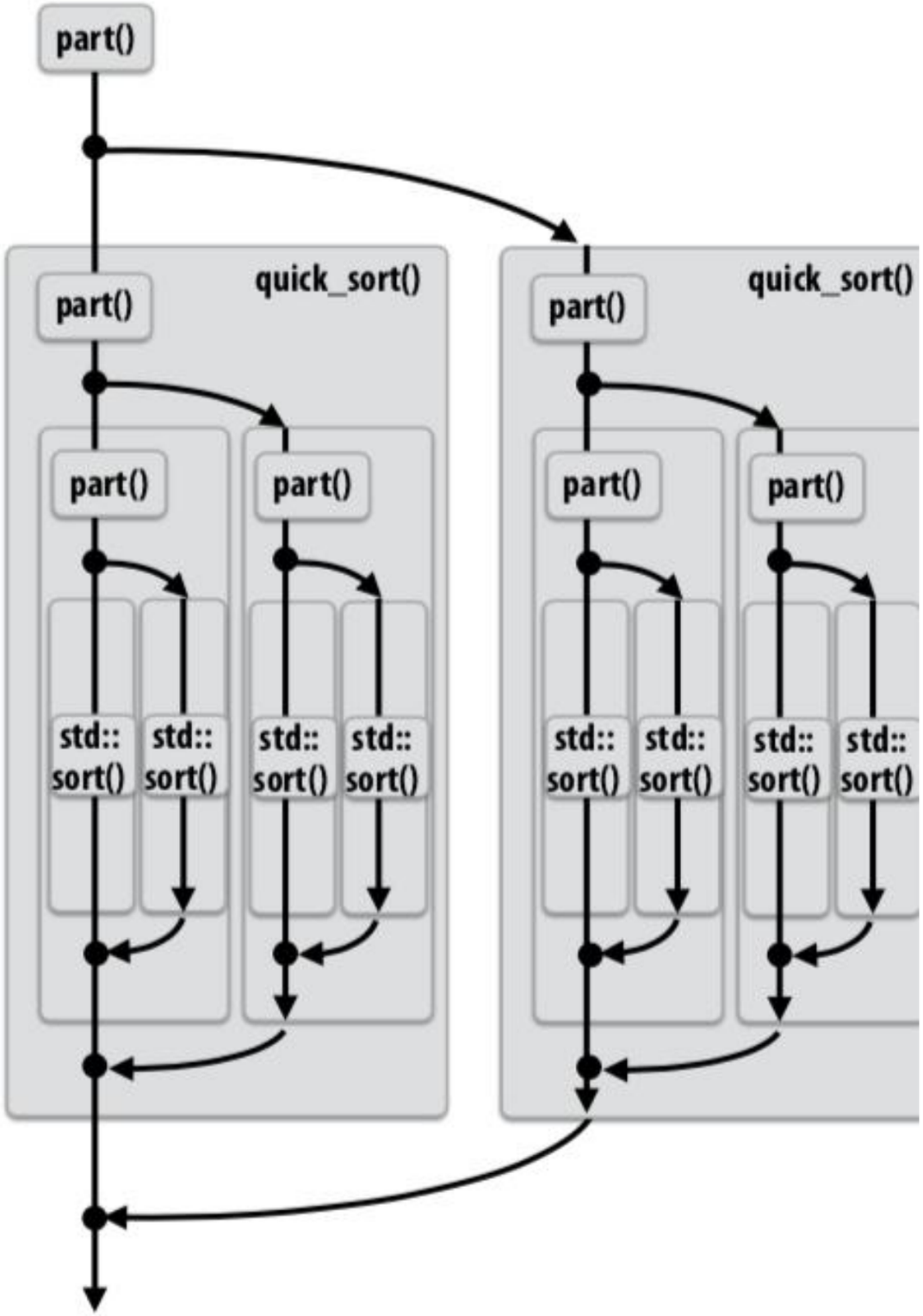
抽象 vs. 实现

- 注意， cilk_spawn 的抽象并未指定生成调用（spawned calls）的执行方式或时间
  - spawned call可以与调用者同时运行（以及调用者产生的所有其他调用）
- 但是 cilk_sync 确实对调度过程起到了**约束**作用
  - 所有生成的调用都必须在cilk_sync 返回之前完成

Cilk++ 中的并行快速排序

```
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);
  }
}
```

如果问题大小足够小，则按顺序排序
(产生的开销超过了潜在并行化的好处)



编写 fork-join 程序

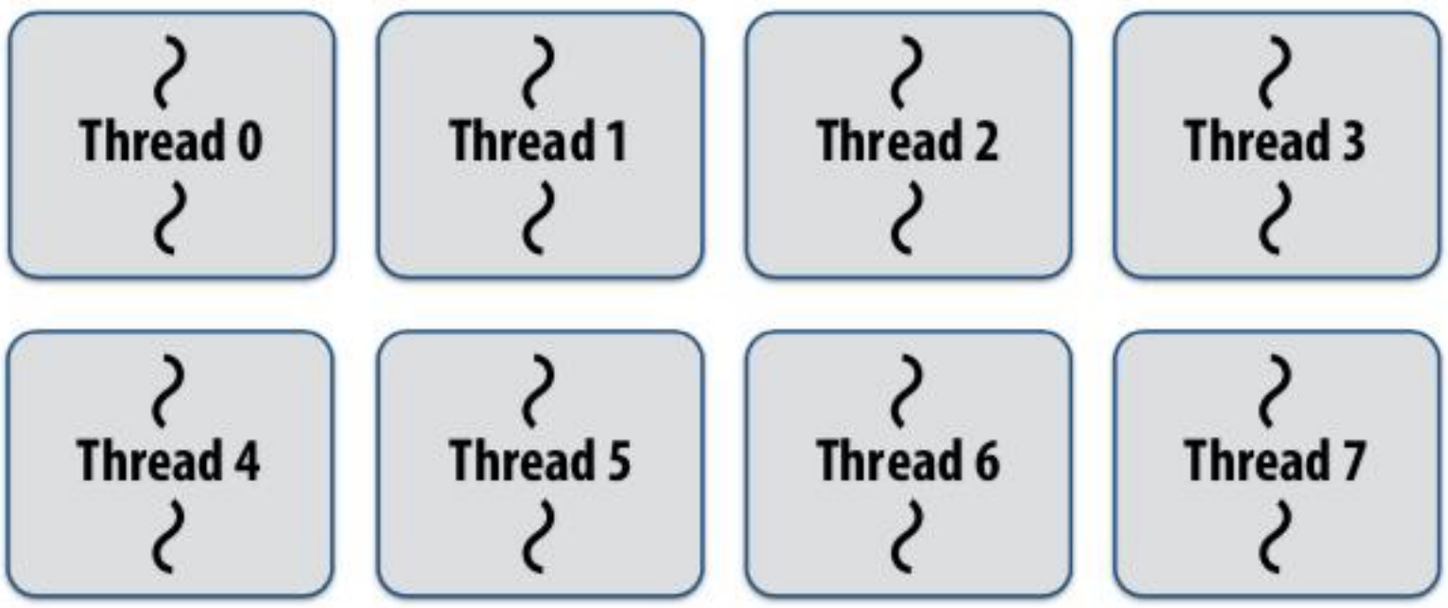
- 主要思想：使用cilk_spawn将彼此独立的“小程序”（潜在的并行性）暴露给系统
- 回顾并行编程的经验法则
  - **至少**产生与并行计算核心**一样多**的“小作业”（例如，程序应该至少产生与内核一样多的工作）
  - 需要比并行计算核心数量**更多（成倍数）**的“小工作”，以便在计算核心上实现负载平衡
    - “parallel slack” = 独立工作与机器并行执行能力的比率（实际上：~8 是很好的比率）（**8倍**）
- **但是独立的“小作业”也不能太多，以至于工作粒度太小，会产生管理细粒度工作的开销更多**

调度 fork-join 程序

- 一起来看一个基于pthread的fork-join 并行程序都有什么?
  - 线程创建：使用 pthread_create 为每个 cilk_spawn 启动 pthread
  - 同步指令：将 cilk_sync 转换为适当的 pthread_join 调用
- 潜在的性能问题
  - **线程启动与管理开销**超级大!
  - 并发运行的线程比内核**多得多!**
    - 上下文切换开销
    - 每个线程要处理的数据更少，计算量巨大，缓存局部性更低

工作线程池

- Cilk++ 运行时维护工作线程池
  - 思考一下：在应用程序启动时创建的所有工作线程
  - 与机器中的执行上下文一样多的工作线程（并行线程不能过多，不能超过机器并行的上限）

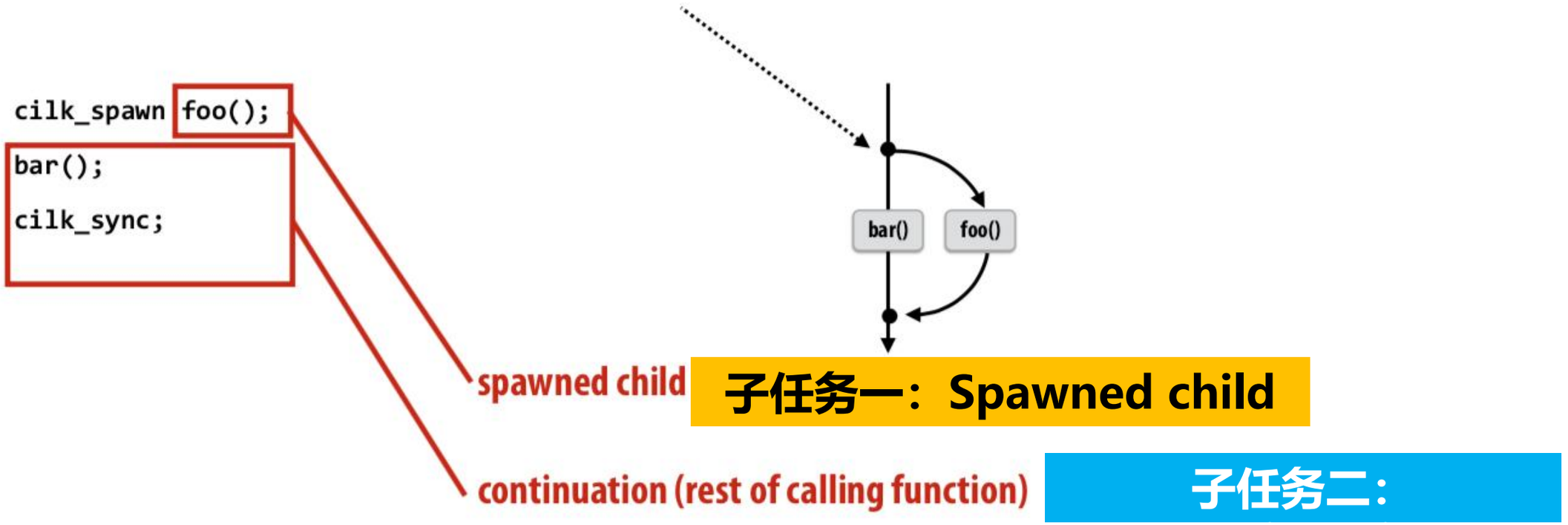


示例：我的4核笔记本电脑具有超线程（Hyper-Threading）的8线程工作池

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

思考一下：执行以下代码

- 具体来说，考虑从生成 foo() 的点开始执行



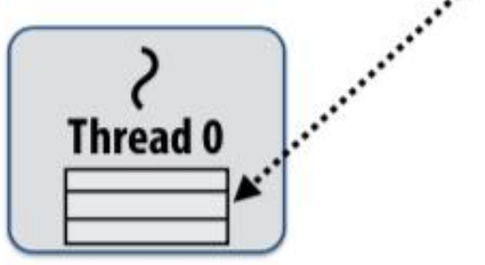
- foo() 和 bar() 应该由哪些线程执行?

首先，考虑串行实现

- 先运行子进程 **子任务一：Spawned child** 通过常规函数调用
  - 整个程序的执行过程：先运行 foo()。然后，等待从 foo() 返回结果。最后，再运行 bar()
  - foo到bar的函数调用切换，隐含在线程的堆栈

传统线程调用栈

(bar 将在foo的 return 之后执行)



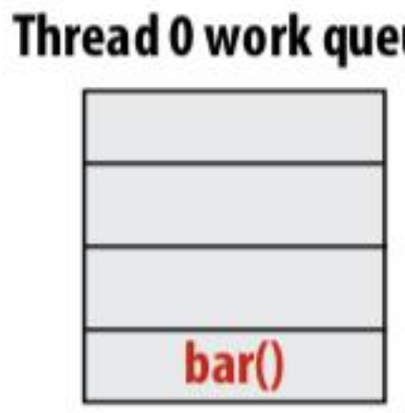
线程0：执行foo()函数....

线程1：在执行 foo（）过程中，线程1 空闲了会怎么样?

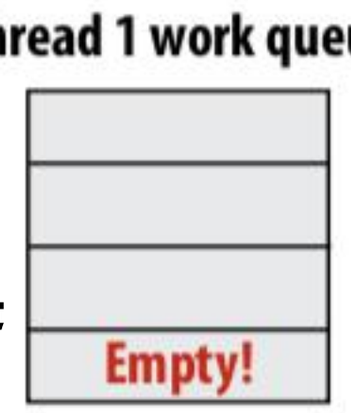
低效：此时线程1本可以去执行bar（）函数!

每个线程的工作队列会存储“要做的工作”

- 到达 cilk_spawn foo() 后，线程将继续放在其工作队列中，并开始执行 foo()



bar()在Thread的队列中，等待被执行.....



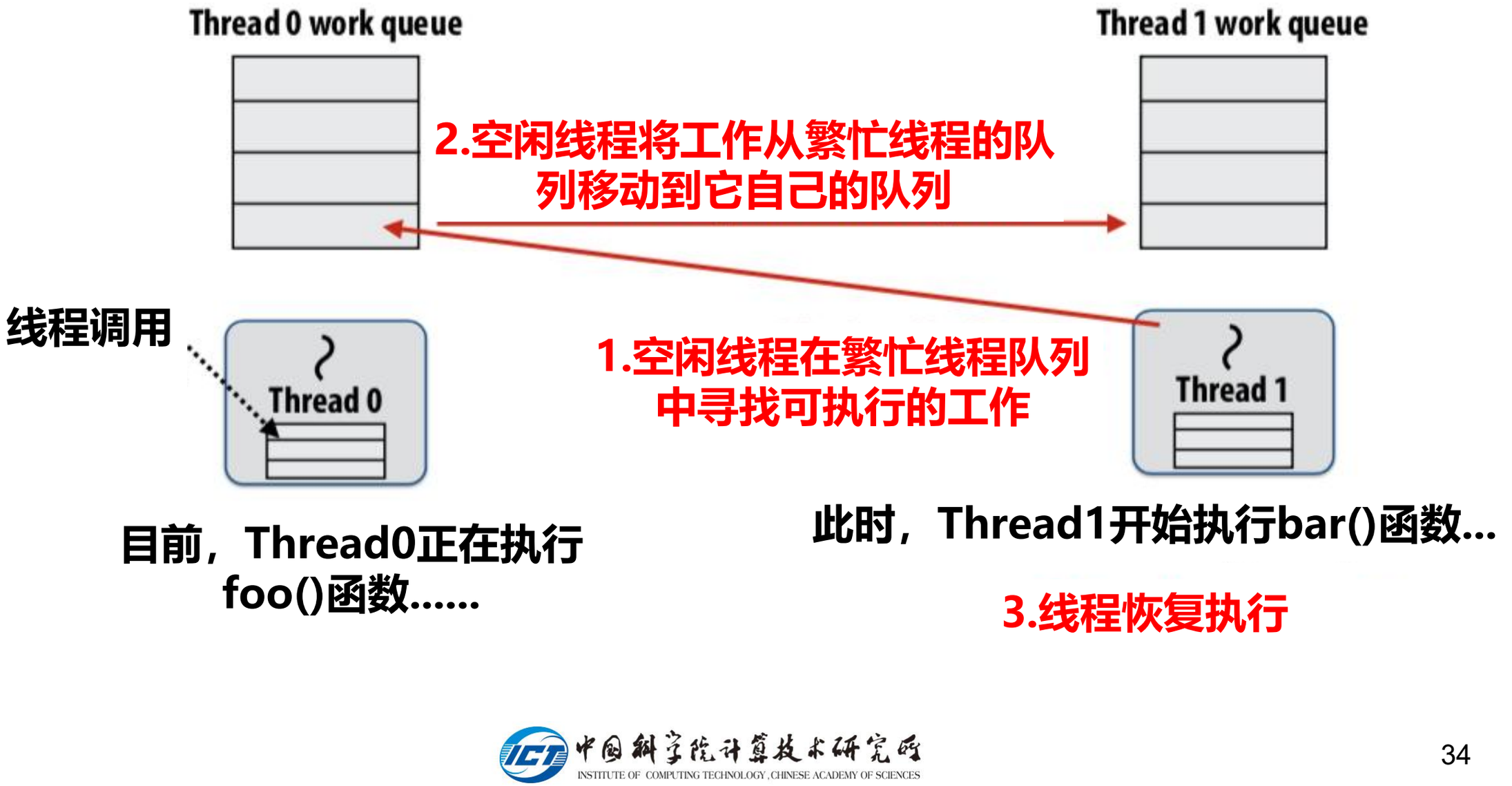
Thread1的队列是空闲状态，非常闲

线程调用

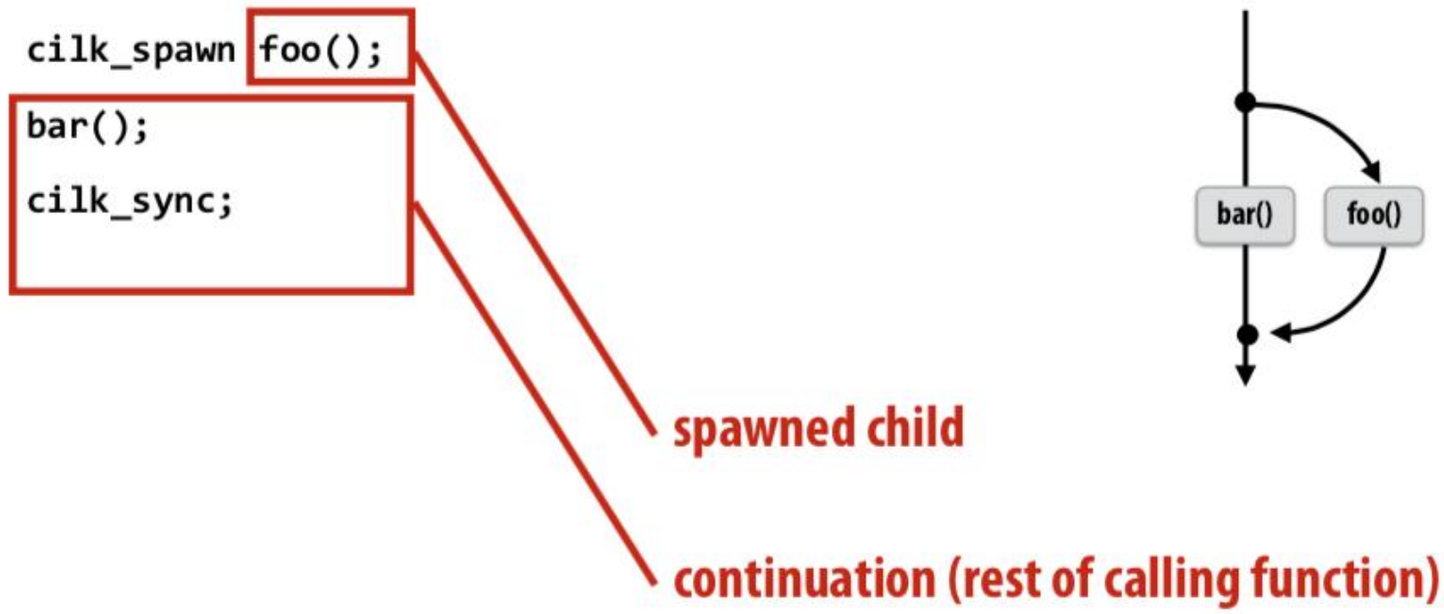
目前，Thread0正在执行foo()函数.....

空闲线程从繁忙线程“窃取”工作

- 如果thread 1 空闲（也就是它自己的队列中没有工作），那么它会在thread 0 的队列中查找要执行的工作



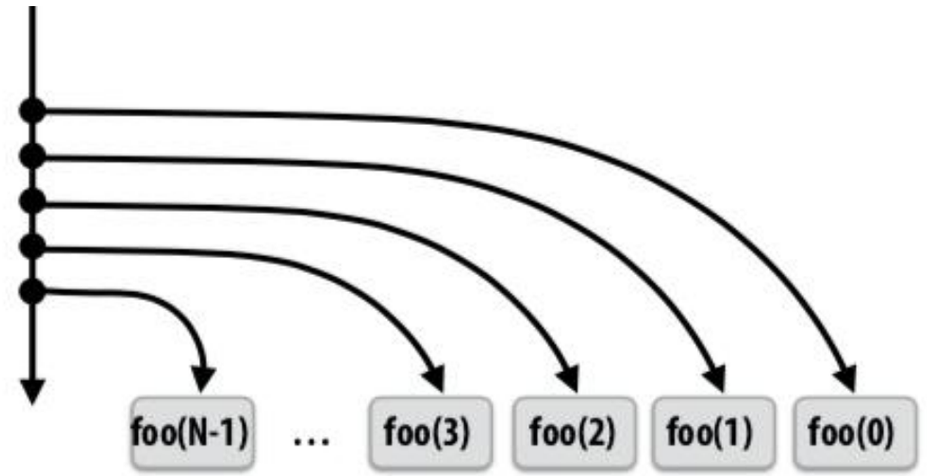
那么在 **子任务一：Spawned child** 要被执行的阶段，哪个子任务应该被进程调度执行?



- 情况一：先执行 **子任务二：Continuation** 记录child进入队列，以供以后执行
  - Child 可以被其他线程窃取执行（“child stealing”）
- 情况二：先执行 **子任务一：Spawned child** 记录continuation进入队列，以供以后执行
  - Continuation可以被其他线程窃取执行（“continuation stealing”）

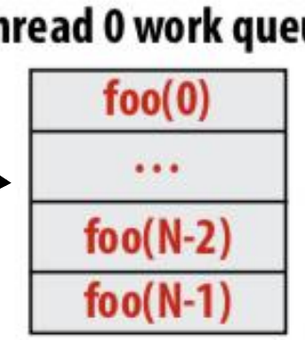
思考一下：用线程执行以下代码（派生多个可并行子任务）

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

cilk_sync;
```



- 情况一：先执行 **子任务二：Continuation**（“child stealing”）

- 调用者线程(Caller thread)在执行任何迭代之前，为所有迭代生成需要计算的任务work（产生多个子任务）

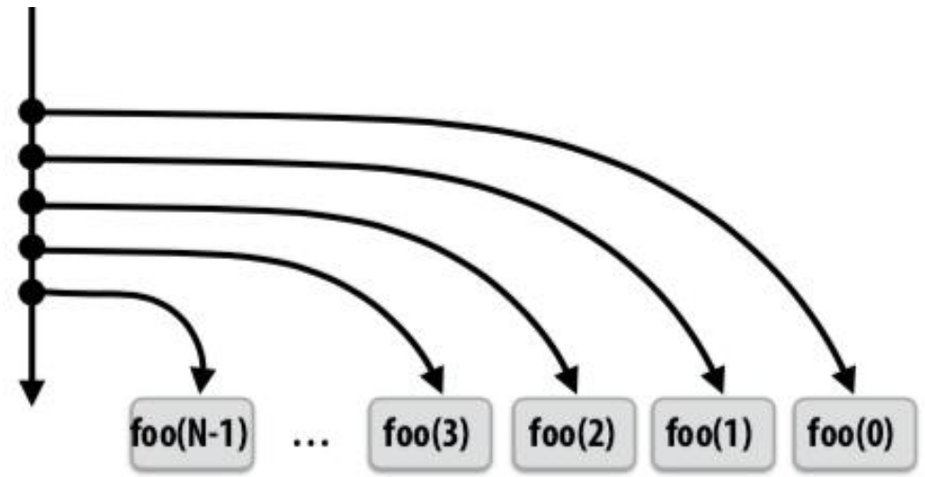


- 思考：调用图的广度优先遍历。生成spawned各个子任务的复杂度为 O(N)

- 如果没有其他进程窃取这些child子任务，cilk_spawn 的各个子任务的执行顺序，将依赖于各个子任务要处理数据的编号（i值）（各个子任务的执行顺序，与删除 cilk_spawn 的不相同）

思考一下：用线程执行以下代码（派生多个可并行子任务）

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

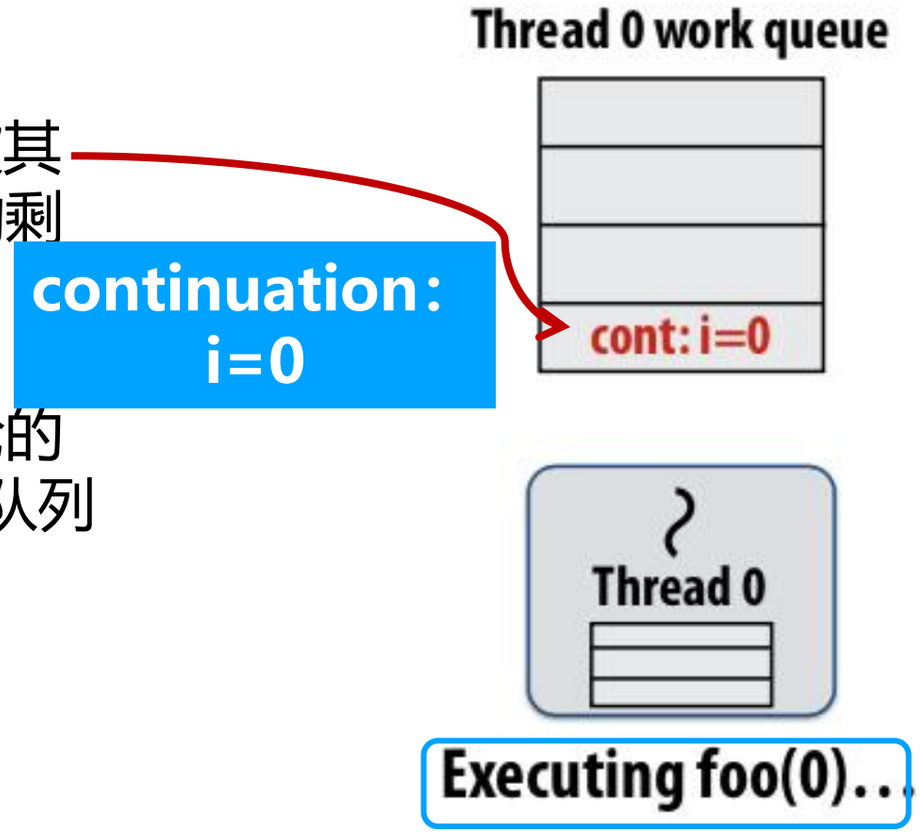


- 情况二：先执行 **子任务一：Spawned child**（“continuation stealing”）

- 调用者线程**只创建一个子任务**（如：cont:i=0），用来被其他线程窃取(continuation代表所有i=0这一轮迭代任务的剩余任务)

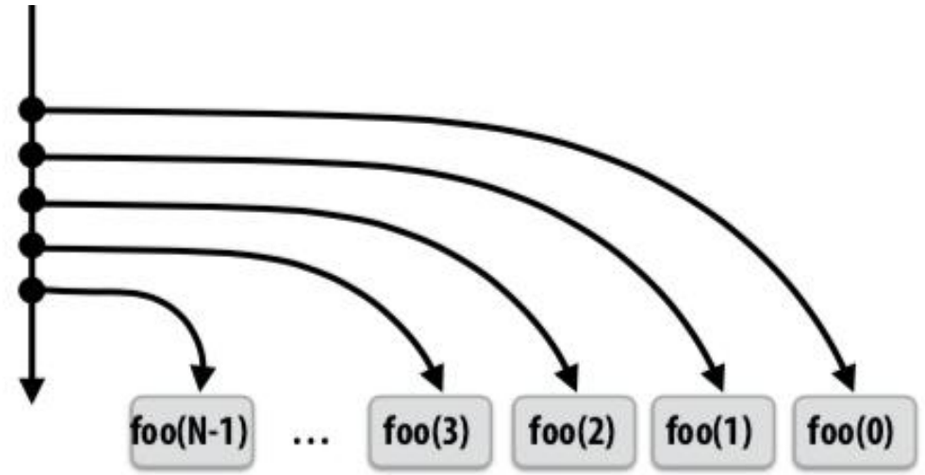
- **如果没有发生窃取**，线程不断地从工作队列中弹出下一轮的迭代任务，将新的迭代任务（具有更新的值 i）排入等待队列

- 各个子任务的执行顺序，与删除 cilk_spawn 的程序相同



思考一下：用线程执行以下代码（派生多个可并行子任务）

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

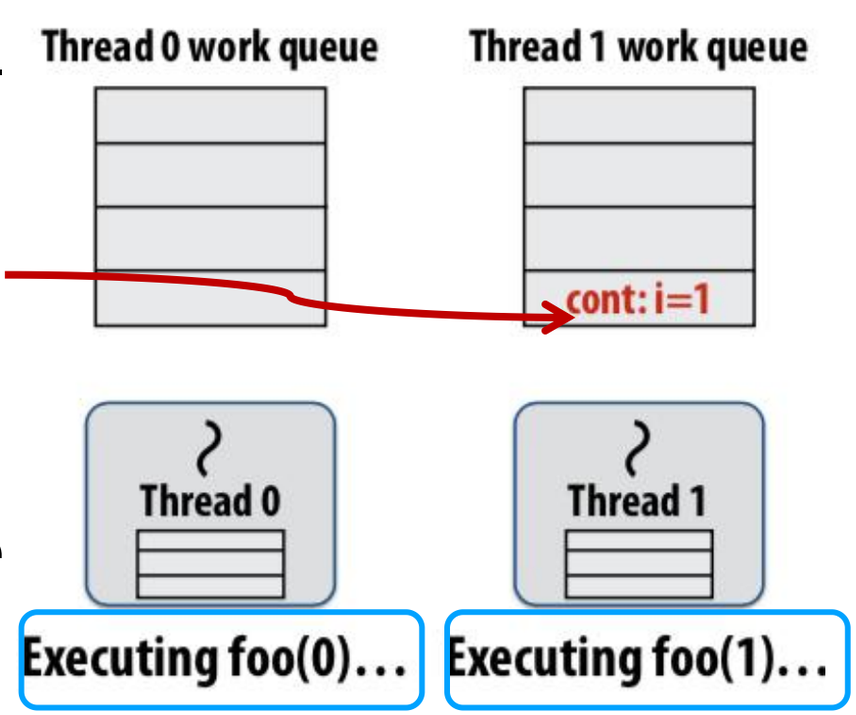


- 情况二：先执行 **子任务一：Spawned child** continuation stealing”）

- **如果“continuation”被窃取**，窃取线程产生并执行下一次迭代

- 在窃取线程中，让continuation（简称：cont）的i 自增1并入队等待执行

- 可以证明：具有T个线程的系统，其工作队列Work queue的存储总量，不会超过单线程执行栈存储总量的T倍

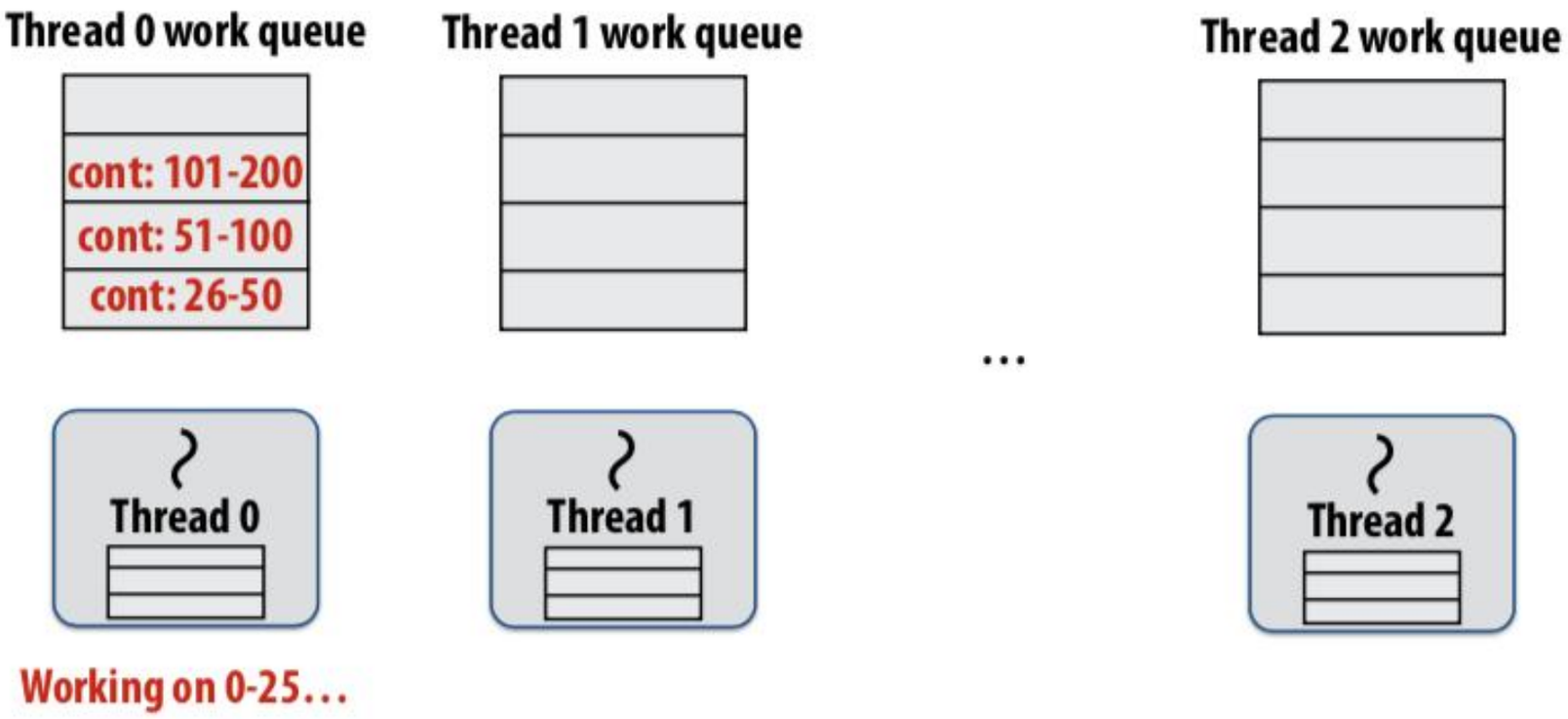


对快排程序进行调度

- 假设有 200 个元素

```
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);
  }
}
```

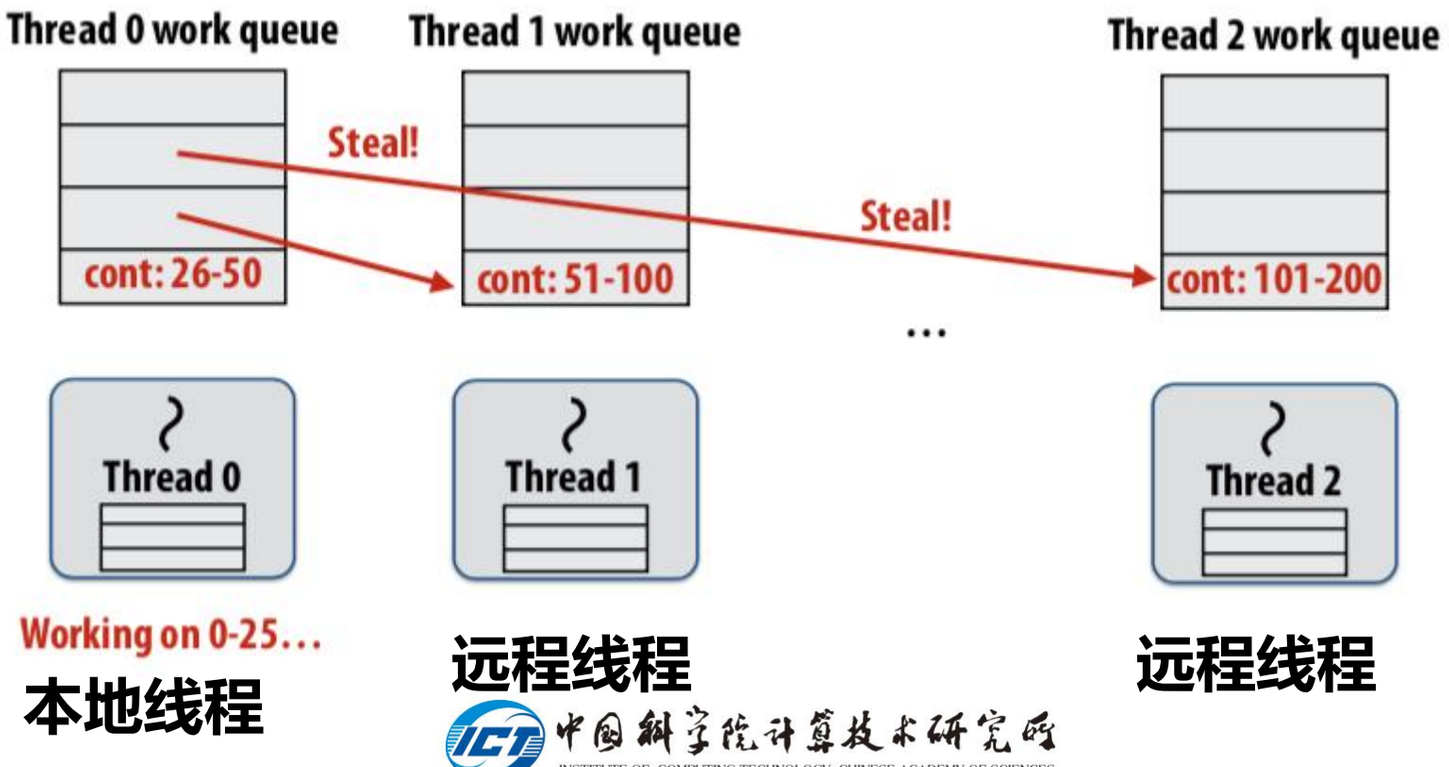
队列中的哪些子任务应该被其他线程窃取?



子任务窃取（work stealing）的实现

步骤一：子任务的出队

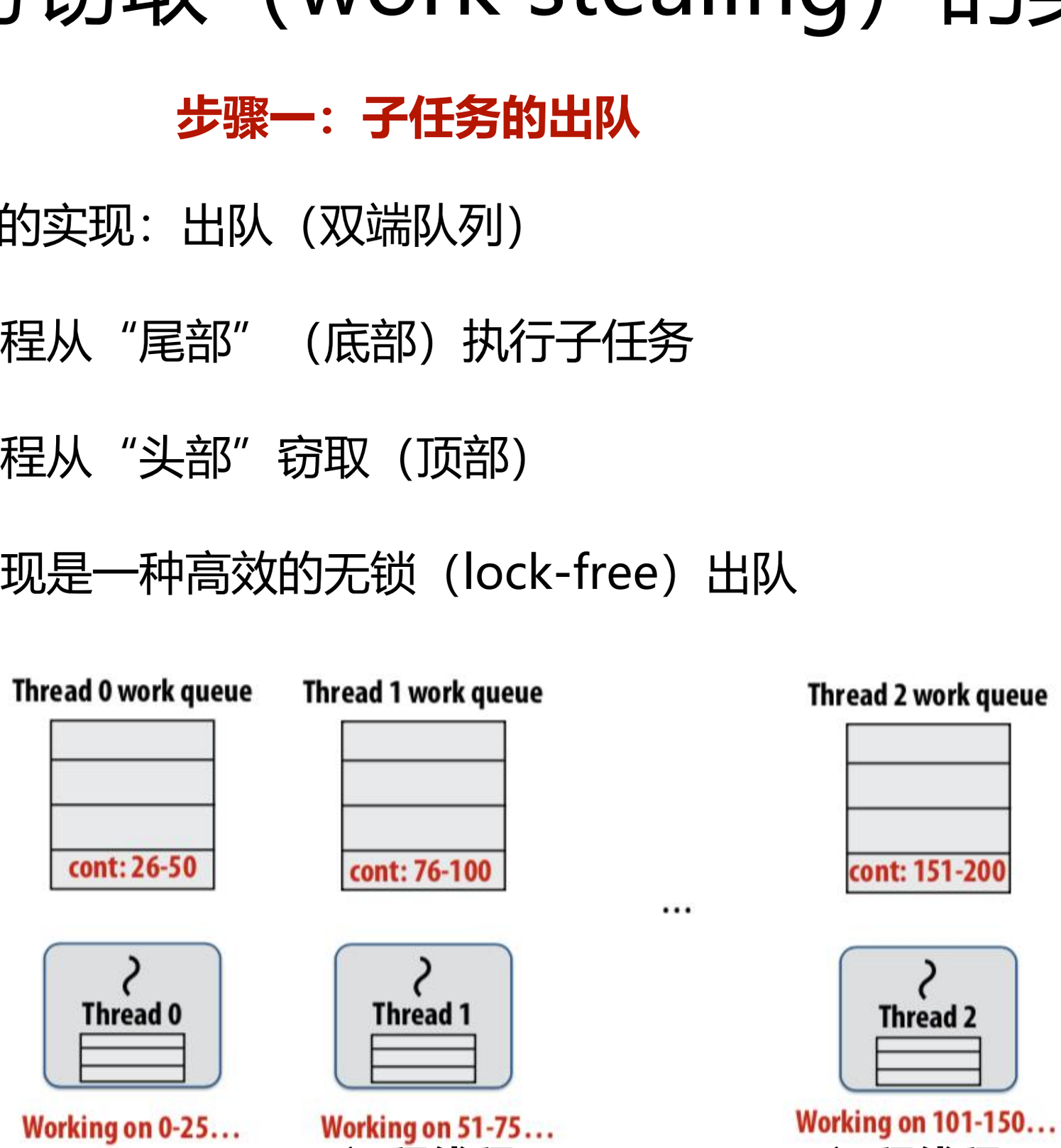
- 工作队列的实现：出队（双端队列）
  - 本地线程从“尾部”（底部）执行子任务
  - 远程线程从“头部”窃取（顶部）
- 这种实现是一种高效的无锁（lock-free）出队



子任务窃取（work stealing）的实现

步骤一：子任务的出队

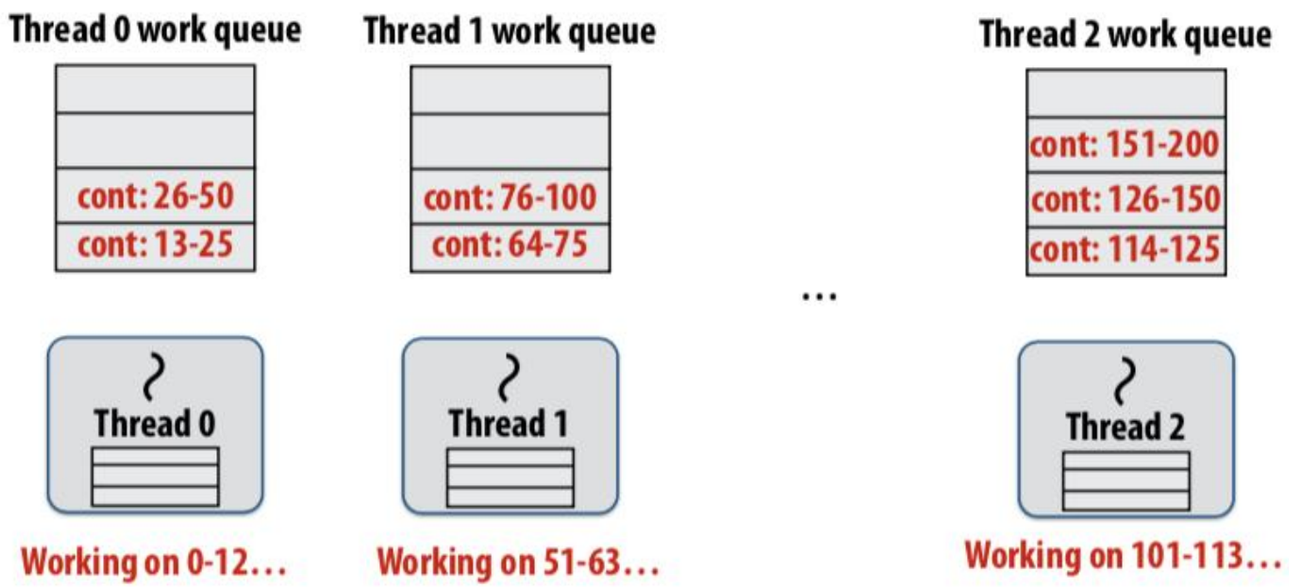
- 工作队列的实现：出队（双端队列）
  - 本地线程从“尾部”（底部）执行子任务
  - 远程线程从“头部”窃取（顶部）
- 这种实现是一种高效的无锁（lock-free）出队



子任务窃取（work stealing）的实现

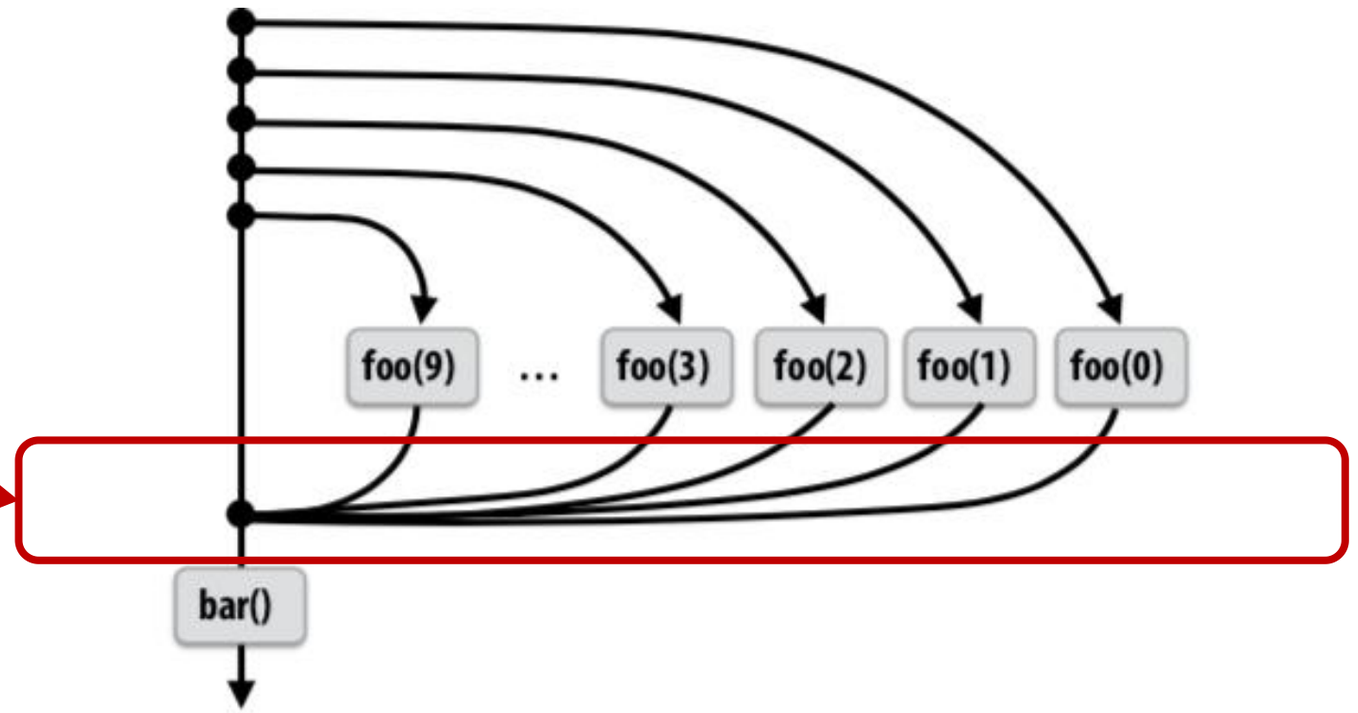
步骤二：随机选择“受害者”

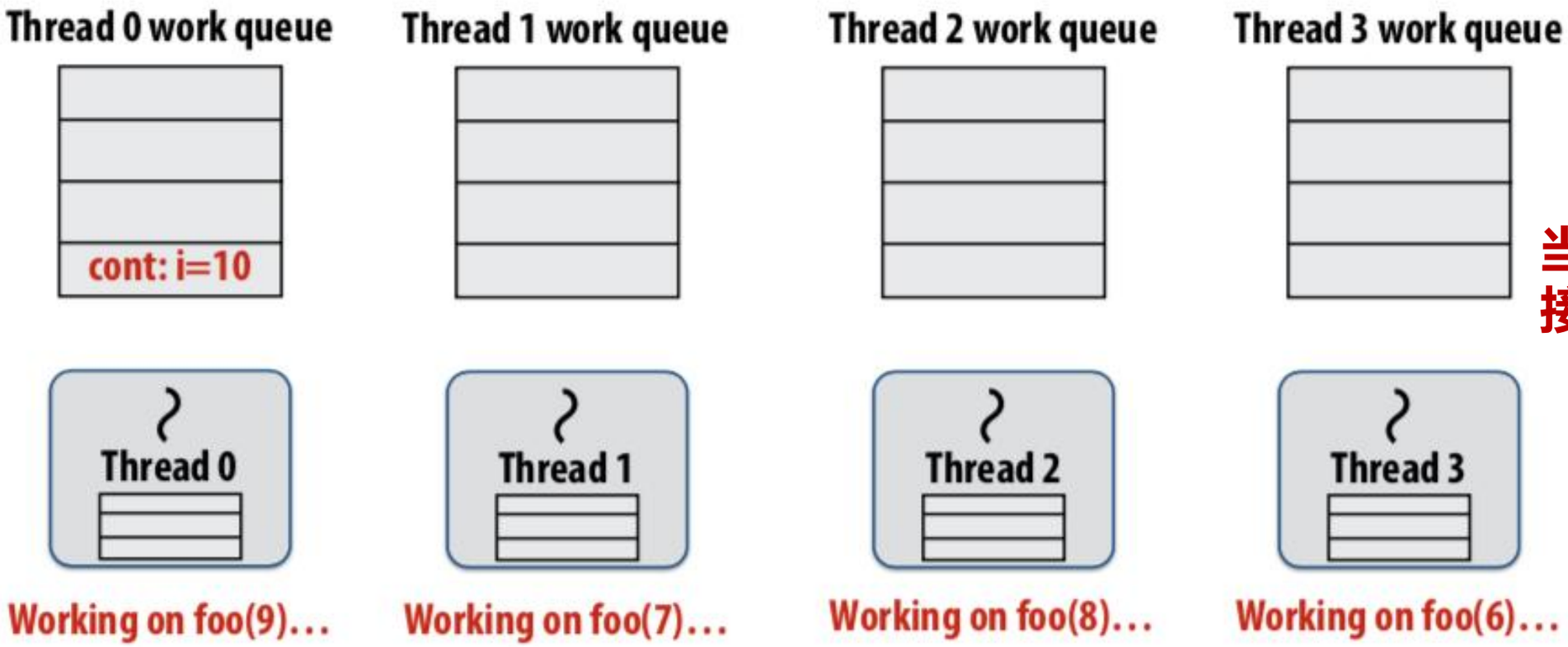
- 空闲线程随机选择一个线程来尝试窃取
- 从等待队列的顶部窃取
  - 减少与本地线程的争用：本地线程没有访问与窃取线程相同的待执行子任务
  - 在整个程序调用的最开始就窃取工作：从整个程序的全部计算任务的视角，“work stealing”的开销很小，因此执行窃取的成本会在未来更长的计算中分摊
- 最大化局部性：结合 run-child-first 策略，本地线程在调用树（call tree）所在的线程，开始执行最初的child子任务



同步（sync）的实现

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

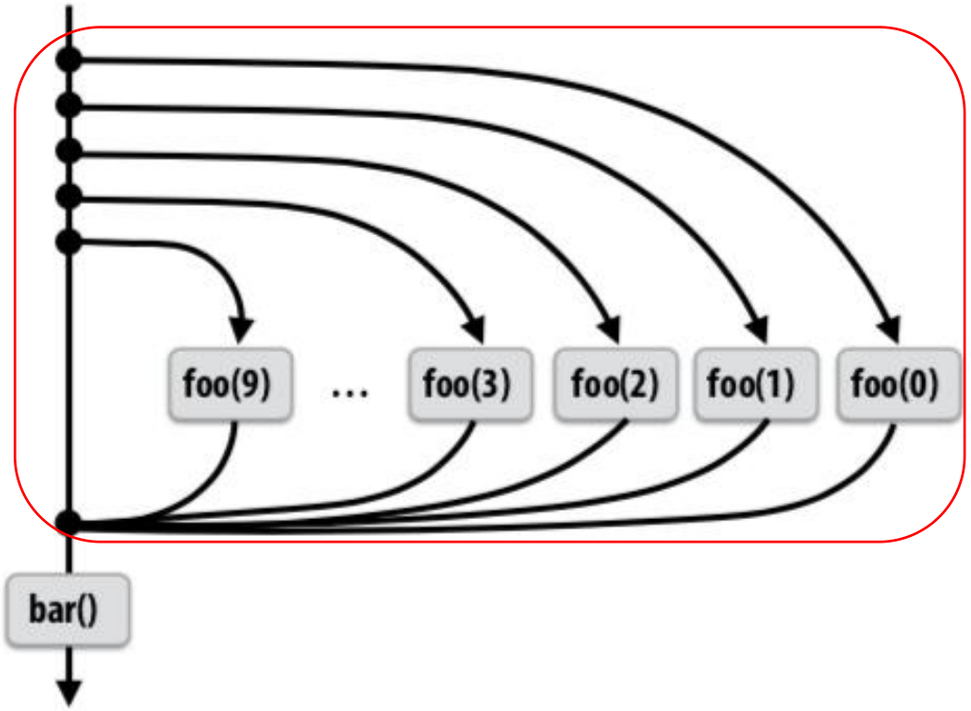


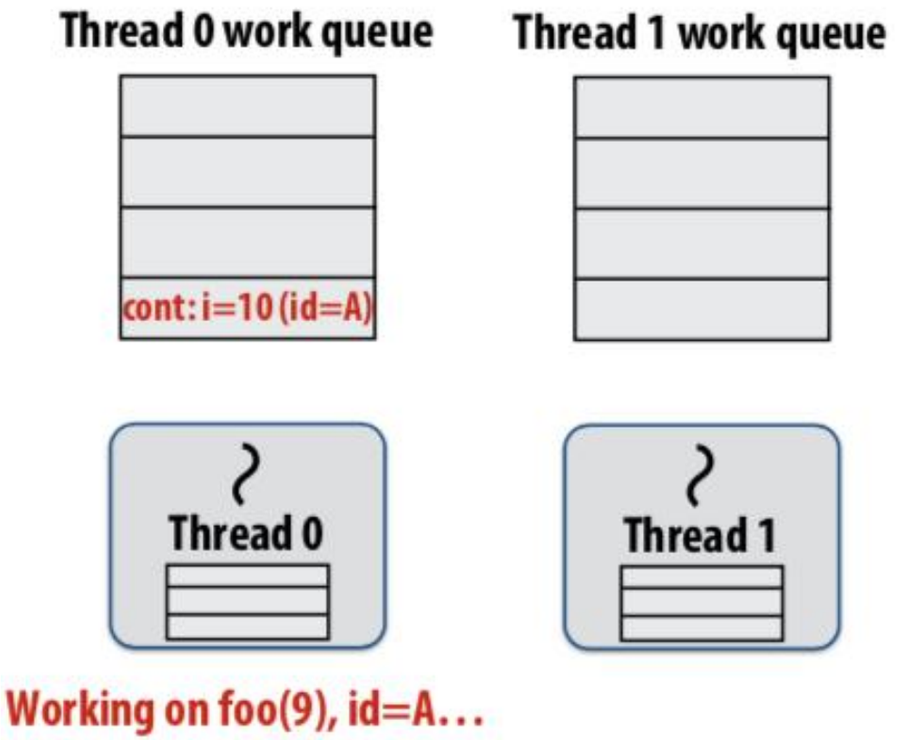


当循环中的所有工作接近完成时，各个工作线程的状态

同步的实现:无子任务窃取情况

```
block (id: A)
for (int i=0; i<10; i++) {
  cilk_spawn foo(i);
}
cilk_sync; Sync for all calls spawned within block A
bar();
```



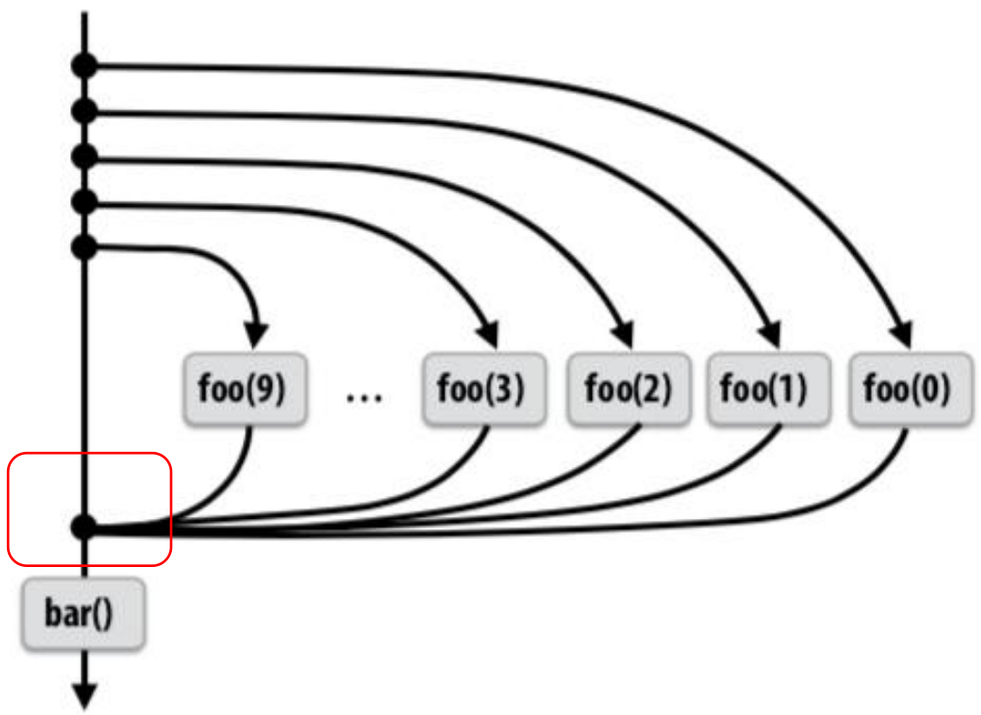


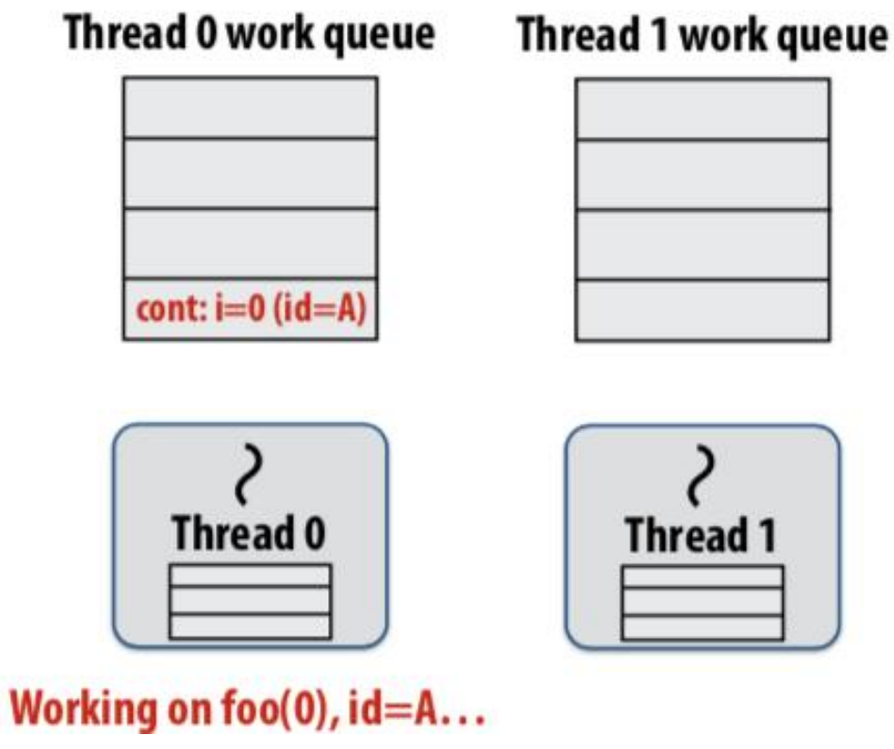
- 如果其他线程没有窃取任何工作，那么在同步点就没有需要同步执行的操作了。
- 没有线程窃取意味着，只有一个线程在计算block A

cilk_sync is a no-op.

同步的实现: 等待并入（stalling join）策略

```
block (id: A)
  for (int i=0; i<10; i++) {
    cilk_spawn foo(i);
  }
  cilk_sync;  Sync for all calls spawned within block A
  bar();
```





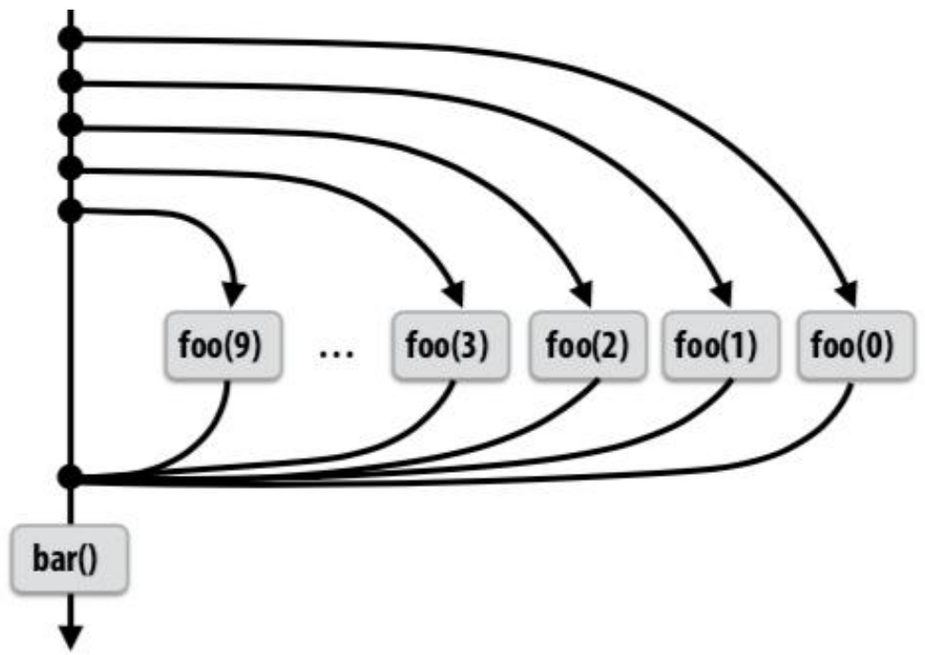
示例 1：“停顿”加入策略

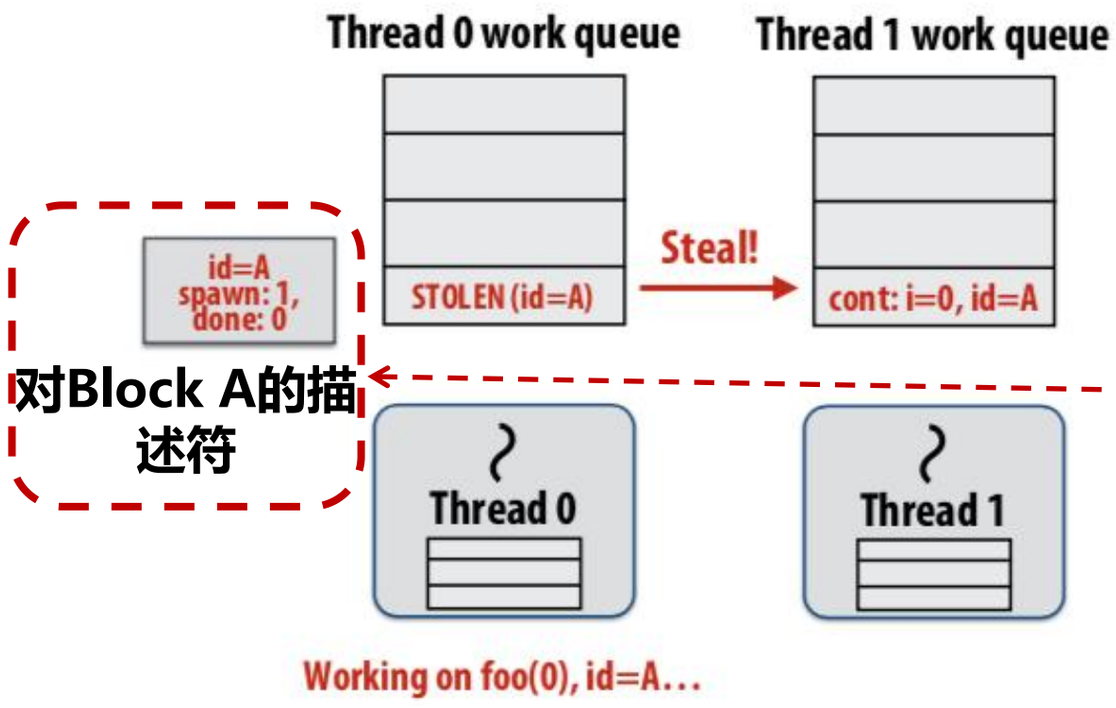
启动 fork 的线程，必须执行cilk_sync的同步操作。

- **因此，Thread0，它必须要要等待所有派生（spawned）的工作完成，即：执行到并入（join）阶段，因此该策略也称为等待并入（stalling join）策略。**
- **线程 0 是启动 fork 的线程**

同步的实现: 等待并入（stalling join）策略

```
block (id: A)
  for (int i=0; i<10; i++) {
    cilk_spawn foo(i);
  }
  cilk_sync;  Sync for all calls spawned within block A
  bar();
```



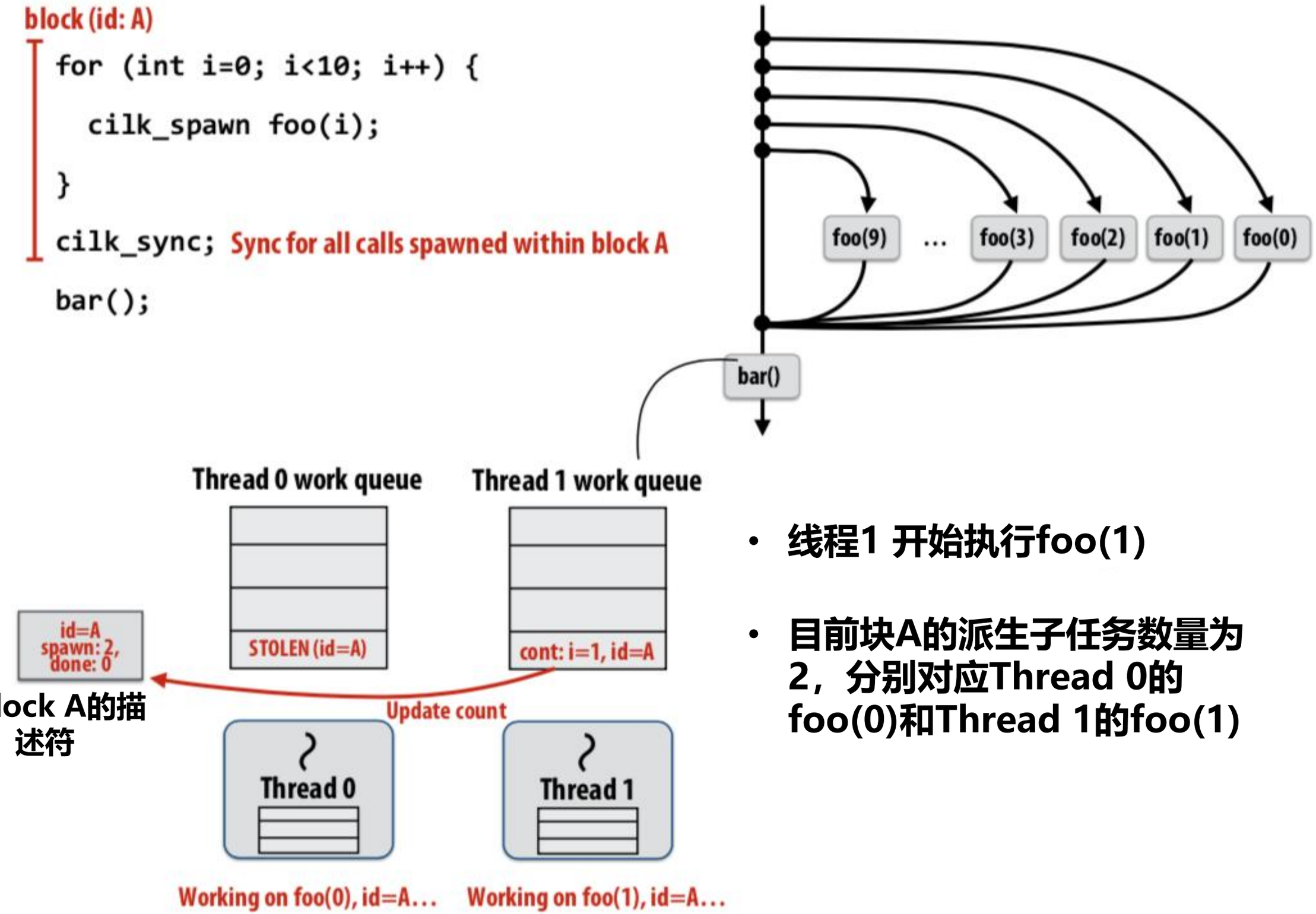


空闲线程 1 从繁忙线程 0 窃取
注意：创建块 A 的描述符

- **该描述符跟踪块的未完成生成数，以及已完成的生成数。**
- **在左图中，“1 spawn” 对应于 Thread 0 正在运行尚未完成的 foo(0)。**

同步的实现: 等待并入（stalling join）策略

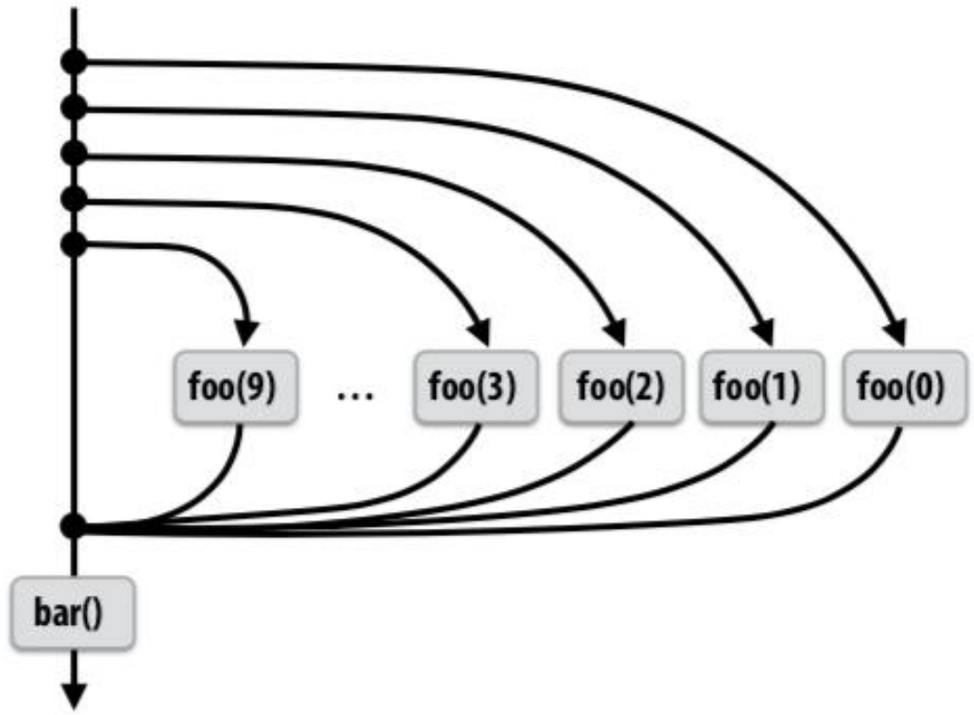
```
block (id: A)
for (int i=0; i<10; i++) {
  cilk_spawn foo(i);
}
cilk_sync; Sync for all calls spawned within block A
bar();
```

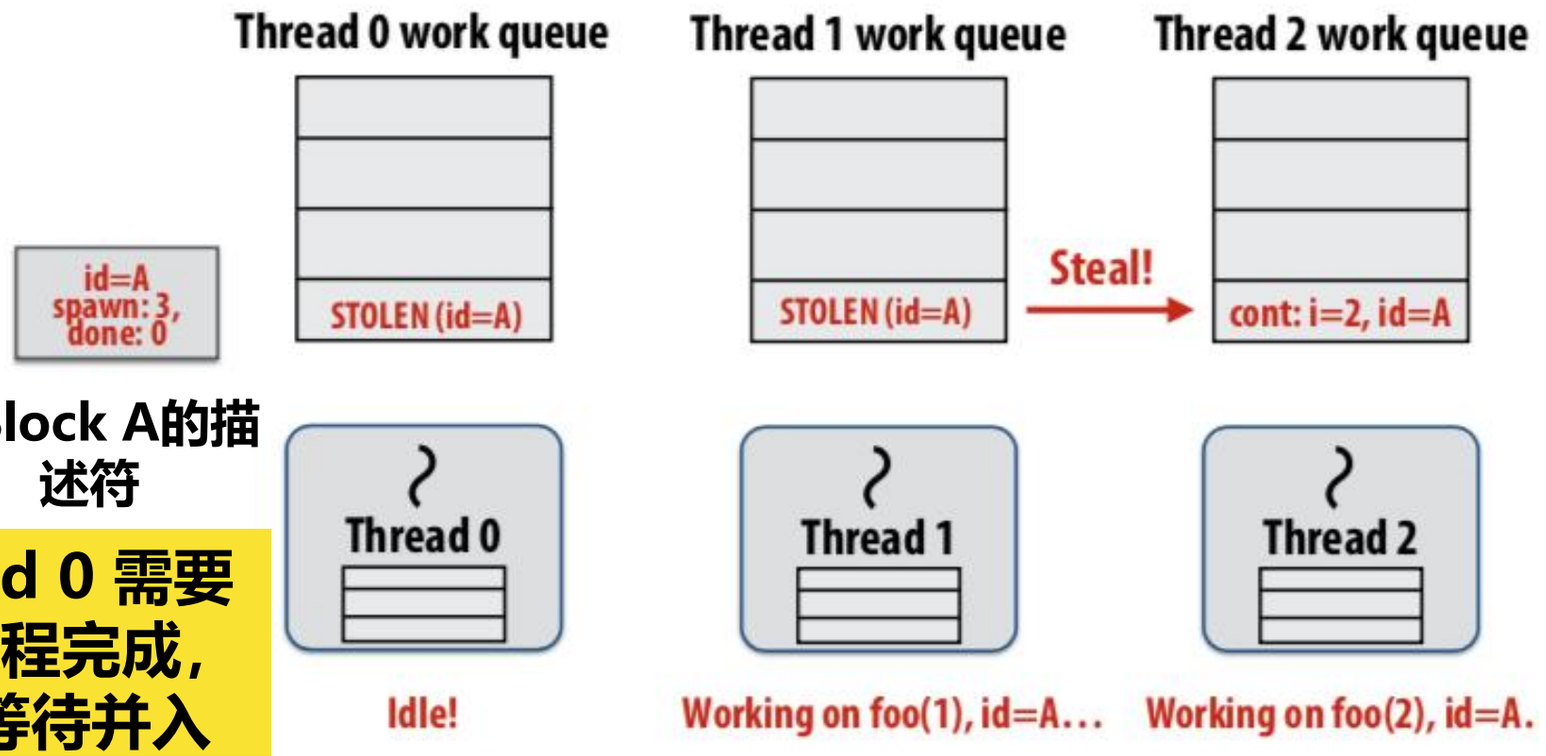


- 线程1 开始执行foo(1)
- 目前块A的派生子任务数量为2，分别对应Thread 0的foo(0)和Thread 1的foo(1)

同步的实现: 等待并入（stalling join）策略

```
block (id: A)
  for (int i=0; i<10; i++) {
    cilk_spawn foo(i);
  }
  cilk_sync;  Sync for all calls spawned within block A
  bar();
```





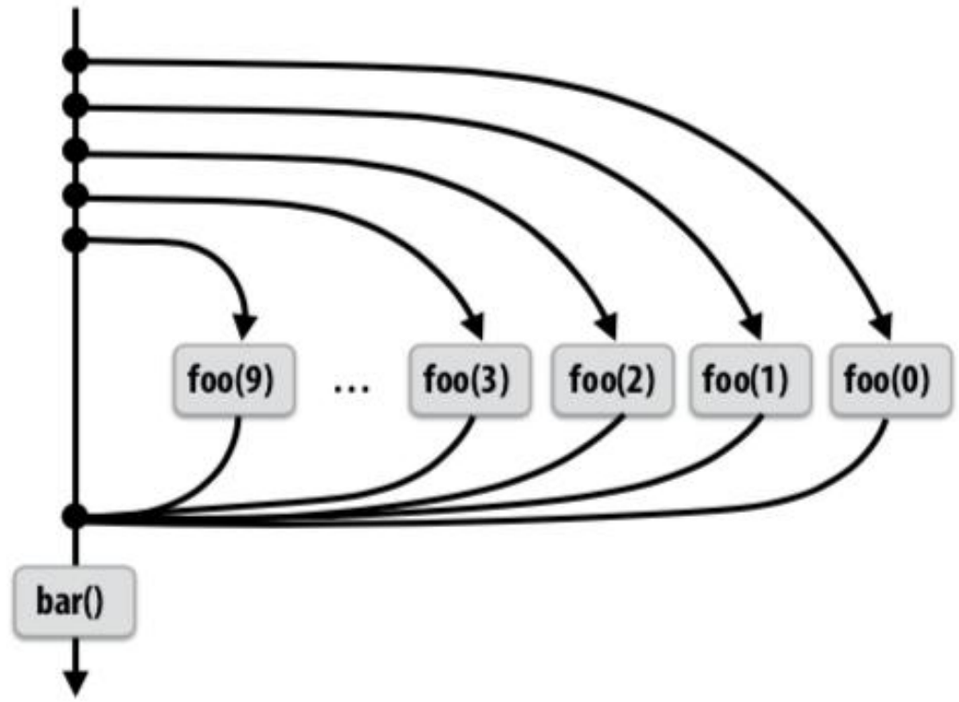
对Block A的描述符

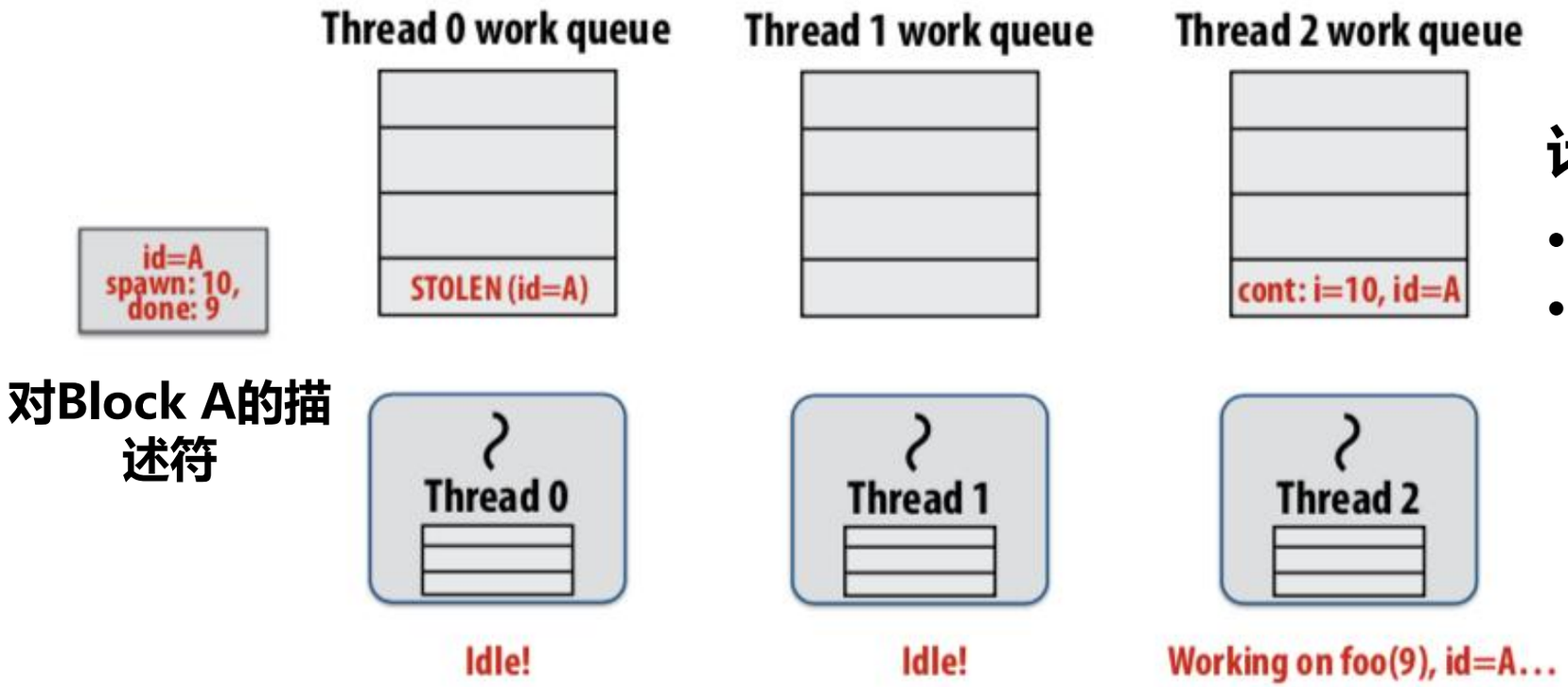
Thread 0 需要所有线程完成，即：等待并入

- **Thread 2偷取剩下任务foo(2)，更新派生数量为3**
- **Thread 0完成了foo(0)，Thread 0不能再执行其他迭代子任务了，一直Idle直到迭代都结束**
- **Thread 2开始执行foo(2)**

同步的实现: 等待并入（stalling join）策略

```
block (id: A)
  for (int i=0; i<10; i++) {
    cilk_spawn foo(i);
  }
  cilk_sync;  Sync for all calls spawned within block A
  bar();
```



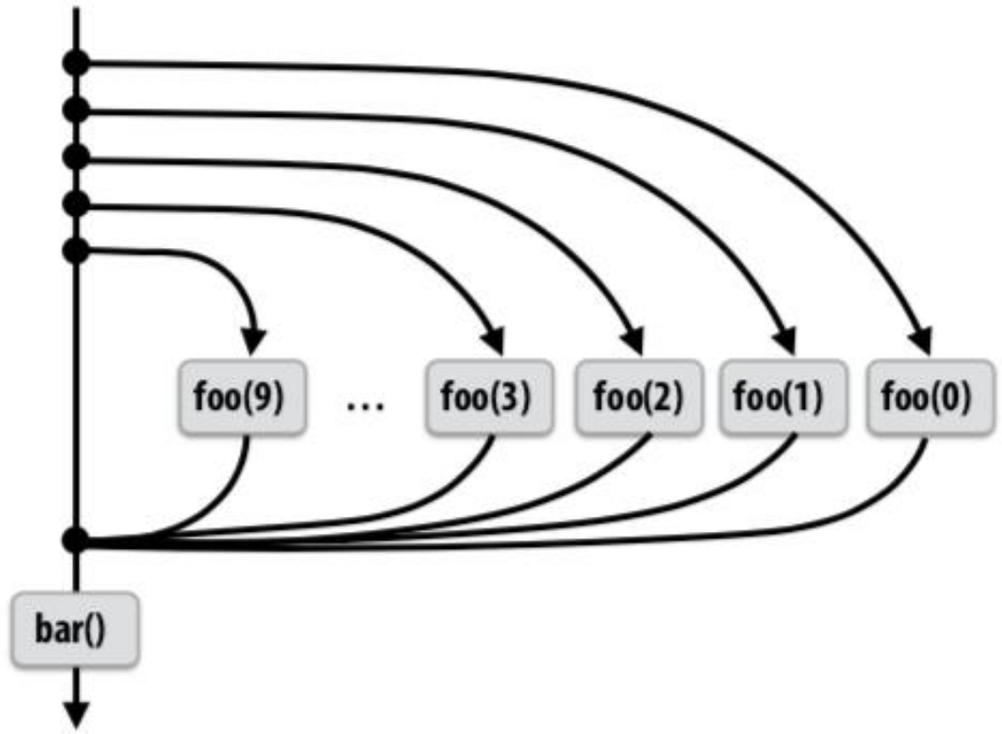


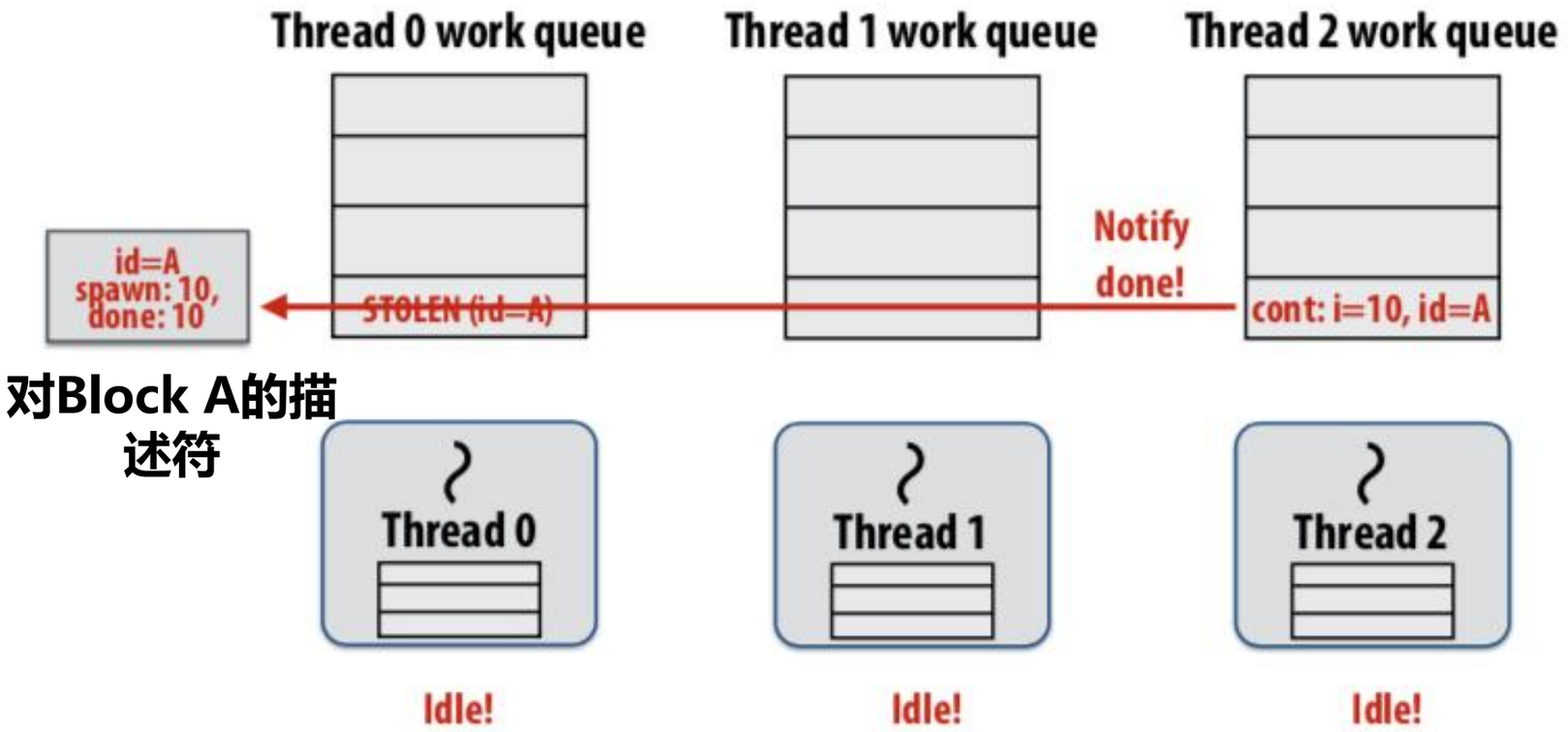
计算即将结束时：

- **只有foo(9)还没有完成**
- **对Block A的描述符：派生了10个子任务，完成了9个**

同步的实现: 等待并入（stalling join）策略

```
block (id: A)
  for (int i=0; i<10; i++) {
    cilk_spawn foo(i);
  }
  cilk_sync;  Sync for all calls spawned within block A
  bar();
```



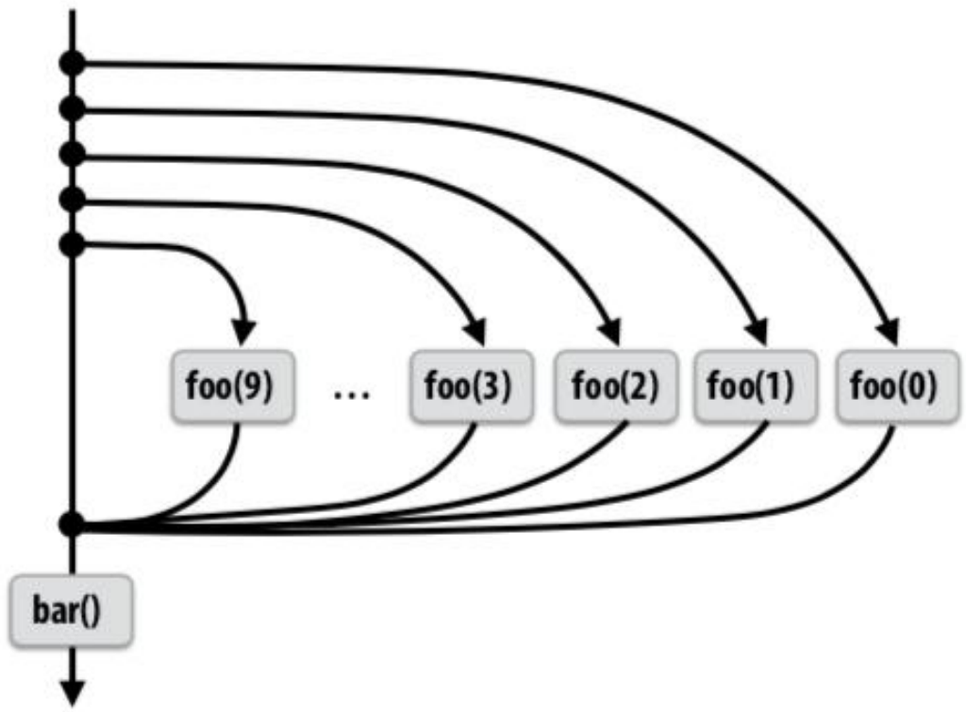


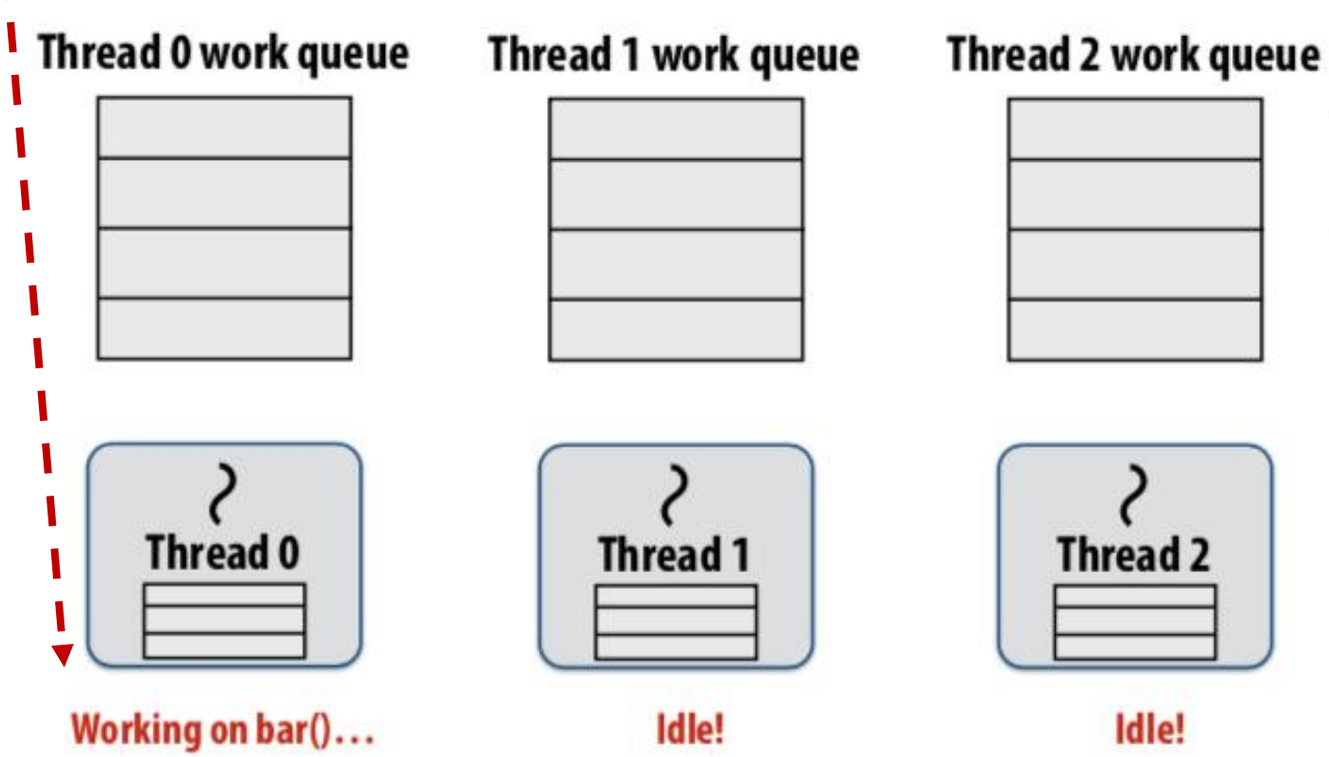
对Block A的描述符

- **最后一个派生子任务完成了，更新对Block A的描述符**

同步的实现: 等待并入（stalling join）策略

```
block (id: A)
  for (int i=0; i<10; i++) {
    cilk_spawn foo(i);
  }
  cilk_sync;  Sync for all calls spawned within block A
  bar();
```

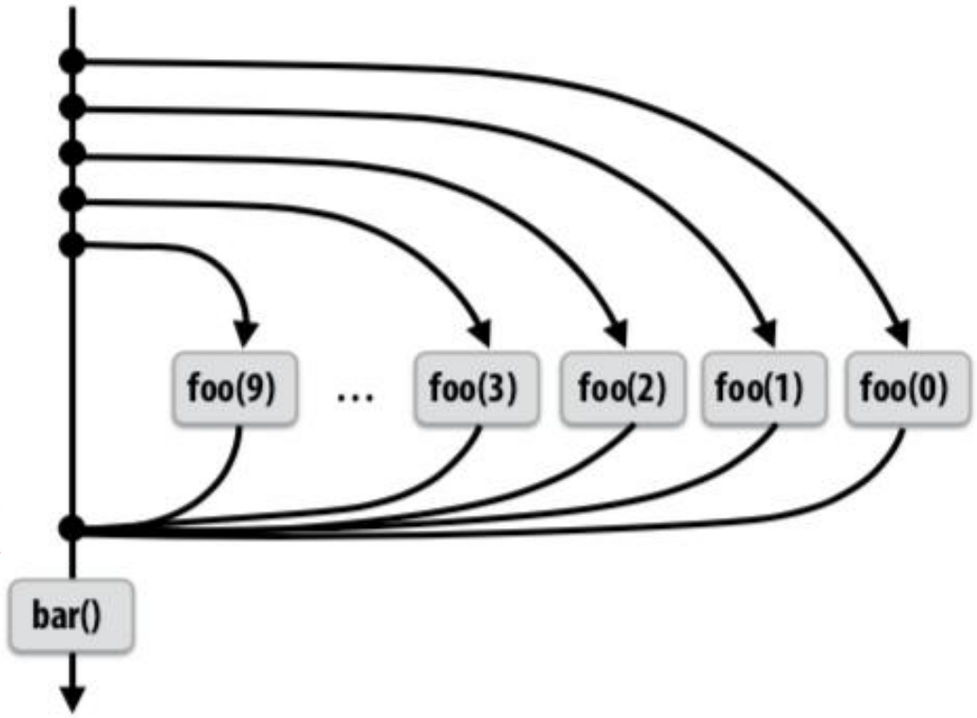




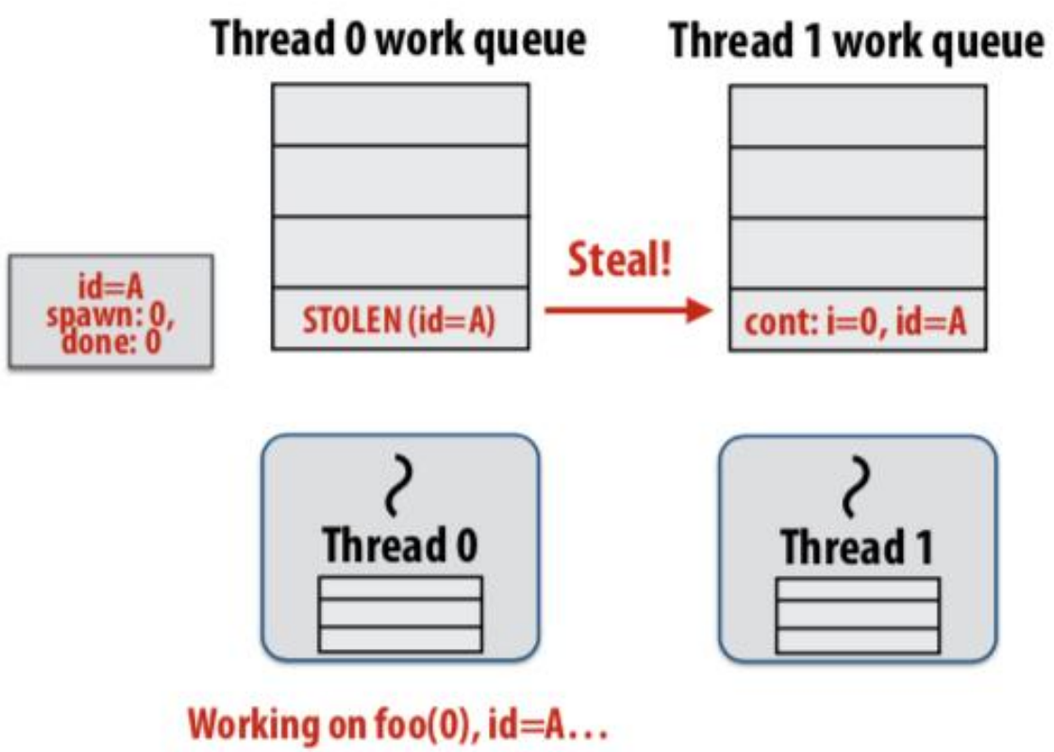
- **Thread 0目前开始执行剩余任务，开始执行bar()**
- **注意：对Block A的描述符目前已经被释放了。**

同步的实现: 贪心策略

```
block (id: A)
  for (int i=0; i<10; i++) {
    cilk_spawn foo(i);
  }
  cilk_sync;  Sync for all calls spawned within block A
  bar();
```



join点

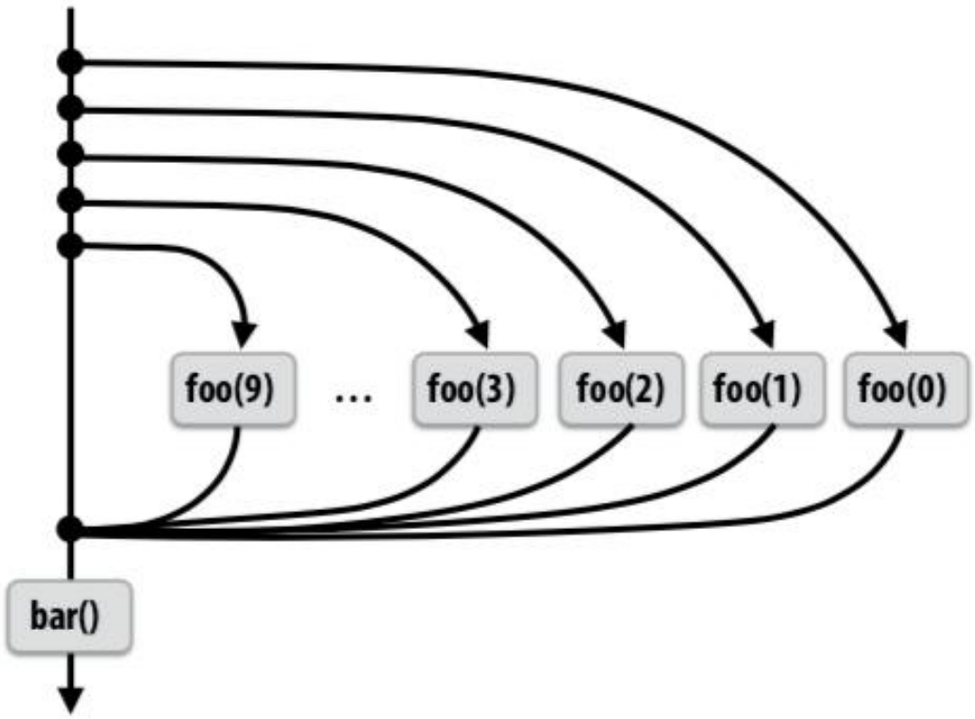


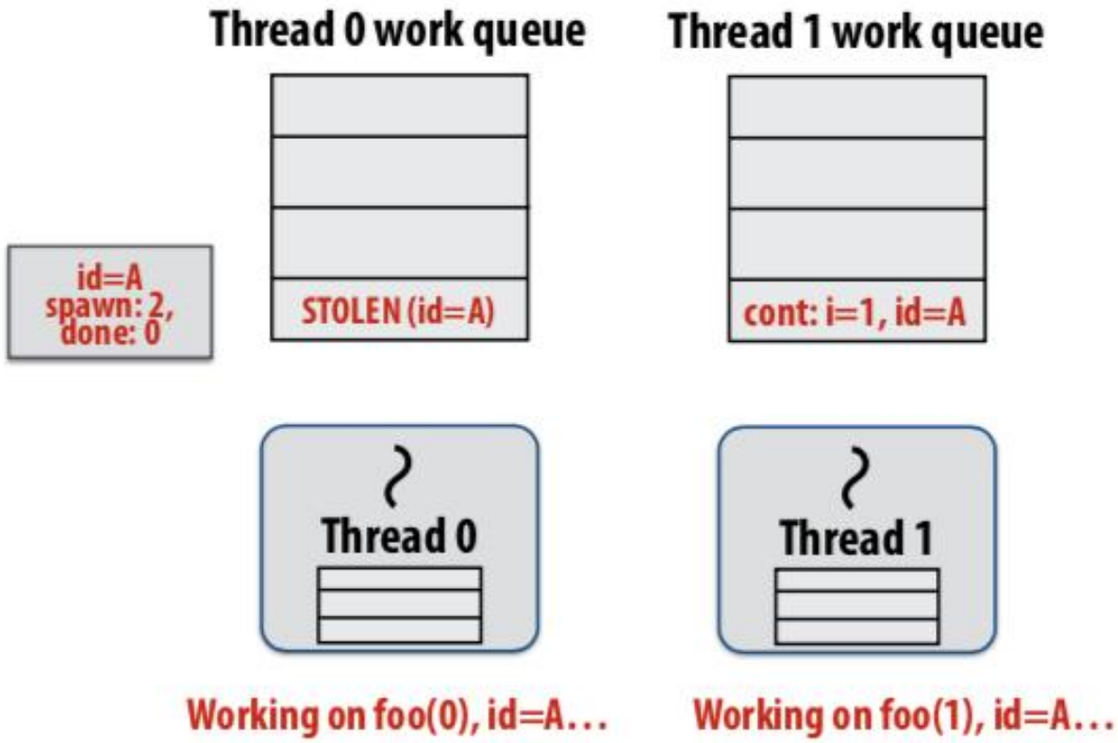
示例 2：贪心策略

- **当启动 fork 的线程空闲时，它会寻找新的工作。**
- **最后一个到达join点的线程，在同步后继续执行剩下的任务**

同步的实现: 贪心策略

```
block (id: A)
for (int i=0; i<10; i++) {
  cilk_spawn foo(i);
}
cilk_sync;  Sync for all calls spawned within block A
bar();
```

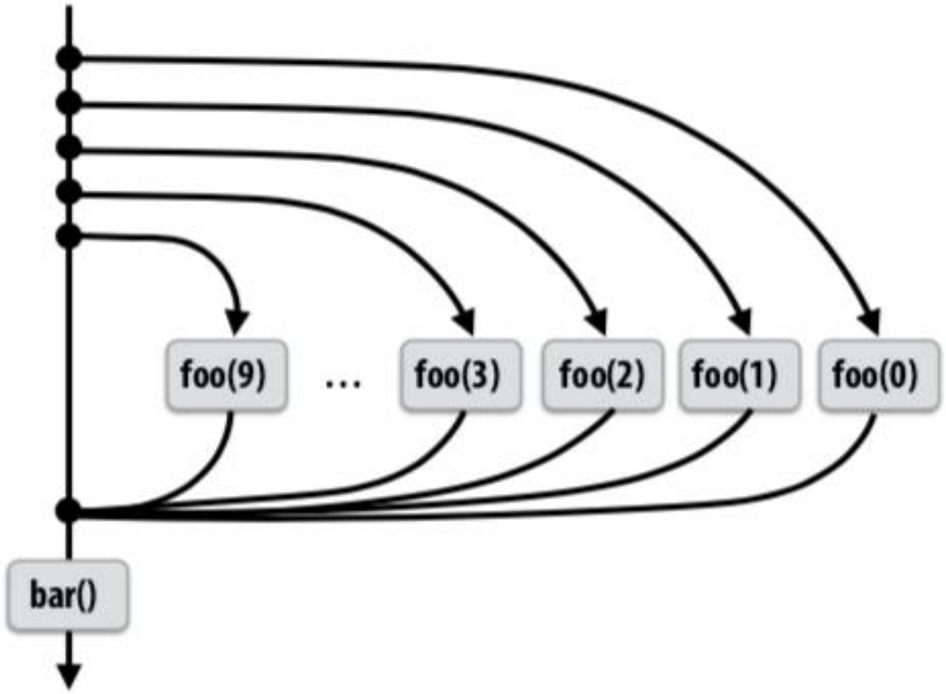


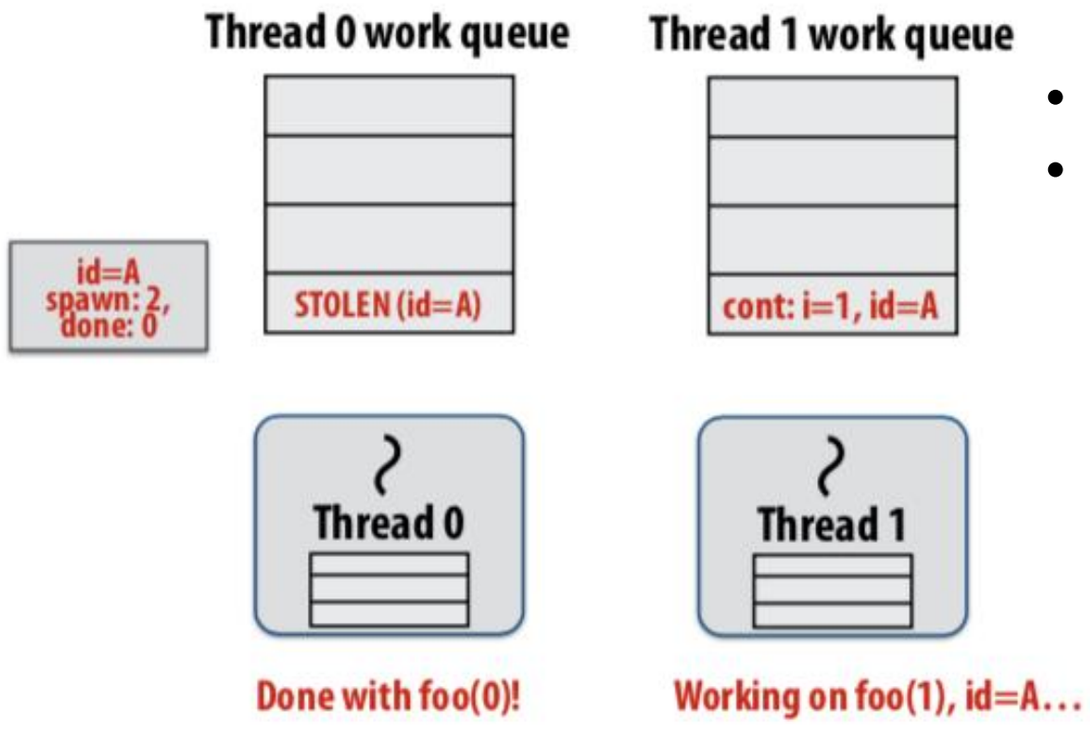


空闲线程 1 从繁忙线程 0 窃取

同步的实现: 贪心策略

```
block (id: A)
  for (int i=0; i<10; i++) {
    cilk_spawn foo(i);
  }
  cilk_sync;  Sync for all calls spawned within block A
  bar();
```

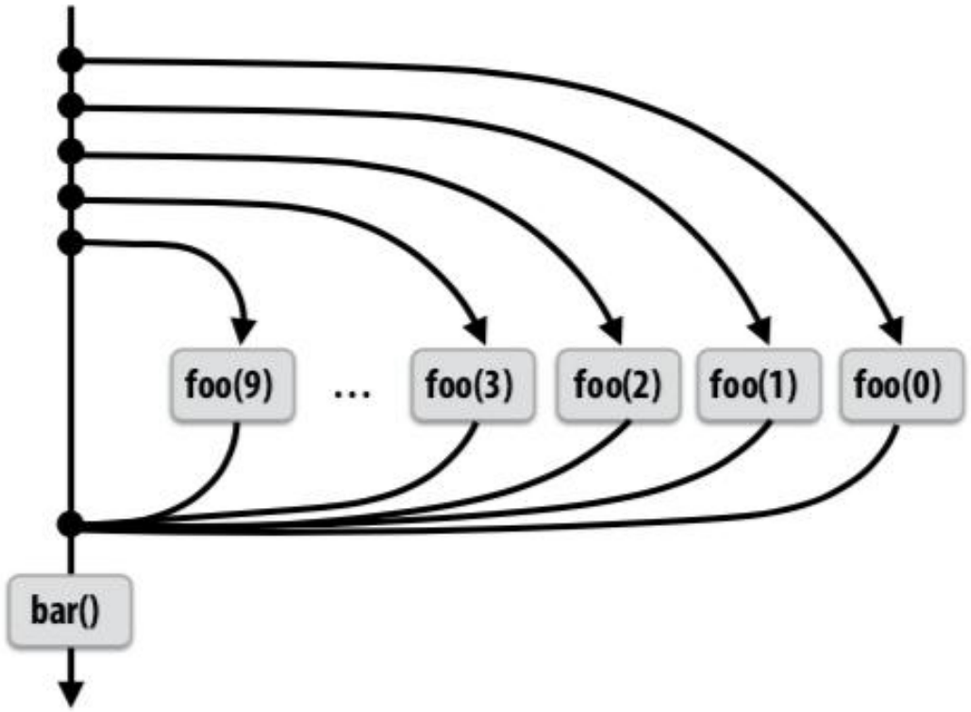


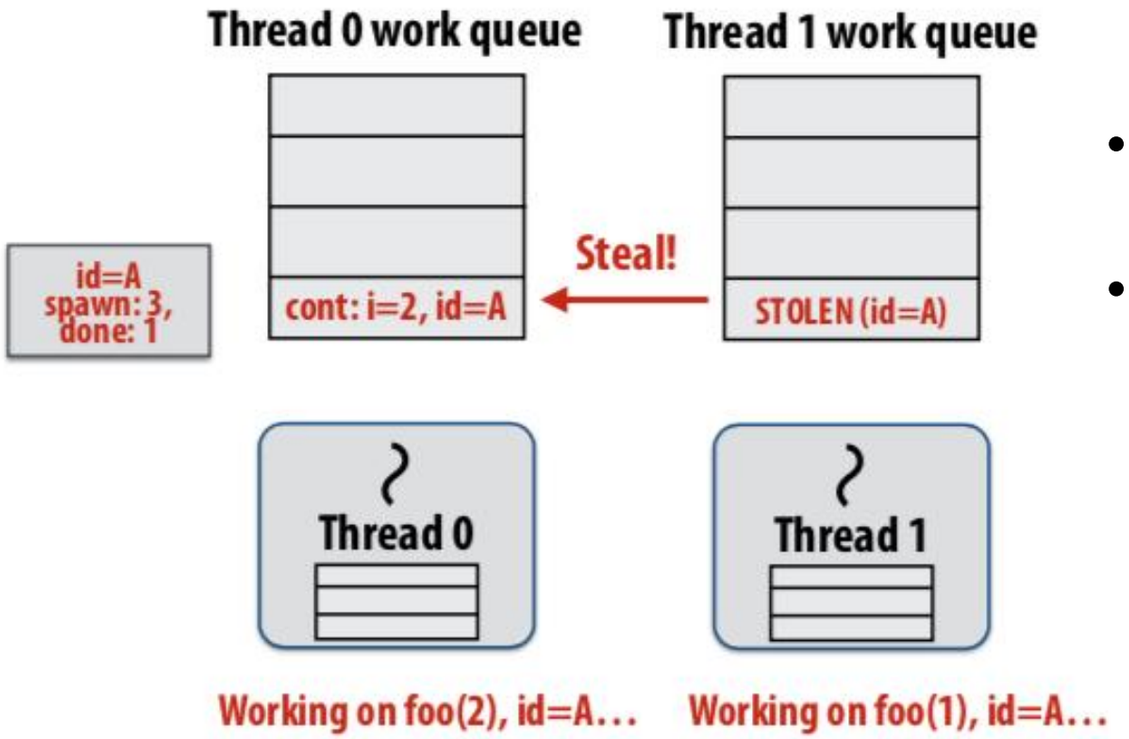


- **Thread 0完成了foo(0)**
- **线程0已经没有剩余任务要处理，因此开始寻找偷窃子任务的机会**

同步的实现: 贪心策略

```
block (id: A)
  for (int i=0; i<10; i++) {
    cilk_spawn foo(i);
  }
  cilk_sync;  Sync for all calls spawned within block A
  bar();
```

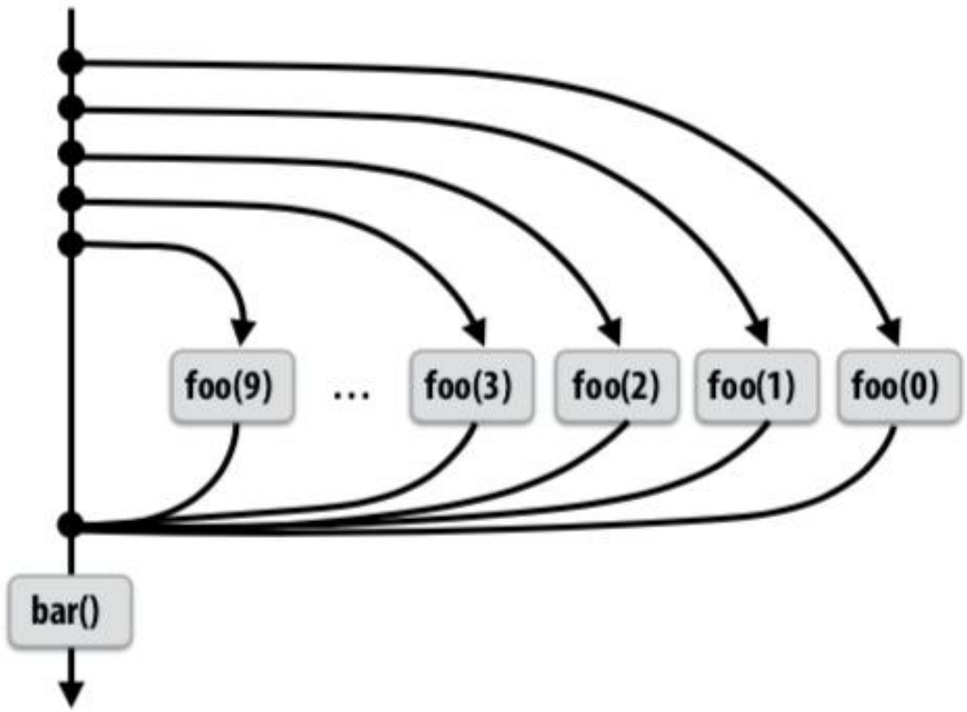


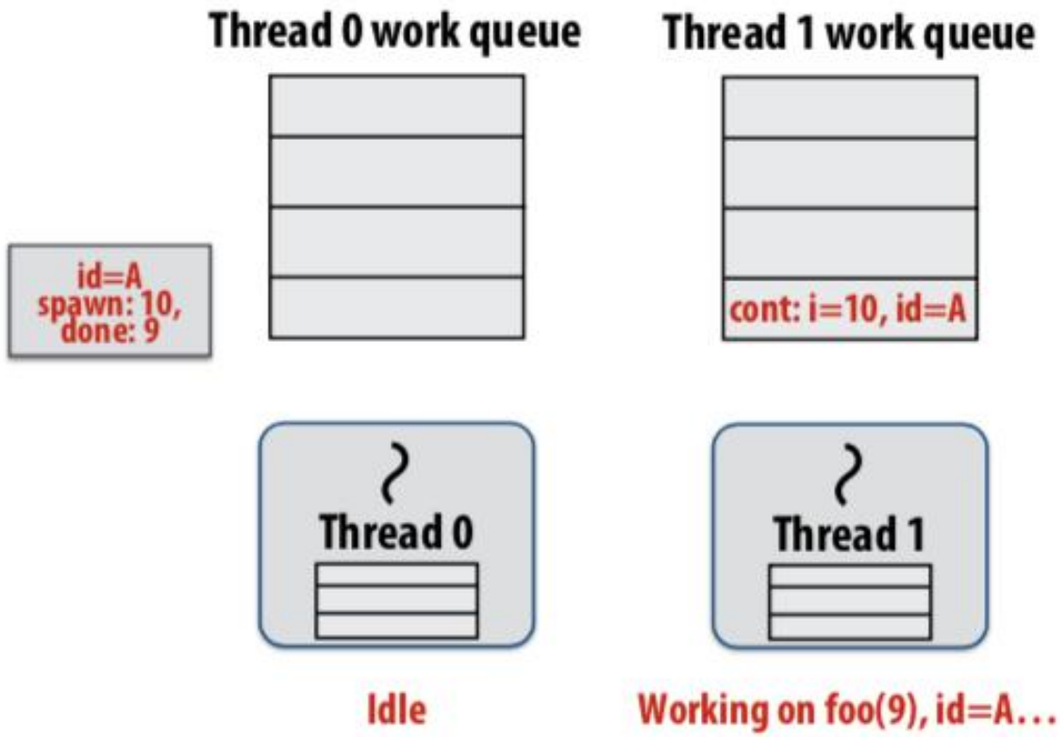


- **Thread 0从Thread1偷取了foo(2)，并开始执行**
- **注意描述符中，派生数量为3，完成数量为1**

同步的实现: 贪心策略

```
block (id: A)
  for (int i=0; i<10; i++) {
    cilk_spawn foo(i);
  }
  cilk_sync;  Sync for all calls spawned within block A
  bar();
```

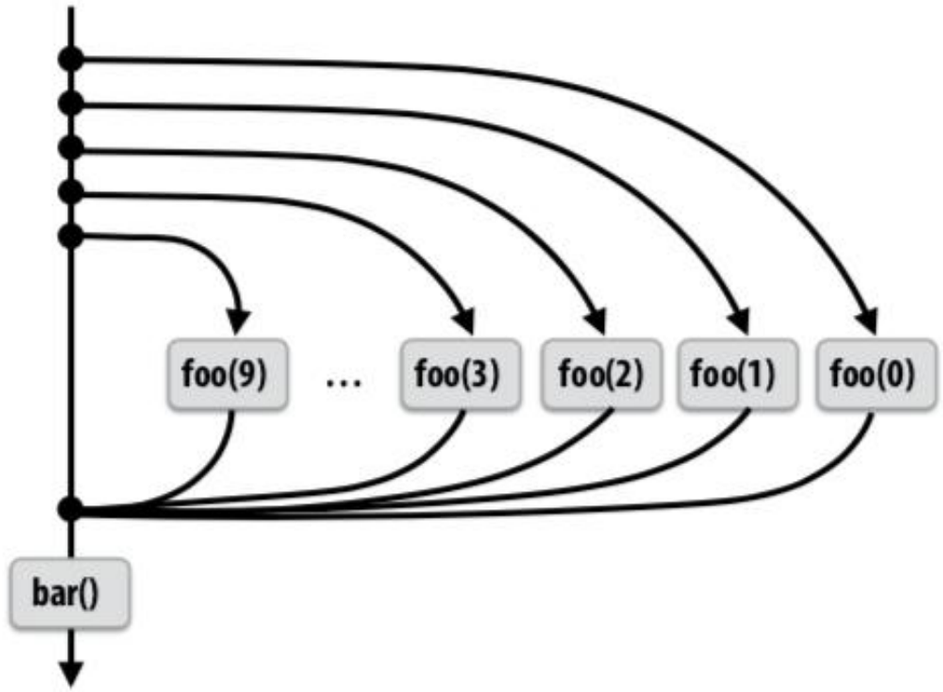


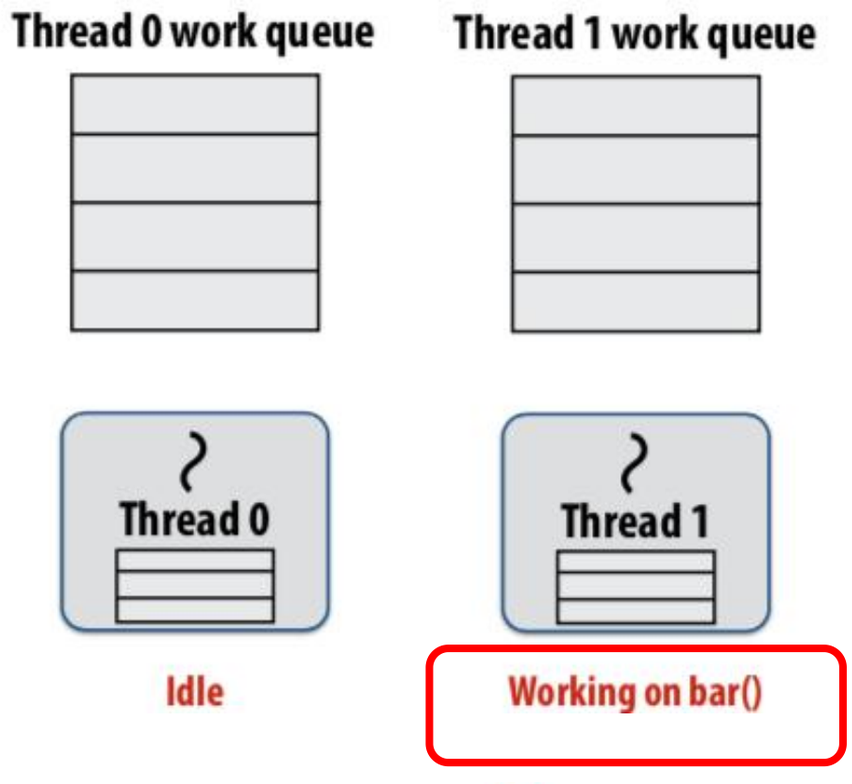


- **假设Thread 1是最后一个完成block A派生子任务的线程**

同步的实现: 贪心策略

```
block (id: A)
  for (int i=0; i<10; i++) {
    cilk_spawn foo(i);
  }
  cilk_sync;  Sync for all calls spawned within block A
  bar();
```





- 由于，Thread 1是最后一个完成block A派生子任务的线程
- 所以，Thread 1目前开始执行剩余任务，开始执行bar()
- 注意：对Block A的描述符已经被释放了。

Cilk 使用贪心的 join调度策略

- 贪心join 调度政策
  - 如果处于空闲状态，所有线程总是尝试窃取别的线程待完成的子任务（如果系统中没有要窃取的工作，线程只能空闲）
- 调度开销的问题
  - 记录进程间的任务窃取和管理同步点的开销，仅在在**子任务发生窃取时**出现
  - 利用局部性降低开销：如果**计算量较大的任务（切分任务的粒度）**被窃取，以上开销出现次数会降低（开销会降低）
    - 大部分时间，线程只会从其本地作业队列中执行本地任务

总结

- Fork-join 并行性：一种表达分治（divid-and-conquer）算法的自然方式
  - 讨论了 Cilk++。OpenMP 也有 fork/join 原语
- Cilk++ 运行时利用局部性感知工作窃取调度策略（locality-aware work stealing scheduler），完成了对 spawn/sync 抽象的实现
  - 始终运行派生的child进程(continuation stealing)，通过窃取continuation，来窃取下一轮迭代任务
  - 贪心的 join调度策略（各个线程都不用等待其他线程，立即寻找其他工作来窃取）