Cilk

Design philosophy

- Integrate with C and C++, uses language extensions
- Target shared memory machines
- User identifies and specifies parallelism, Cilk manages it
  - User identifies function invocations that can execute independently - *spawn*
  - Cilk generates the code to support the parallelism
  - Synchronization is available to control parallel execution
  - Cilk maintains a work queue to efficiently exploit parallelism

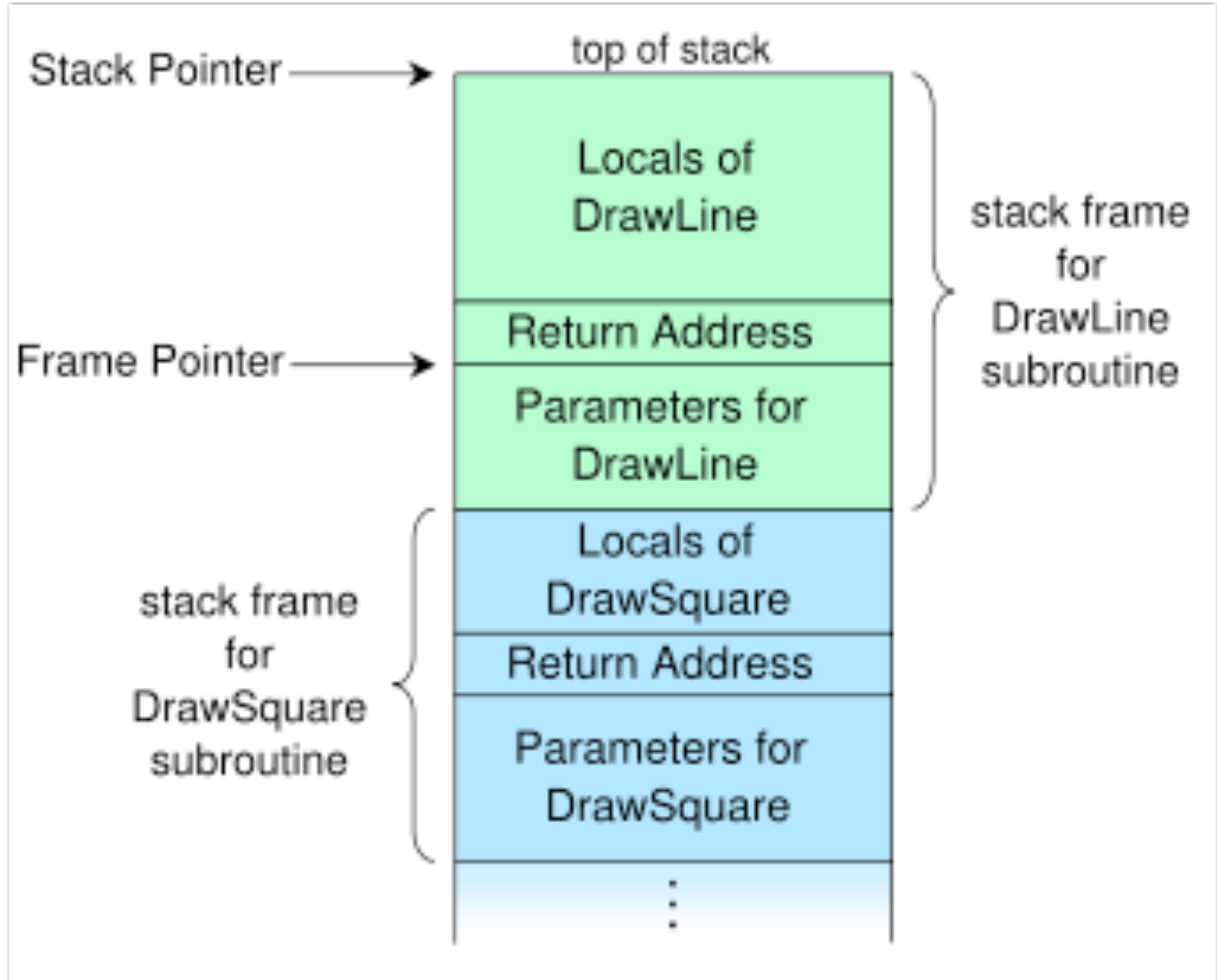
frames or *stack frames*

Stack frames are essential to modern (i.e. since the early 1960s) function invocation

Allow storage to be created that is

- local to an invocation in sequential programs
- automatically removed when the invocation leaves

Allows separate invocations of a function to have a unique identity



Supports recursion and clean returns from deep chains of function calls

Let's look at a simple Cilk function

```
01 cilk int fib (int n)
02 {
03     if (n < 2) return n;
04     else
05     {
06       int x, y;
07
08       x = spawn fib (n-1);
09       y = spawn fib (n-2);
10
11       sync;
12
13       return (x+y);
14     }
15 }
```

Cilk keyword identifies this as a *Cilk* function operating under Cilk rules

But before talking about how Cilk would execute this, let's review how this would be executed sequentially given the call **fib(2);**

Each invocation of **fib** has its own stack frame, and so there is a frame created for **fib(2)**

Because of line **06**, space is created on the frame for **x** and **y**

At line **08** a new frame is created and **fib(1)** is called

Execution begins for **fib(1)**. After **03** is executed, the value of **n** (1) is placed into **fib(2)**'s **x** variable

Execution continues to **09**, the value **0** is placed into **y**, the values are added and returned to the return variable at the call to **fib(2)**

Let’s look at a simple Cilk function

```
01 cilk int fib (int n)
02 {
03     if (n < 2) return n;
04     else
05     {
06        int x, y;
07
08        x = spawn fib (n-1);
09        y = spawn fib (n-2);
10
11        sync;
12
13        return (x+y);
14     }
15 }
```

Let’s now see how Cilk would execute this in parallel

Each invocation of **fib** has its own stack frame, and so there is a frame created for **fib(2)**

When statement **08** is reached, the **spawn** keyword says that **fib(1)** can safely execute in parallel with the rest of the program on a different processor

- Cilk runtime assigns invocation to a processor
- That processor creates a stack frame for **fib(1)**
- **fib(1)** executes in parallel with the rest of the code
- when **fib(1)** finishes it returns, placing **1** into **x**

While the red actions are happening, the green thread executes statement **09**, spawing **fib(0)**

- Actions analogous to what happened with **fib(1)** occur, except with 0 and **y**

What synchs are for

```
01 cilk int fib (int n)
02 {
03    if (n < 2) return n;
04    else
05    {
06      int x, y;
07
08      x = spawn fib (n-1);
09      y = spawn fib (n-2);
10
11      sync;
12
13      return (x+y);
14    }
15 }
```

After spawning **fib(1)** and **fib(0)** execution proceeds to the **sync** statement at line **11**.

The **synch** statement stops processing until *all* function invocations spawned by *this* function (fib(2)) *with this frame* have reached it. It is a form of barrier.

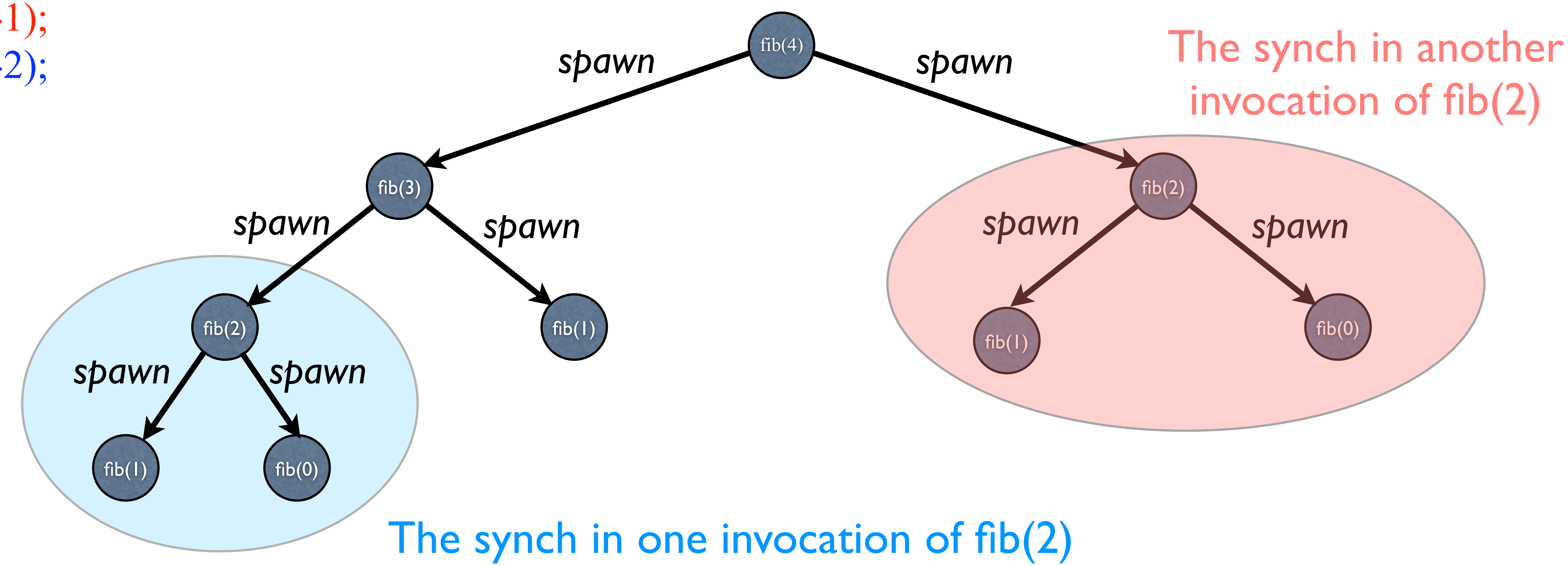
The **synch** ensures that both spawns have returned before the **return** in statement **13** is executed. Not doing this would create a race and an incorrect program.

How synchs work

```
01 cilk int fib (int n)
02 {
03     if (n < 2) return n;
04     else
05     {
06       int x, y;
07
08       x = spawn fib (n-1);
09       y = spawn fib (n-2);
10
11       sync;
12
13       return (x+y);
14     }
15 }
```

After spawning **fib(1)** and **fib(0)** execution proceeds to the **sync** statement at line **11**.

The **synch** statement stops processing until *all* function invocations spawned by *this* function (fib(2)) *with this frame* have reached it. It is a form of barrier.

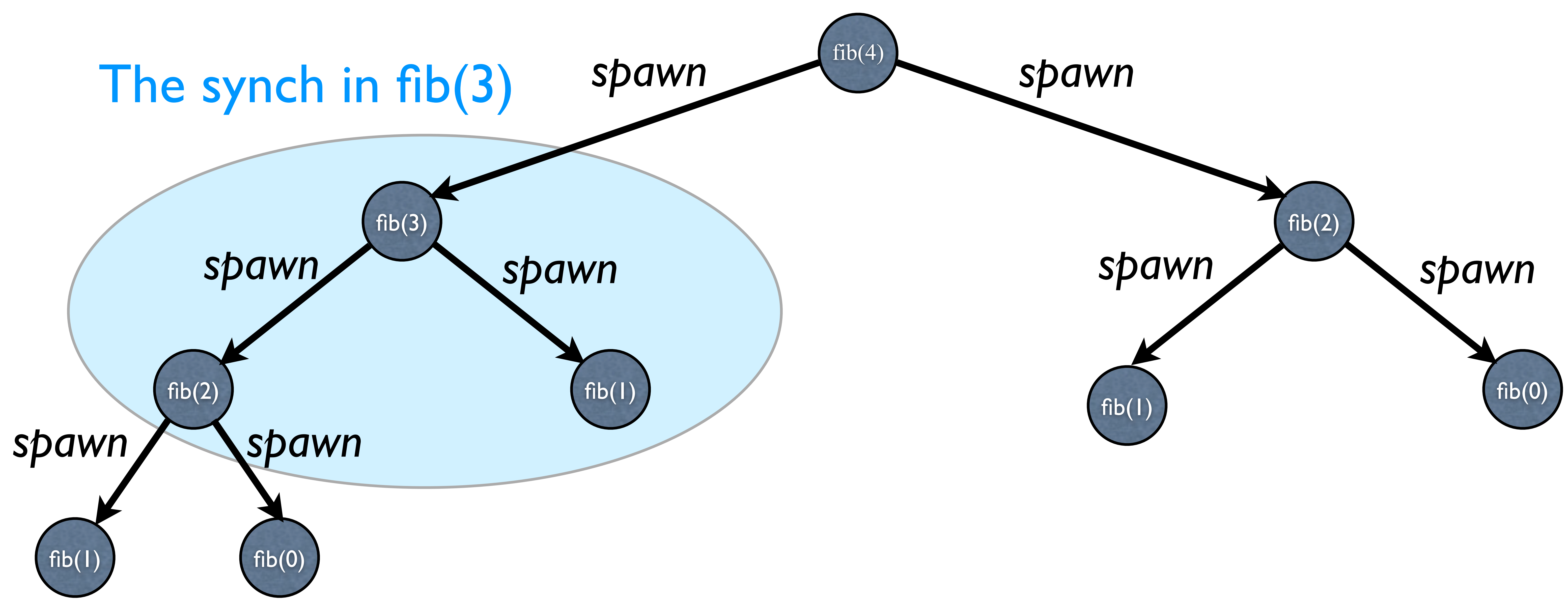


How synchs work

```
01 cilk int fib (int n)
02 {
03    if (n < 2) return n;
04    else
05    {
06      int x, y;
07
08      x = spawn fib (n-1);
09      y = spawn fib (n-2);
10
11      sync;
12
13      return (x+y);
14    }
15 }
```

After spawning **fib(1)** and **fib(0)** execution proceeds to the **sync** statement at line **11**.

The **synch** statement stops processing until *all* function invocations spawned by *this* function (fib(2)) *with this frame* have reached it. It is a form of barrier.

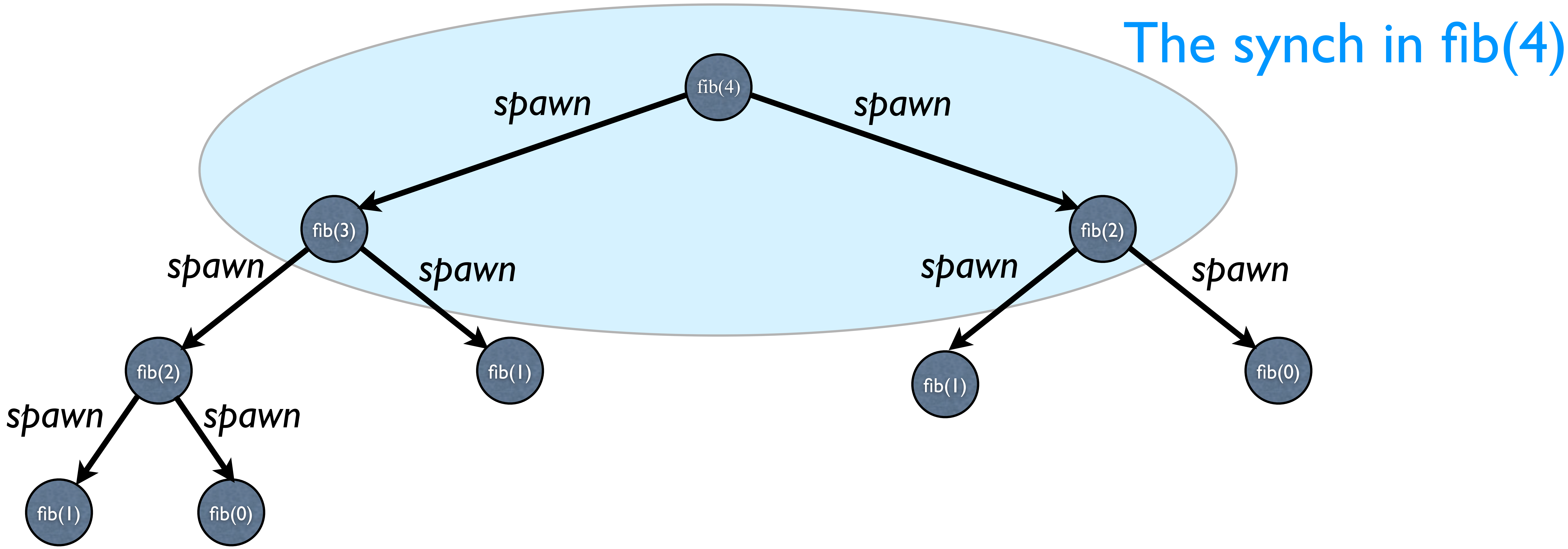


How synchs work

```
01 cilk int fib (int n)
02 {
03     if (n < 2) return n;
04     else
05     {
06       int x, y;
07
08       x = spawn fib (n-1);
09       y = spawn fib (n-2);
10
11       sync;
12
13       return (x+y);
14     }
15 }
```

After spawning **fib(1)** and **fib(0)** execution proceeds to the **sync** statement at line **11**.

The **synch** statement stops processing until *all* function invocations spawned by *this* function (fib(2)) *with this frame* have reached it. It is a form of barrier.

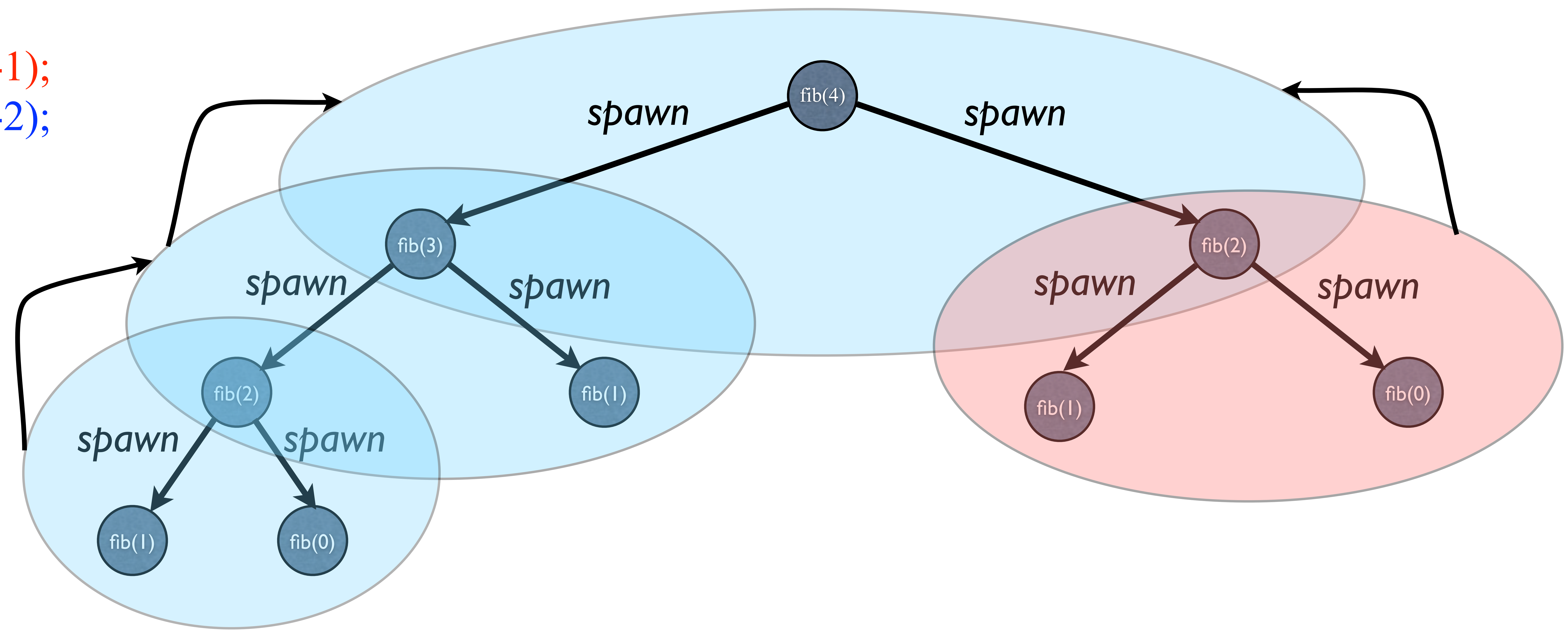


How synchs work

```
01 cilk int fib (int n)
02 {
03    if (n < 2) return n;
04    else
05    {
06      int x, y;
07
08      x = spawn fib (n-1);
09      y = spawn fib (n-2);
10
11      sync;
12
13      return (x+y);
14    }
15 }
```

After spawning **fib(1)** and **fib(0)** execution proceeds to the **sync** statement at line **11**.

The **synch** statement stops processing until *all* function invocations spawned by *this* function (fib(2)) *with this frame* have reached it. It is a form of barrier.



Inlets

```
cilk int fib (int n) {

  int x = 0;
  inlet void summer (int result) {
    x += result;
    return;
  }
  if (n<2) return n;
  else {
    summer(spawn fib (n-1));
    summer(spawn fib (n-2));
    sync;
    return (x);
}
```

Inlets are Cilk constructs that process return values before they are returned.

Inlets always execute atomically

Cilk normally requires a procedure to be spawned as a separate statement and continues with its execution, this rule is relaxed for inlets

fib(n-1) is invoked;

the parent continues executing *after* the inlet

when **fib(n-1)** returns its thread passes control to **summer**

When **summer** is finished, the thread that executed it waits at the **sync**

Implicit Inlets

```
cilk int fib (int n) {

   int x = 0;

   if (n<2) return n;
   else {
      x+= spawn fib (n-1));
      x += spawn fib (n-2));
      sync;
      return (x);
}
```

Give a way of expressing reductions, etc. succinctly

Cannot be mixed with explicit inlets, i.e.

x += summer(spawn(fib(n-1))

would not be legal

aborts

Cilk allows an **abort** statement to appear in an inlet -- it kills all spawned threads of the parent procedure

- They do not die instantly
- They may terminate normally, and return a value
- It is up to the user to handle these situations

Scheduling

- In a sequential execution, when executing a spawn, Cilk will
  - push the frame and program counter of the parent onto a stack
  - execute the spawned procedure
  - dequeue the parent frame and continue its execution

Scheduling - parallel execution

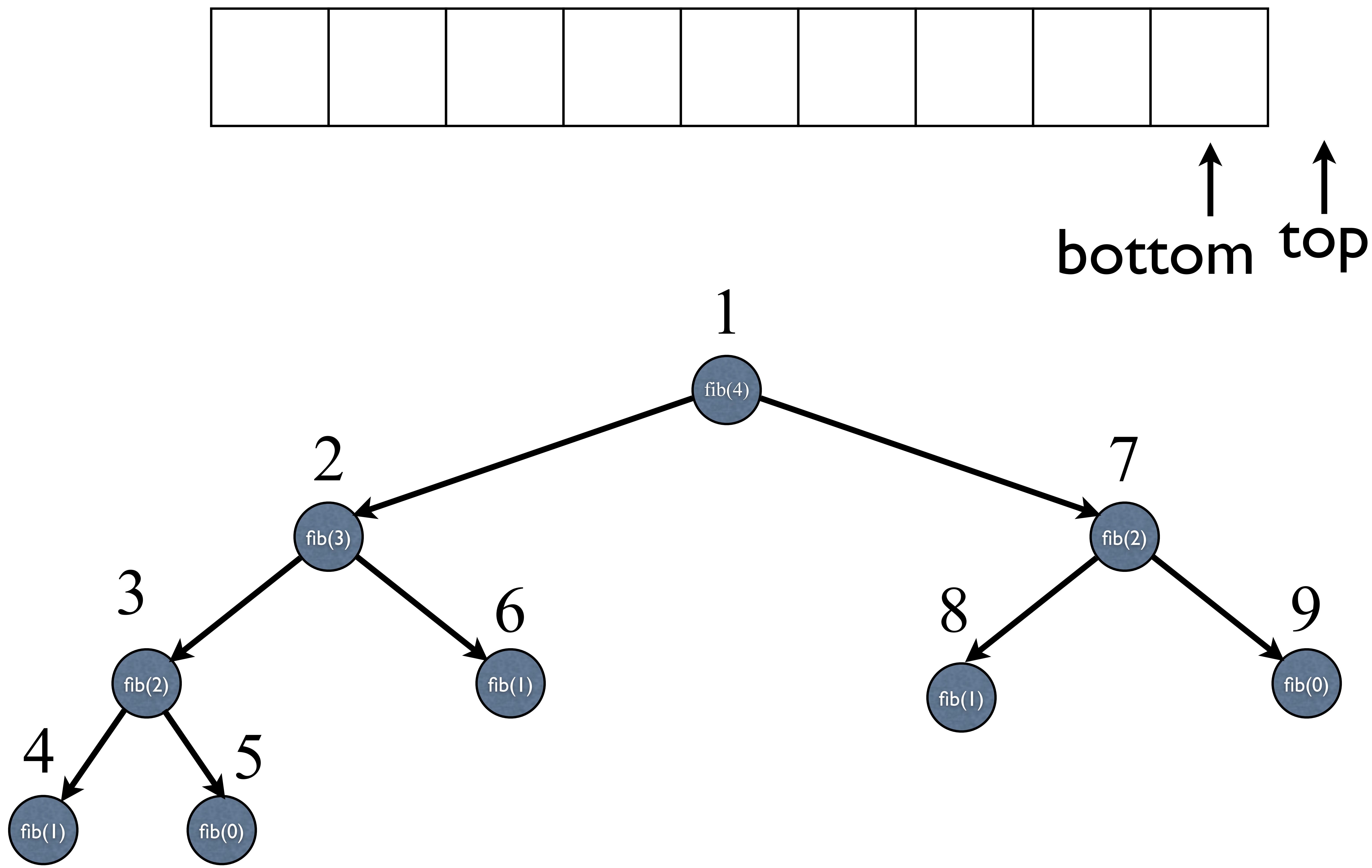
- Each processor maintains a *deque* or *double ended queue*
- When a function is spawned any frames that need to be suspended are placed on the deque
- The processor owning the deque can only remove frames from the end it inserted them
- Other processors may remove from the other end

Scheduling - parallel execution

- When an function is spawned
  - place the parent onto the bottom of the deque/“stack”
  - execute the spawned function, which may place itself onto the bottom of the stack if it spawns functions
- When the function returns, pop work off of the bottom (the frame of the parent of the spawned function)
- If a thread is idle, take work off of the ***top*** of the deque

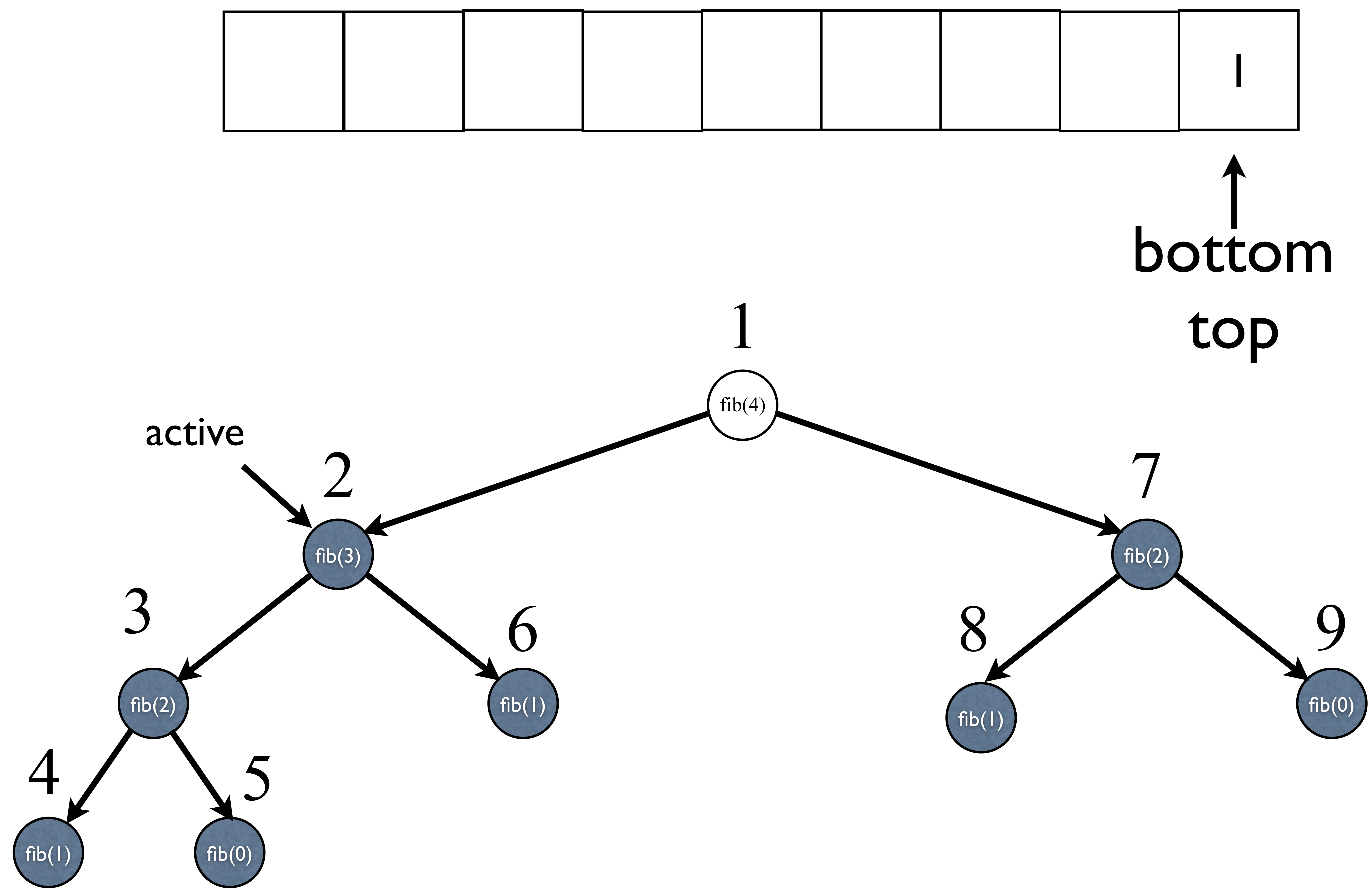
Sequential stack example

```
cilk int fib (int n)
{
   if (n < 2) return n;
   else
   {
     int x, y;

     x = spawn fib (n-1);
     y = spawn fib (n-2);

     sync;

     return (x+y);
   }
}
```



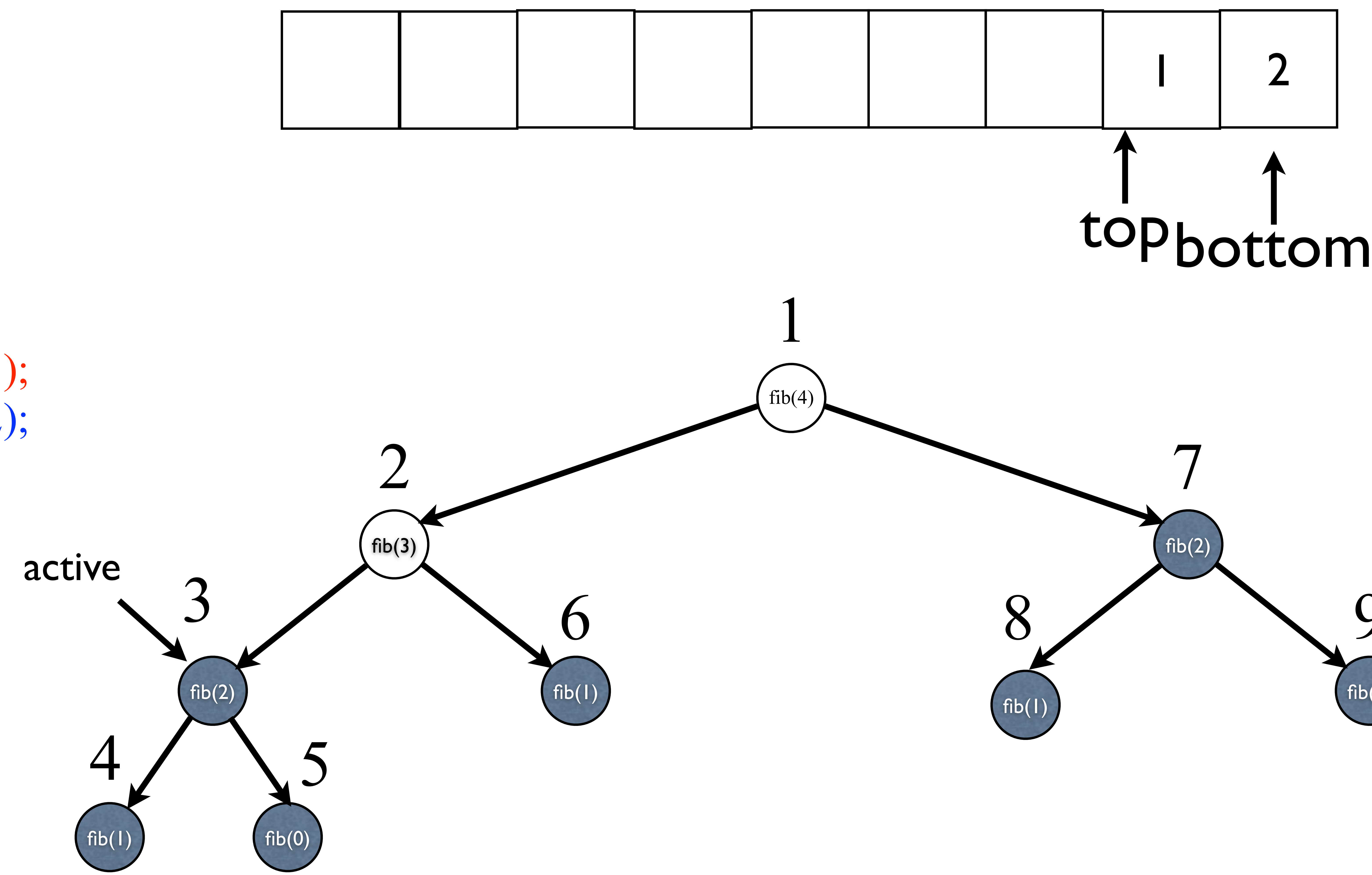
Sequential stack example

```
cilk int fib (int n)
{
   if (n < 2) return n;
   else
   {
     int x, y;

     x = spawn fib (n-1);
     y = spawn fib (n-2);

     sync;

     return (x+y);
   }
}
```



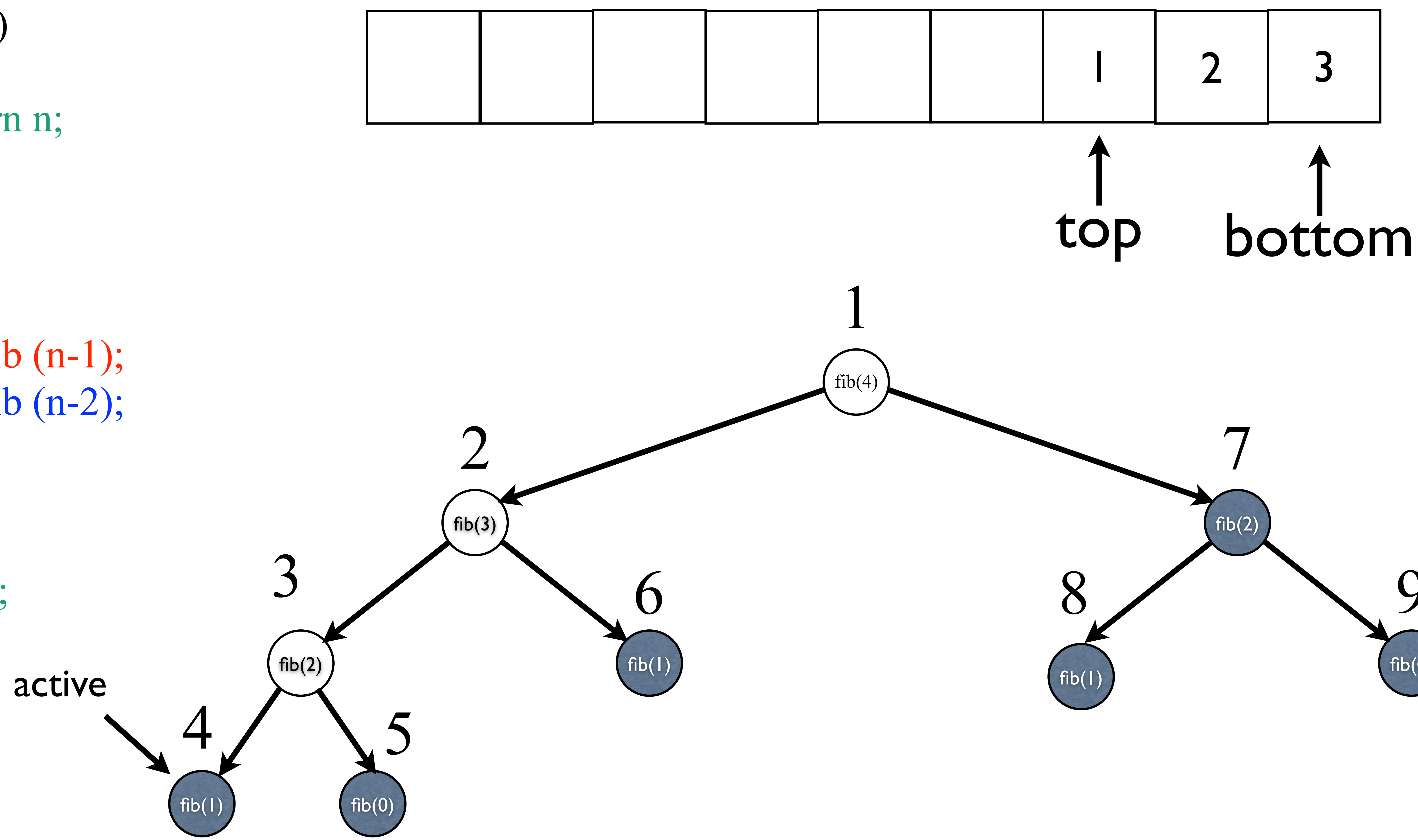
Sequential stack example

```
cilk int fib (int n)
{
   if (n < 2) return n;
   else
   {
      int x, y;

      x = spawn fib (n-1);
      y = spawn fib (n-2);

      sync;

      return (x+y);
   }
}
```



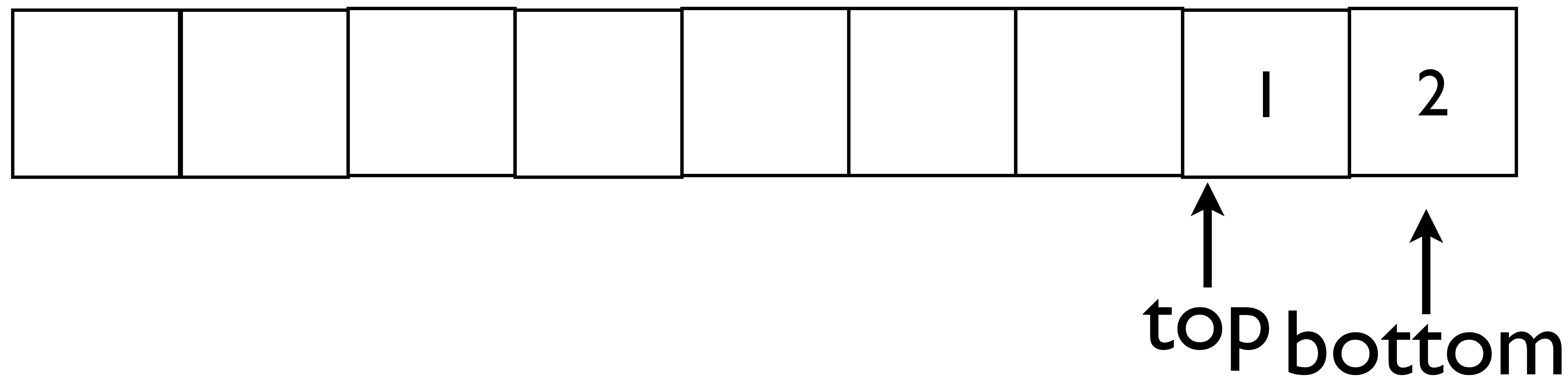
Sequential stack example

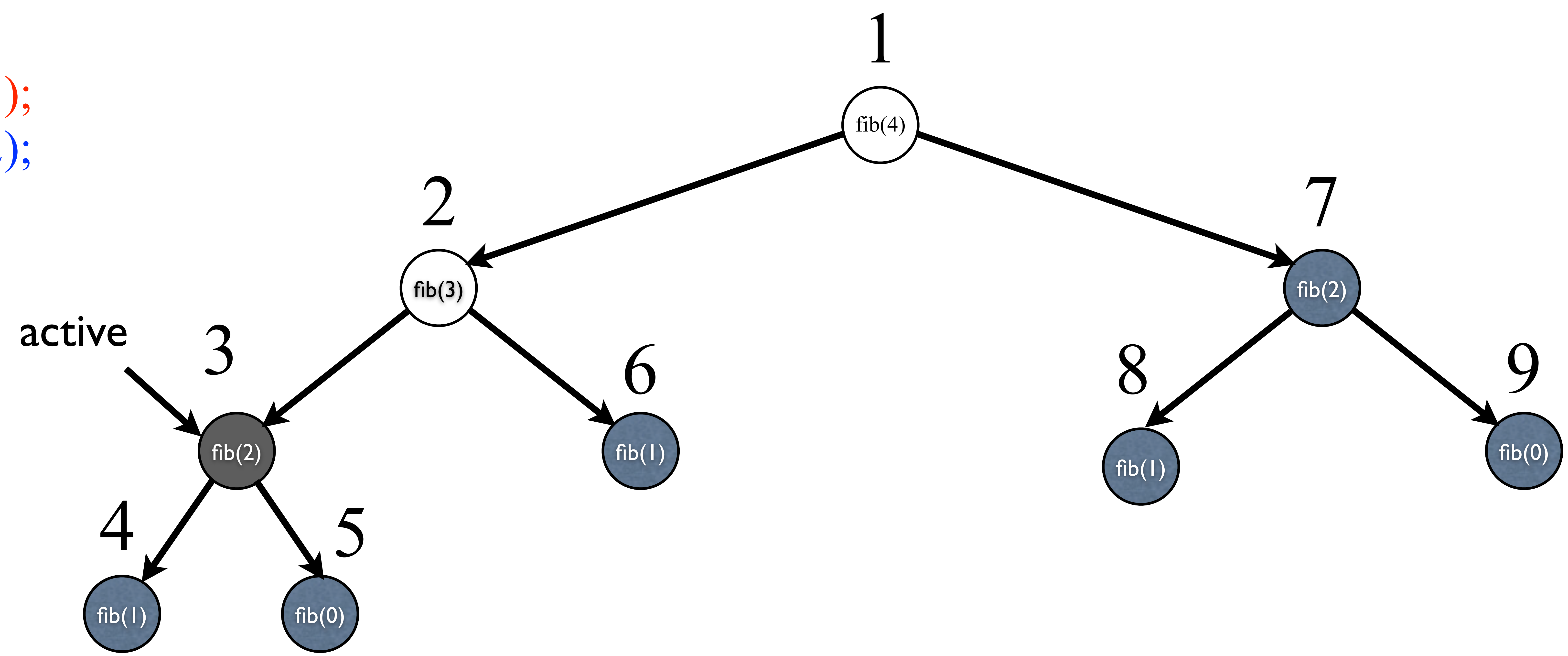
```
cilk int fib (int n)
{
    if (n < 2) return n;
    else
    {
      int x, y;

      x = spawn fib (n-1);
      y = spawn fib (n-2);

      sync;

      return (x+y);
    }
}
```



Sequential stack example

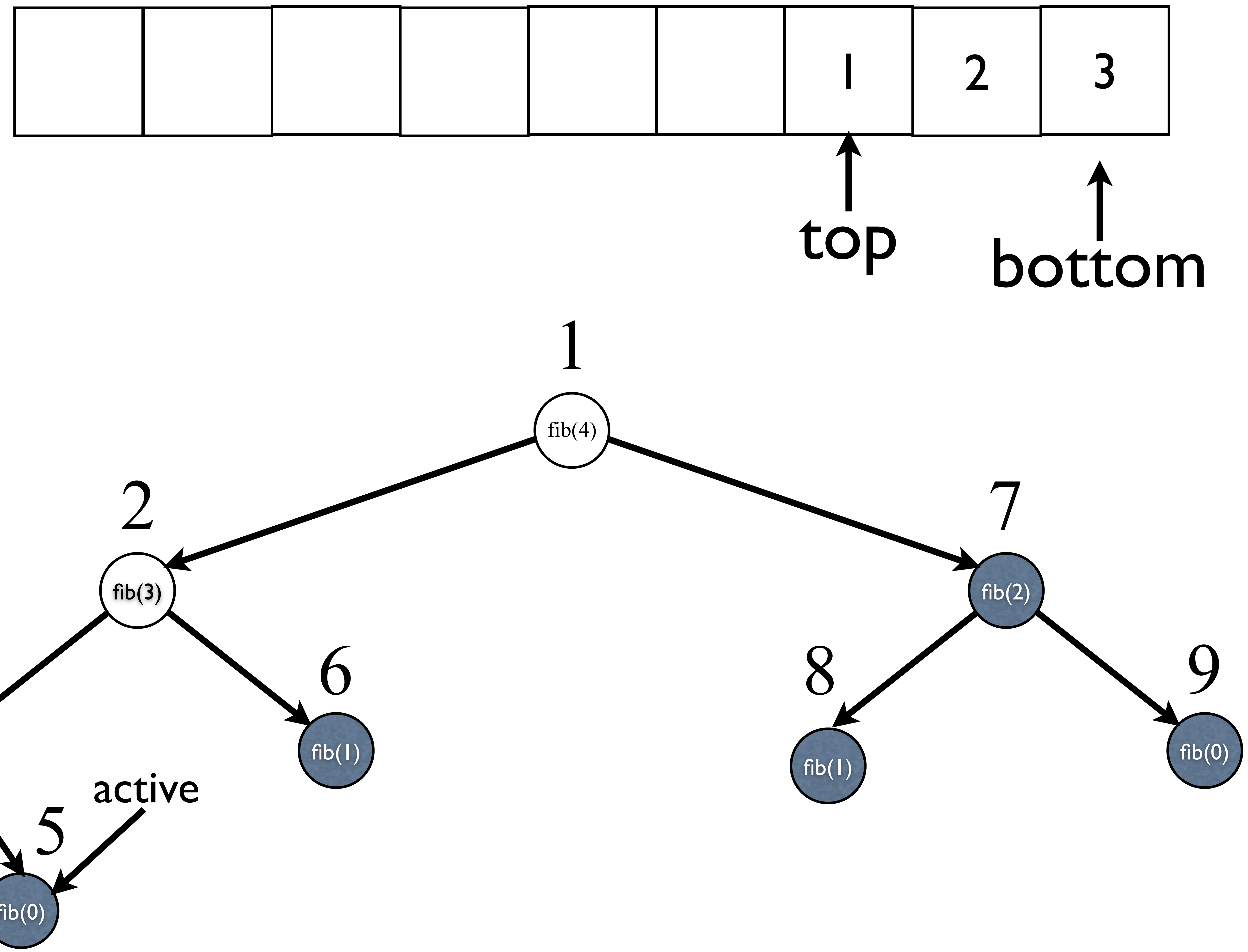
```
cilk int fib (int n)
{
    if (n < 2) return n;
    else
    {
      int x, y;

      x = spawn fib (n-1);
      y = spawn fib (n-2);

      sync;

      return (x+y);
    }
}
```





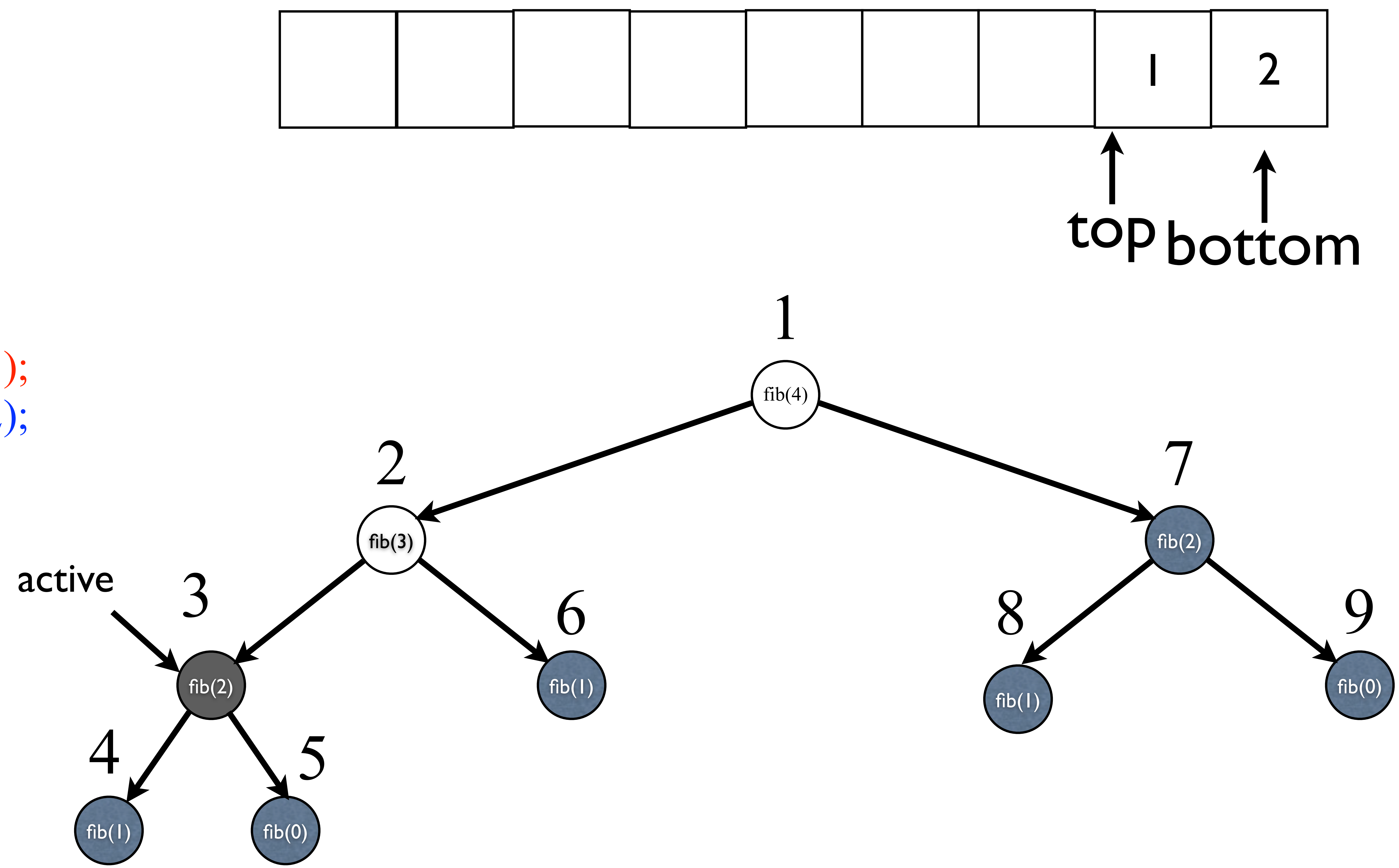
Sequential stack example

```
cilk int fib (int n)
{
   if (n < 2) return n;
   else
   {
     int x, y;

     x = spawn fib (n-1);
     y = spawn fib (n-2);

     sync;

     return (x+y);
   }
}
```



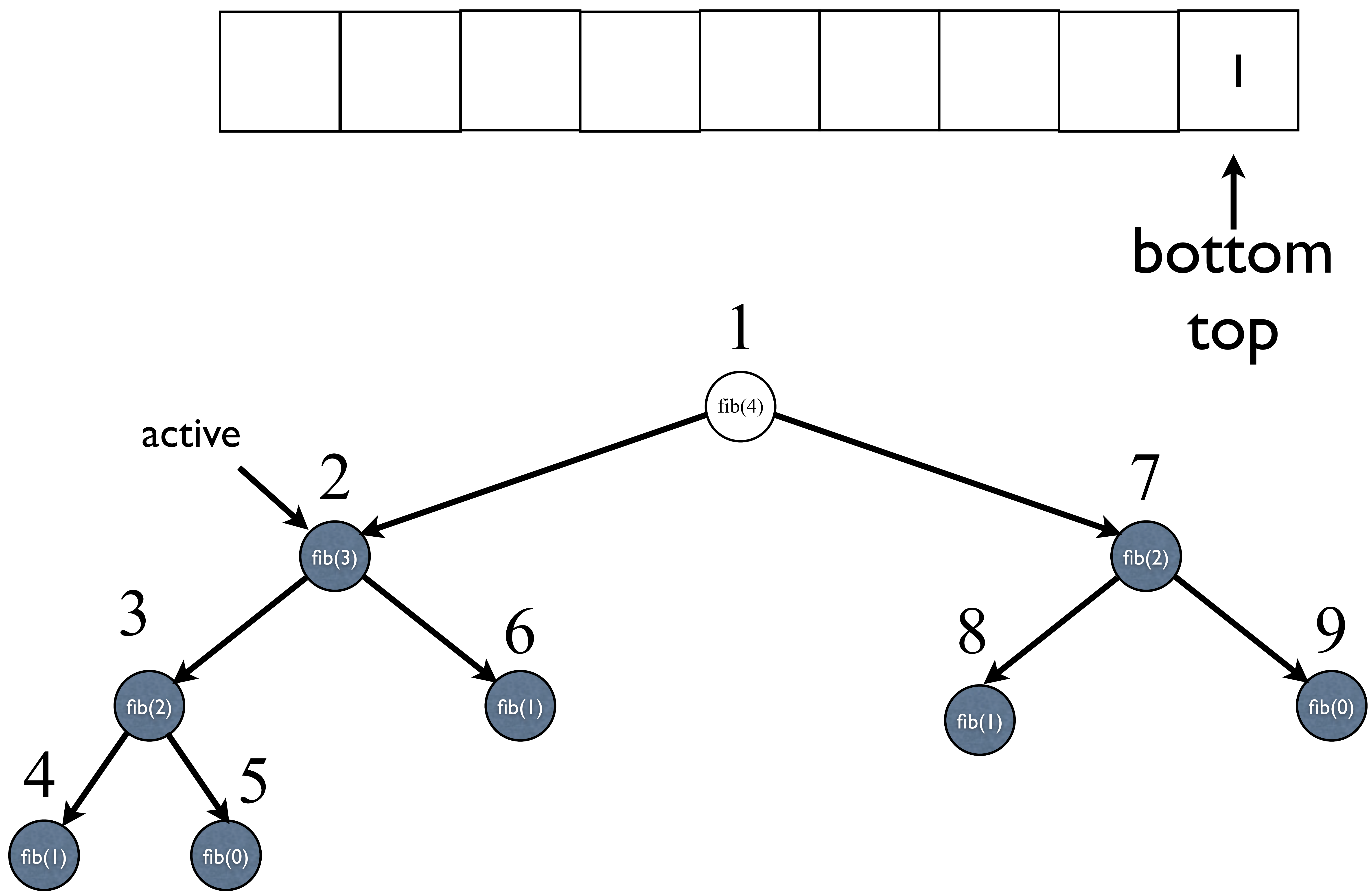
Sequential stack example

```
cilk int fib (int n)
{
    if (n < 2) return n;
    else
    {
      int x, y;

      x = spawn fib (n-1);
      y = spawn fib (n-2);

      sync;

      return (x+y);
    }
}
```



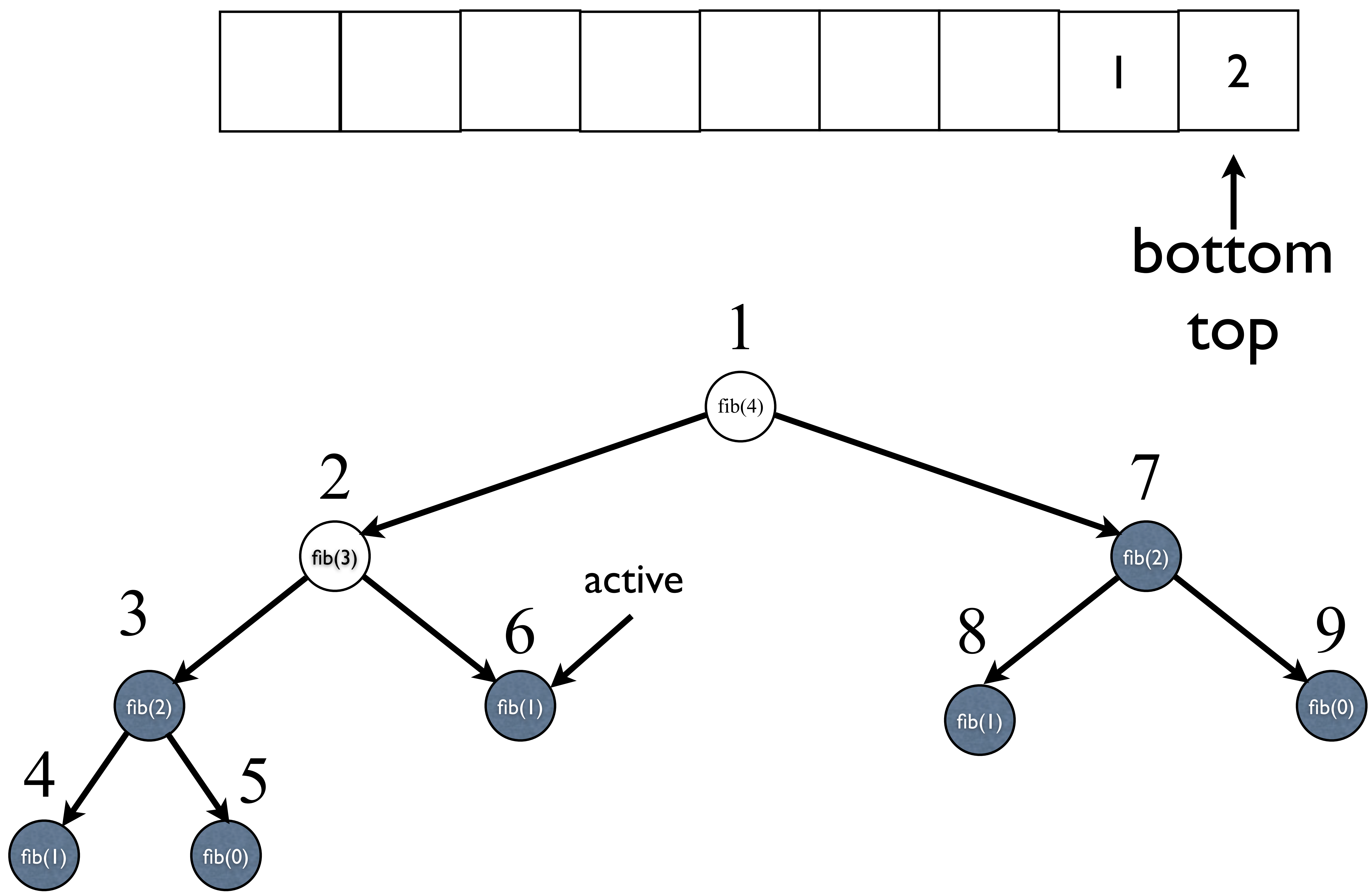
Sequential stack example

```
cilk int fib (int n)
{
    if (n < 2) return n;
    else
    {
        int x, y;

        x = spawn fib (n-1);
        y = spawn fib (n-2);

        sync;

        return (x+y);
    }
}
```



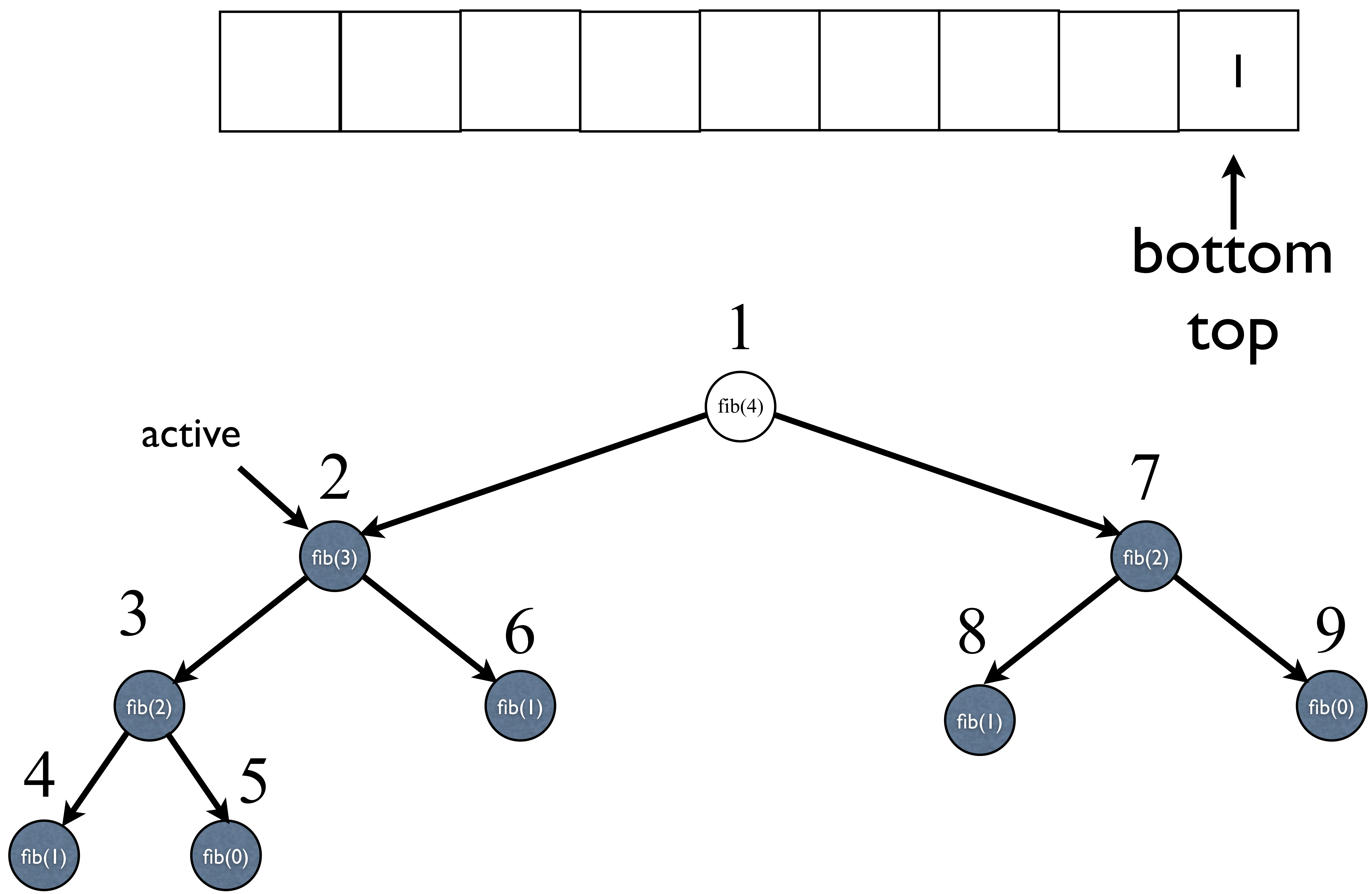
Sequential stack example

```
cilk int fib (int n)
{
   if (n < 2) return n;
   else
   {
     int x, y;

     x = spawn fib (n-1);
     y = spawn fib (n-2);

     sync;

     return (x+y);
   }
}
```



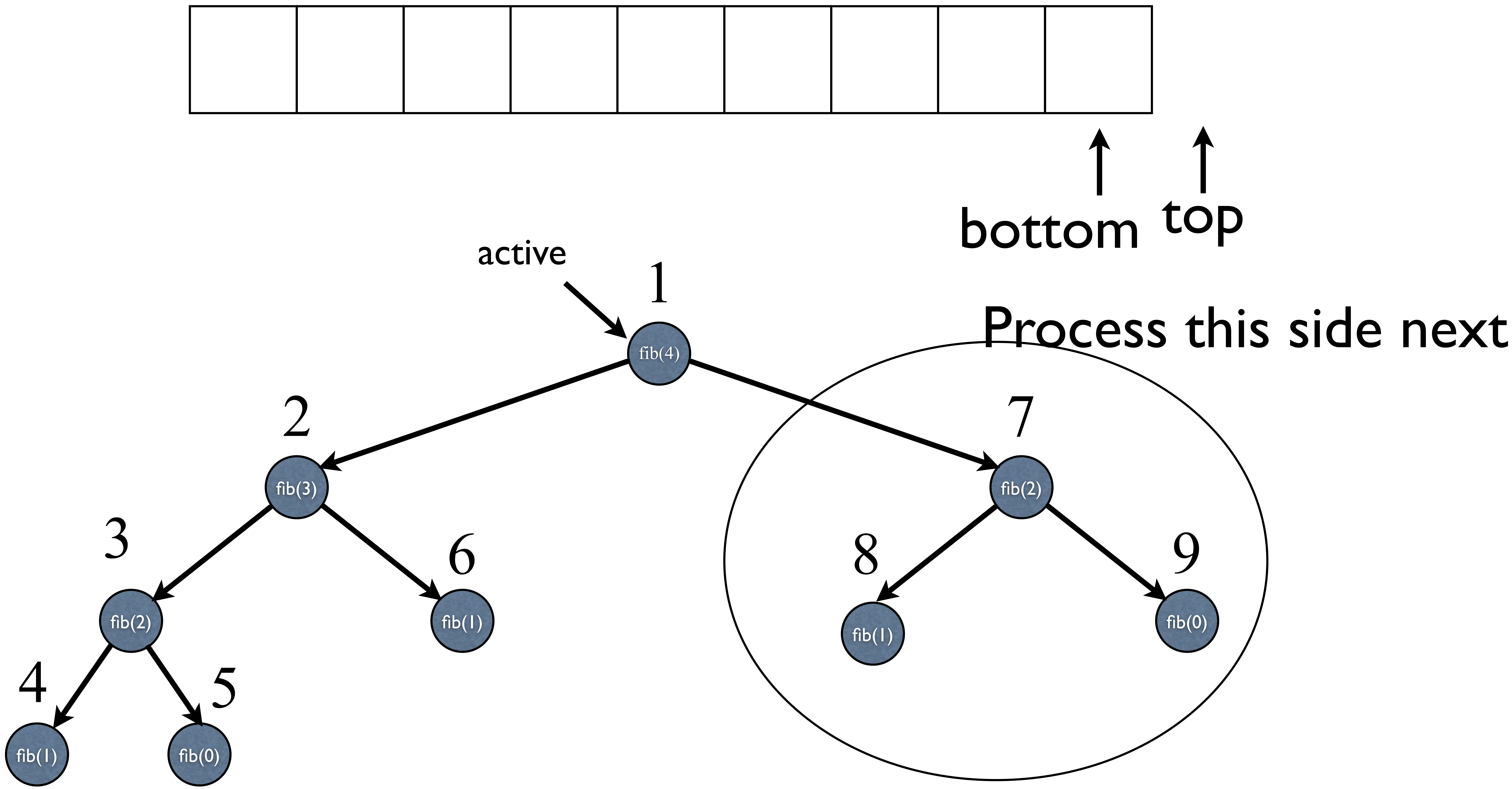
Sequential stack example

```
cilk int fib (int n)
{
    if (n < 2) return n;
    else
    {
       int x, y;

       x = spawn fib (n-1);
       y = spawn fib (n-2);

       sync;

       return (x+y);
    }
}
```

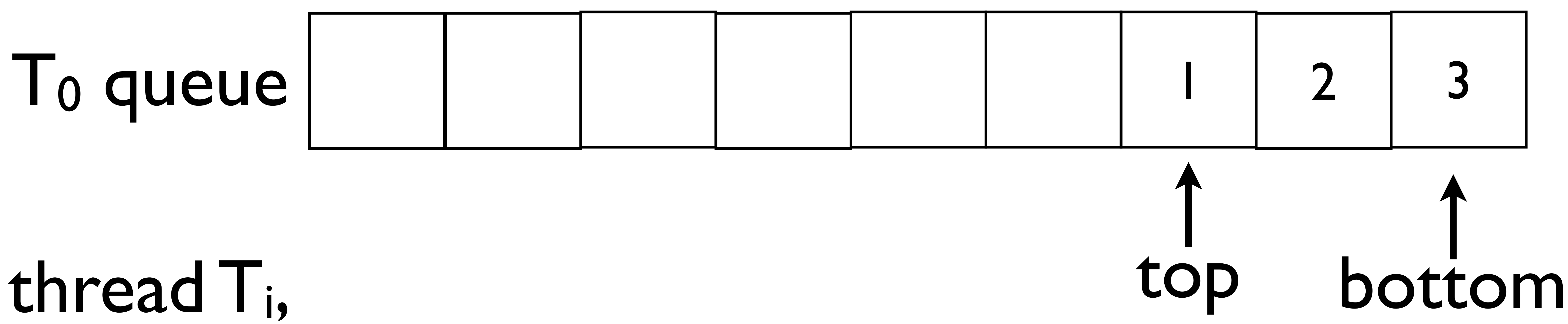


Sequential stack example

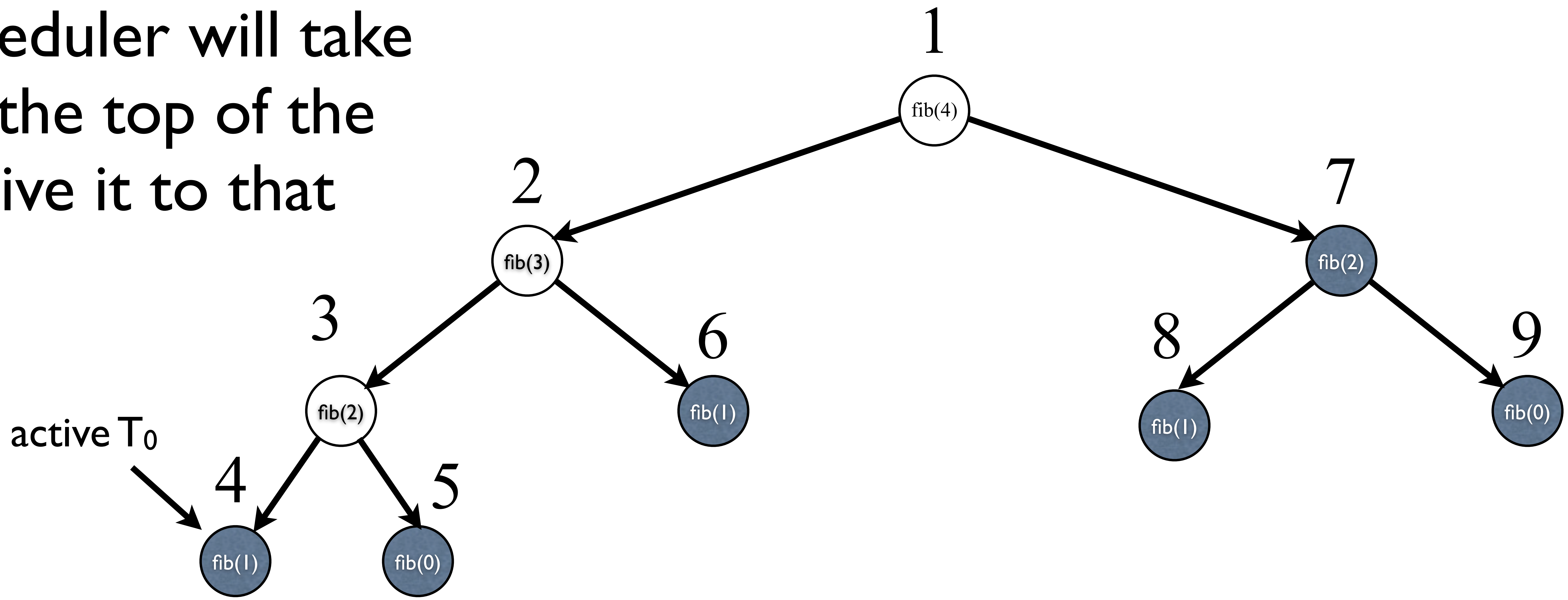
```
cilk int fib (int n)
{
  if (n < 2) return n;
  else
  {
    int x, y;

    x = spawn fib (n-1);
    y = spawn fib (n-2);

    sync;

    return (x+y);
  }
}
```



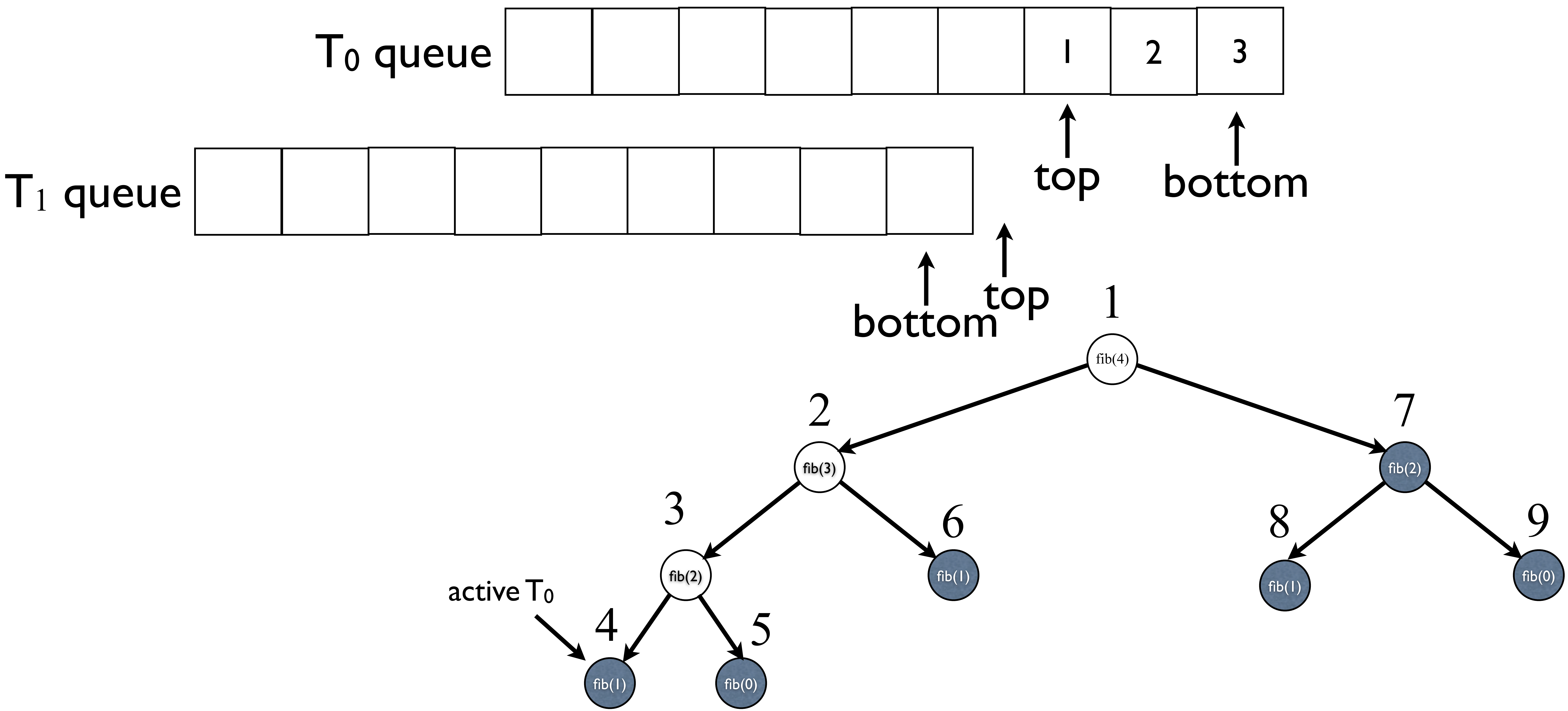
Work stealing example



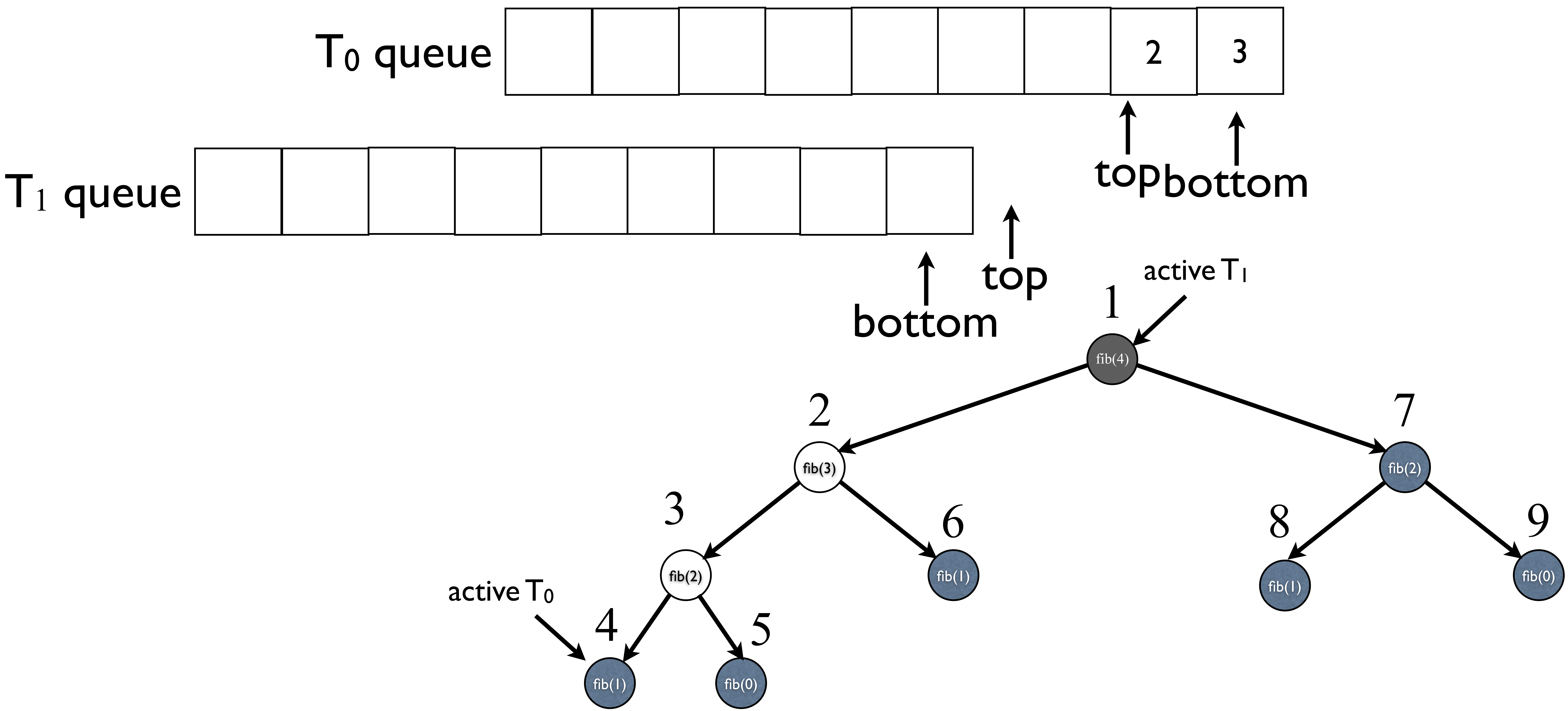
If there is an idle thread T_i, the Cilk scheduler will take work off of the top of the queue and give it to that thread



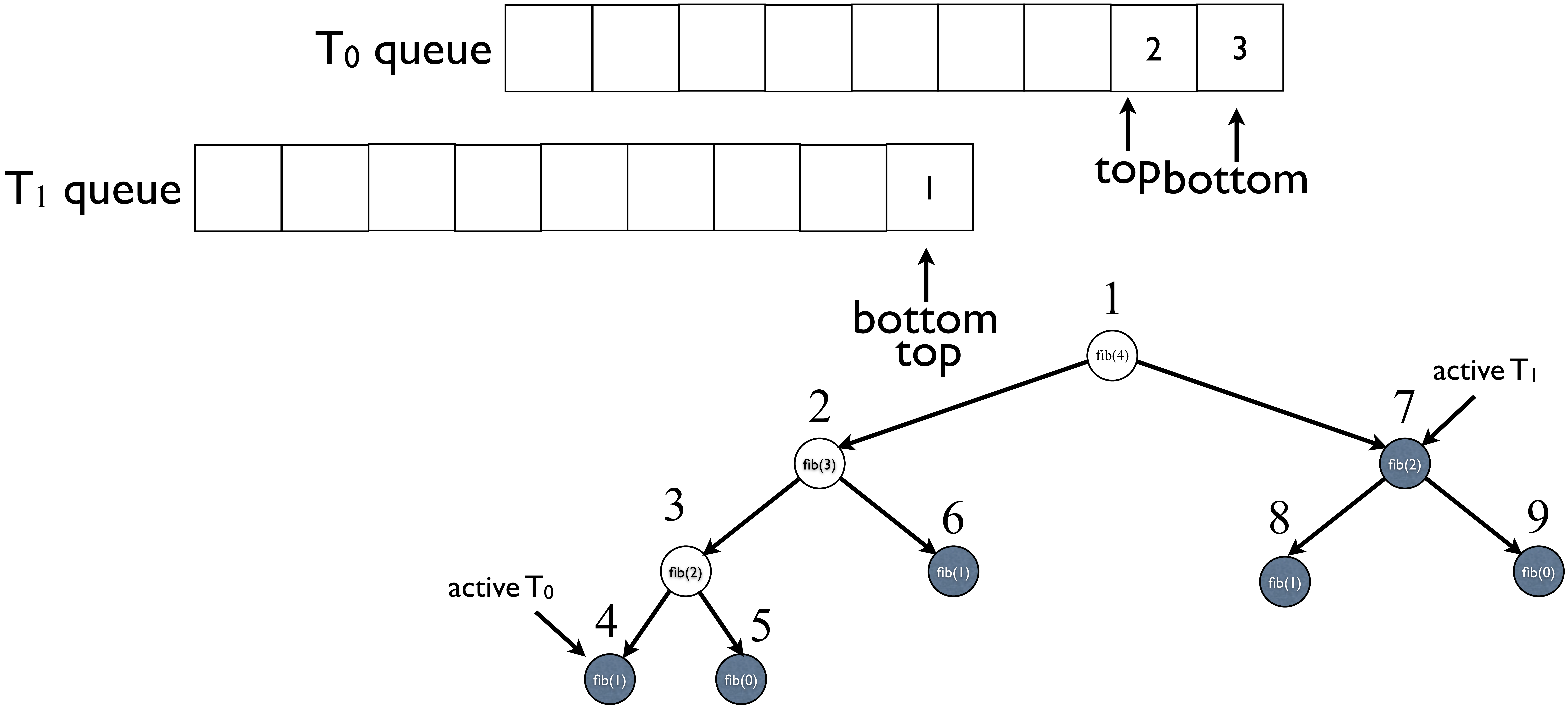
Work stealing example



Work stealing example



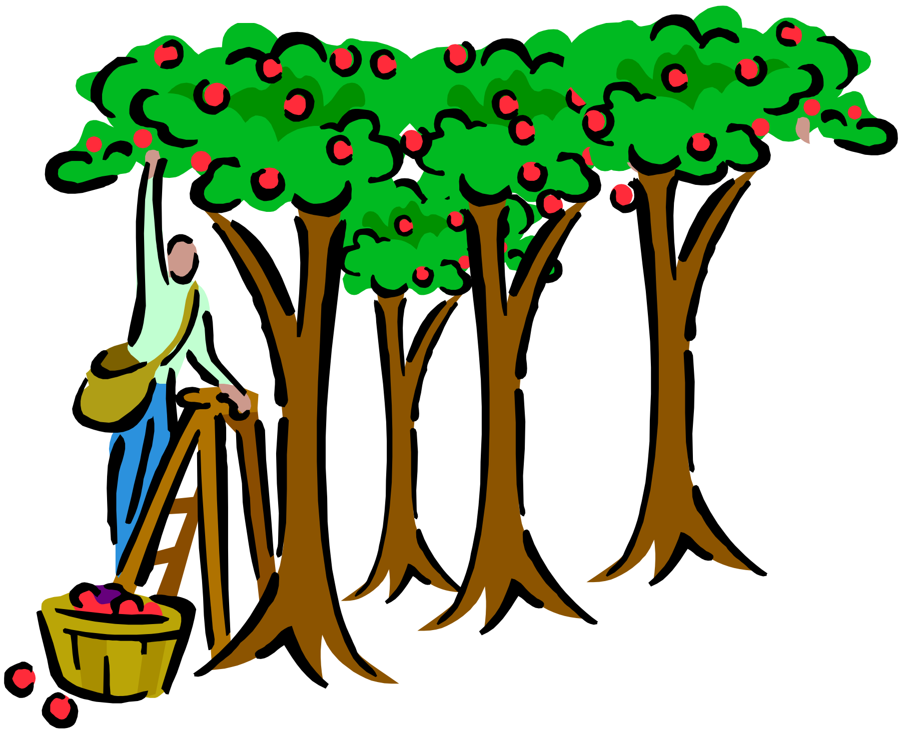
Work stealing example

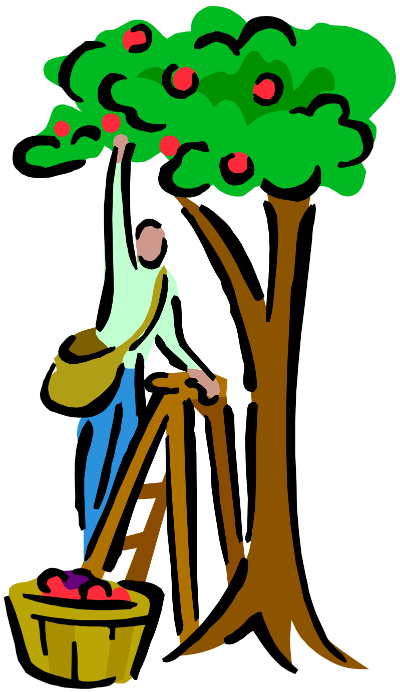


Parallel Programming with Cilk Plus

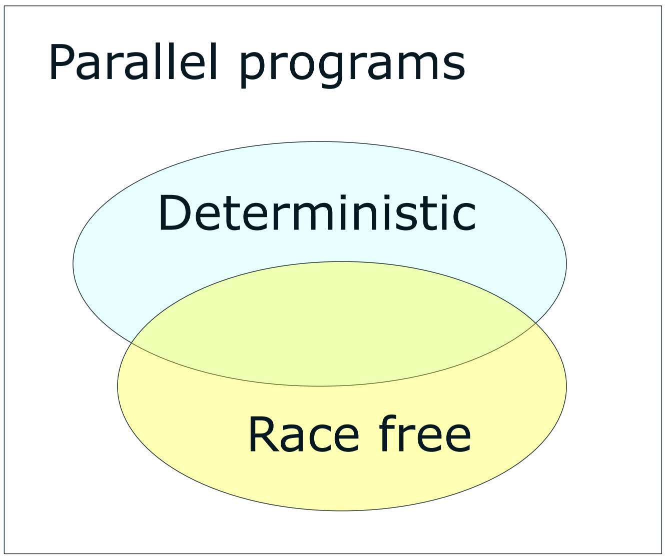
Arch D. Robison

Load Balancing and Locality





Race-Free ≠ Deterministic



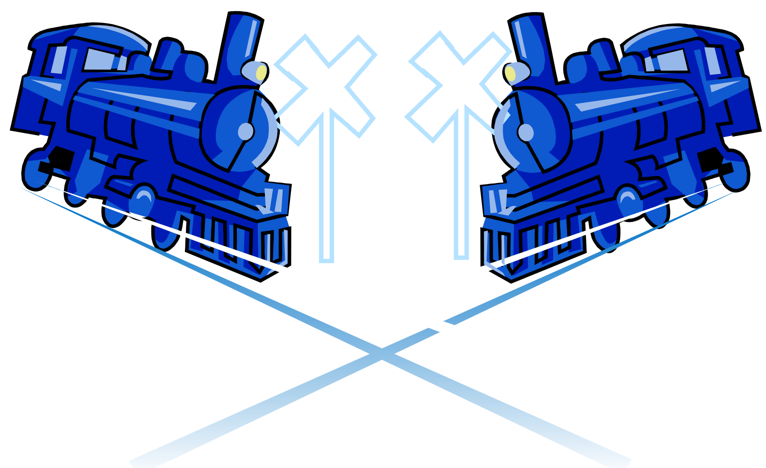
Thread 1	**Thread 2**
`x = 1;`	`x = 1;`

Thread 1	**Thread 2**
`m.lock();`	`m.lock();`
`x = 1;`	`x = 2;`
`m.unlock();`	`m.unlock();`

Deadlock

Thread 1

```
a.lock();
b.lock();
++A;
--B;
b.unlock();
a.unlock();
```



Thread 2

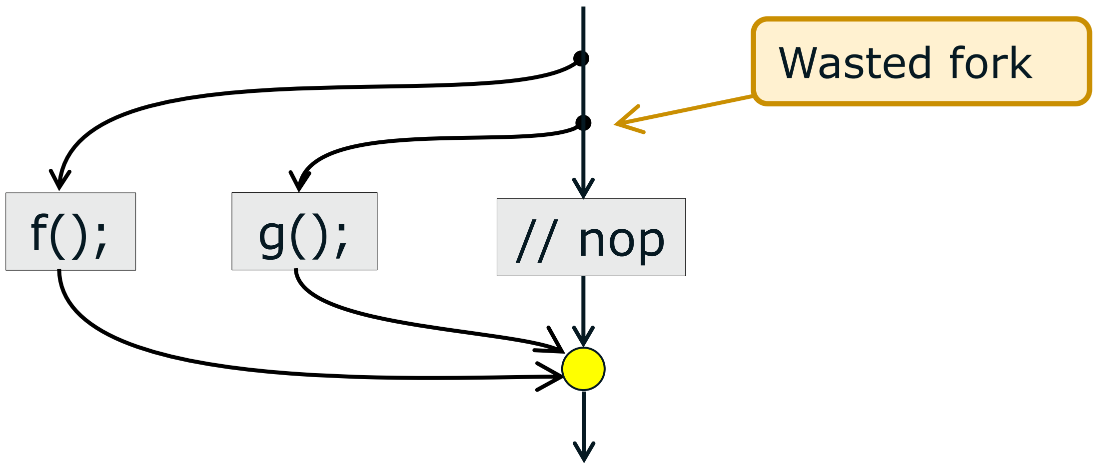
```
b.lock();
a.lock();
--B;
++A;
a.unlock();
b.unlock();
```

Philosophy of Cilk Plus

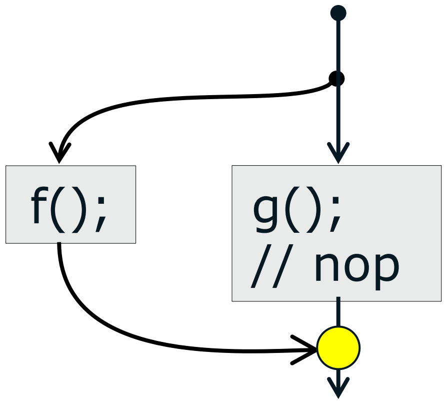
Division of Responsibilities	
Programmer	**Cilk Plus**
Specify what *can* run in parallel.	Make parallelism easy to express. Enable clean composition.
Provide much more *potential* parallelism than system can use.	Throttle *actual* parallelism. • Make unused parallelism cheap. • Balance load.
Express SIMD opportunities.	Make SIMD easy to express. Generate SIMD code.
Avoid races.	Synchronize strands of execution.
Minimize use of locks.	Provide hyperobjects.
Promote locality via cache-oblivious style.	Depth-first serial execution.

Style Issue

```
// Bad Style
cilk_spawn f();
cilk_spawn g();
// nop
cilk_sync;
```




```
// Preferred style
cilk_spawn f();
g();
// nop
cilk_sync;
```



Serial Elision

Cilk keywords can be trivially eliminated:

```
#define cilk_spawn
#define cilk_sync
#define cilk_for for
```

Resulting program is called the **serial elision**

- It is a valid serial C/C++ program!

Likewise, the serial elision is always a valid implementation of a Cilk program:

- Means a Cilk program can always run on a single thread.
- Fundamental requirement for avoiding oversubscription.

Races

Race

- Two unordered memory references and at least one is a write.

Cilk program is deterministic if:

- It has no races

Will talk about automatic race detection later.

- It uses no locks
- Reducer operations are associative

Floating-point + and * are almost associative.

Deterministic Cilk program has same effect as its serial elision.

29

Effective Cilk Plus: Writing Scalable Programs

Work-span model of complexity

Load balancing

Amortizing scheduling overhead

Hazards of locks

Hyperobjects revisited

Correctness tools survey

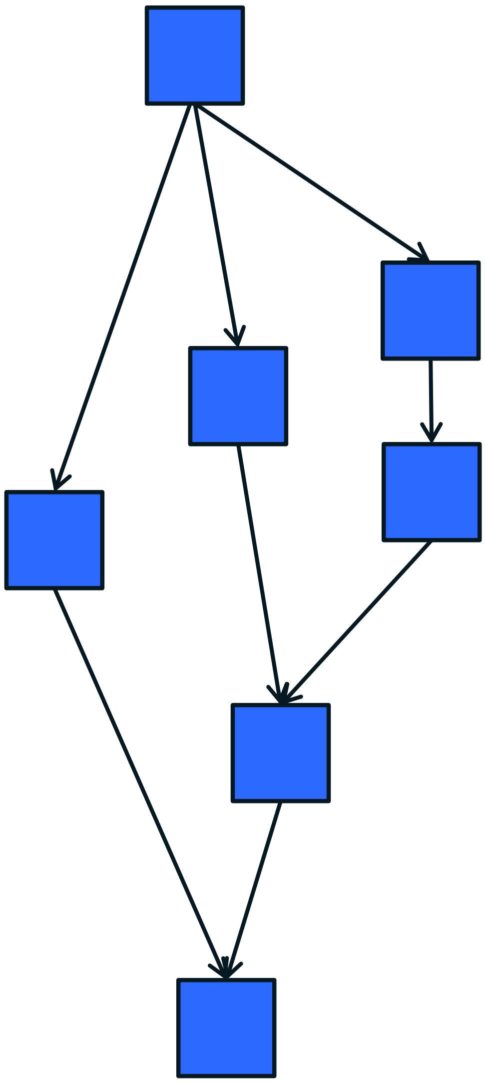
DAG Model of Computation

Program is a directed acyclic graph (DAG) of tasks

The hardware consists of workers

Scheduling is *greedy*

- No worker idles while there is a task available.



Work-Span Model

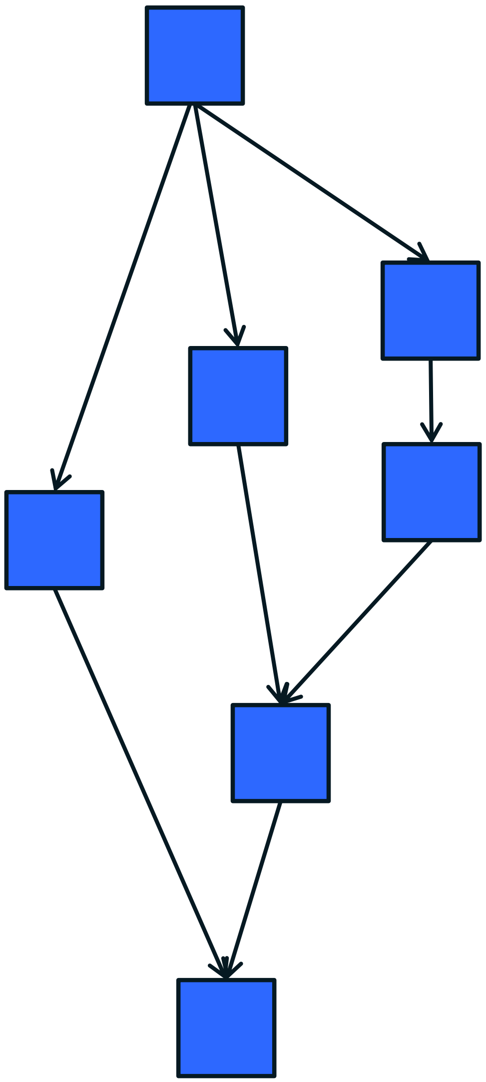
T_P = time to run with P workers

T_1 = *work*

- time for serial execution
- sum of all work

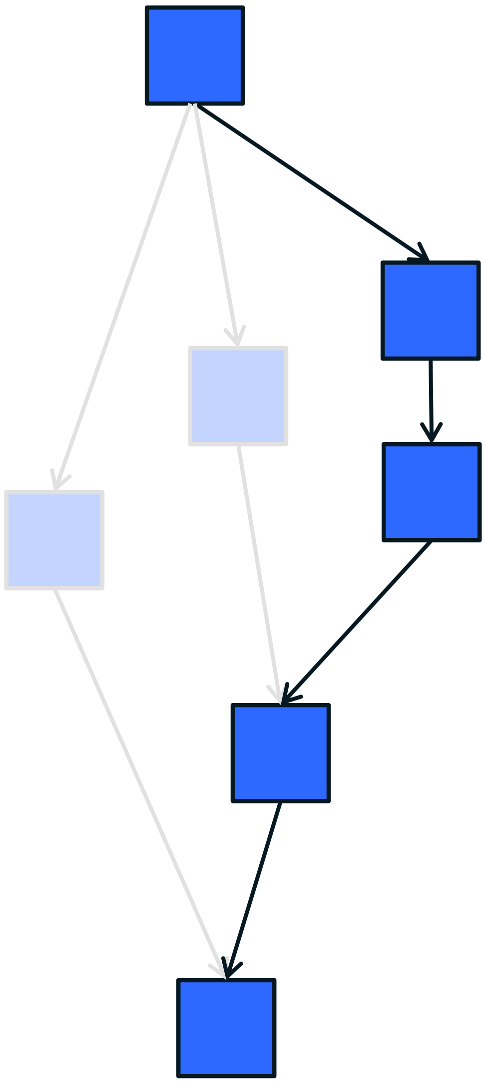
T_∞ = *span*

- time for *critical path*



Work-Span Example

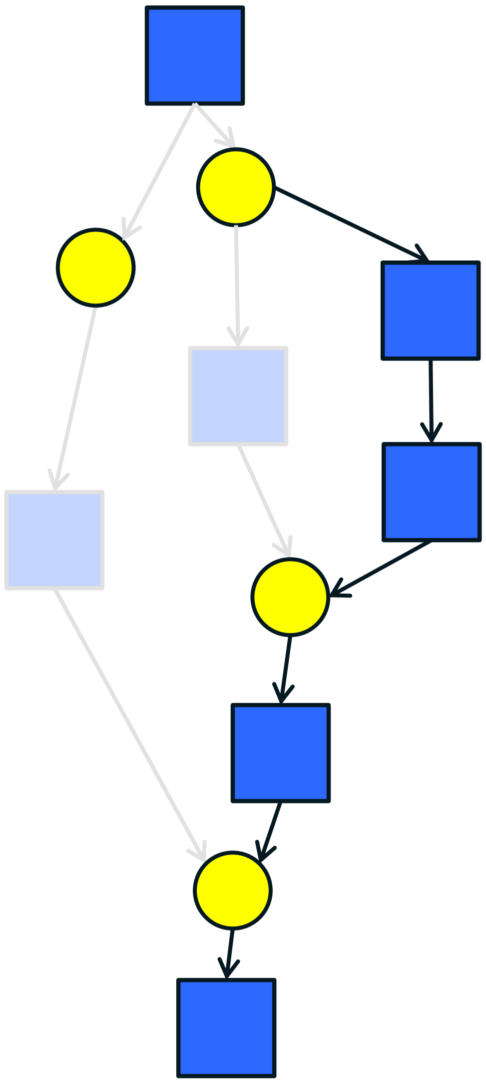
T_1 = *work* = 7
T_∞ = *span* = 5



Burdened Span

Includes extra cost for synchronization

Often dominated by cache line transfers.

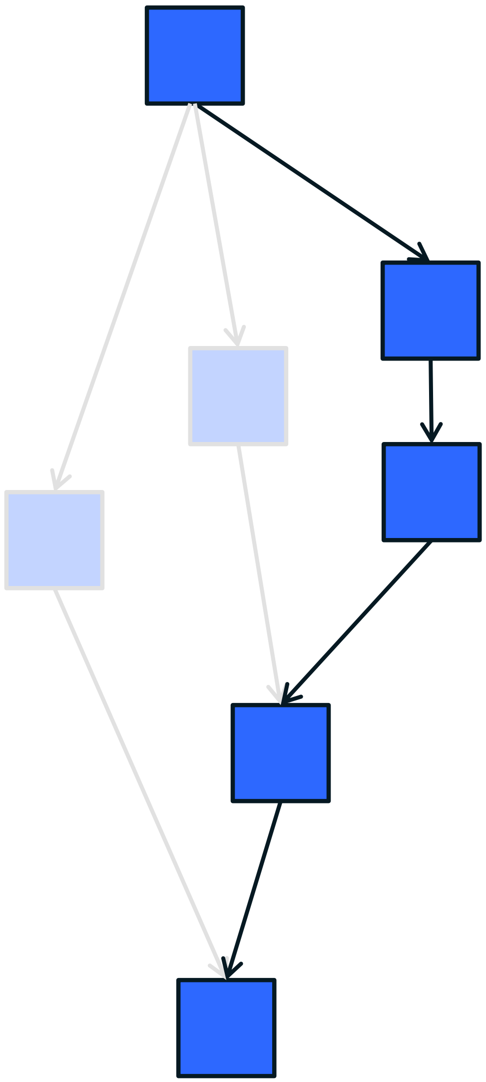


Lower Time Bound on Greedy Scheduling

(Implies upper bound on speedup)

Work-Span Limit

$$\max(T_1/P,\ T_\infty) \le T_P$$

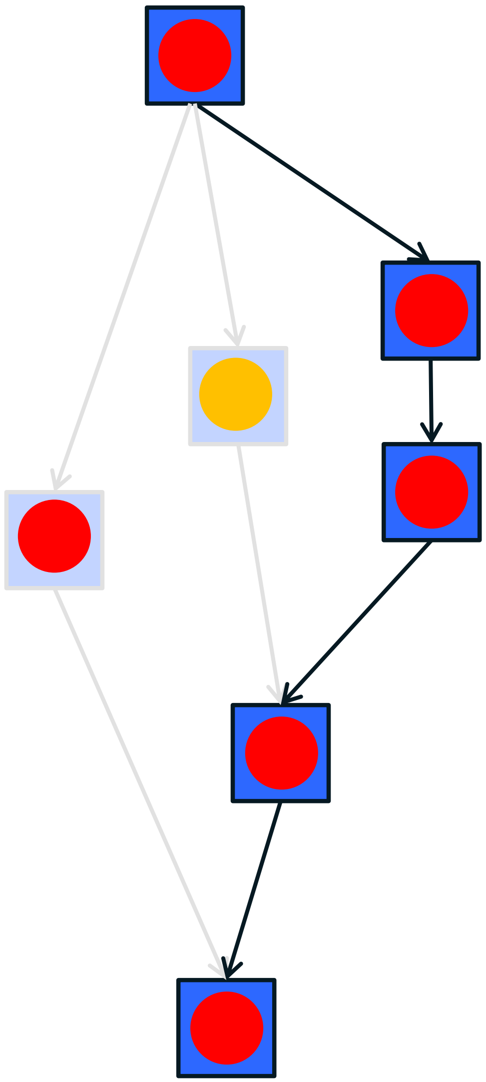


Upper Time Bound on Greedy Scheduling

(Implies *lower* bound on speedup)

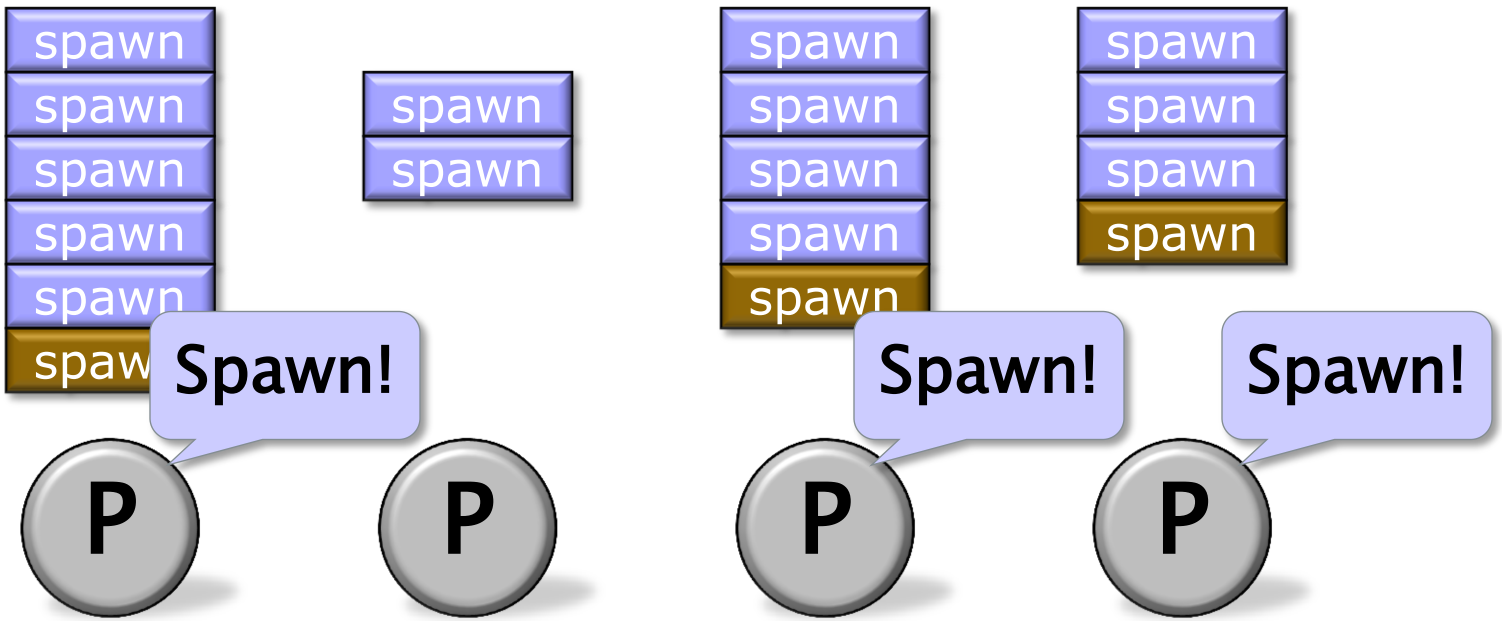
Brent's Lemma

$$T_P \leq (T_1 - T_\infty)/P + T_\infty$$



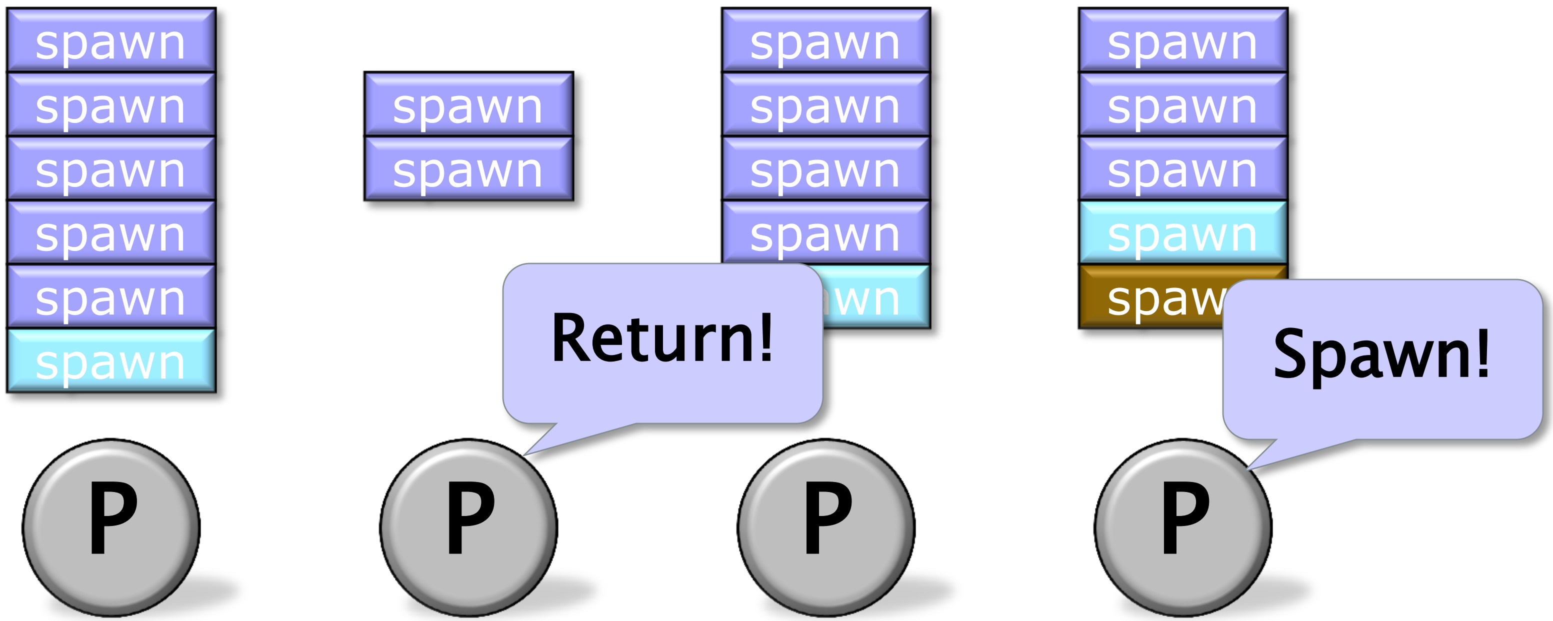
Load Balancing by Work-stealing

Each processor has a deque of spawned tasks.

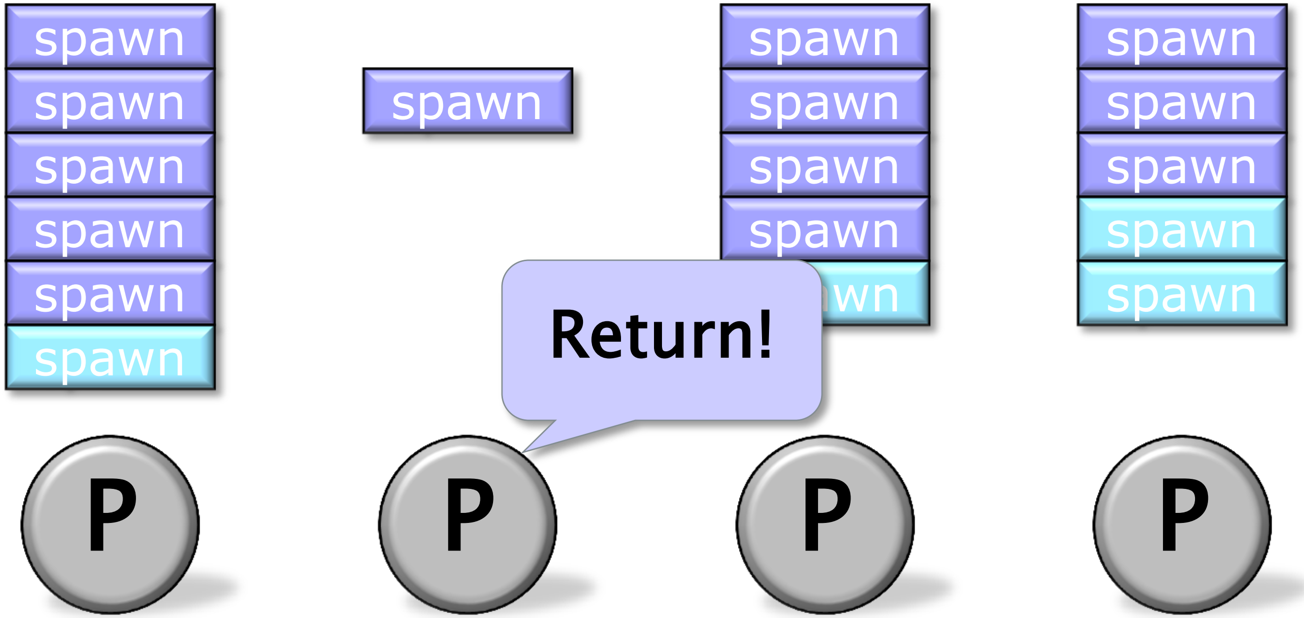


When each processor has work to do, a spawn is roughly the cost of about 25 function calls.

Load Balancing by Work-stealing

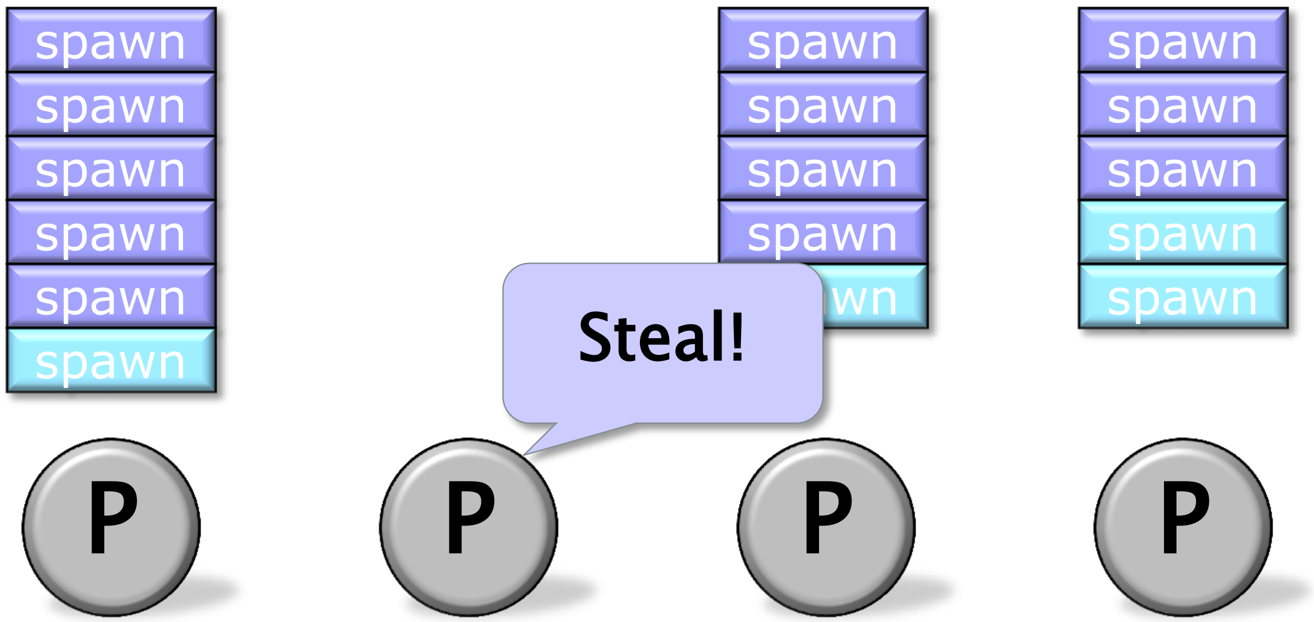


Load Balancing by Work-stealing



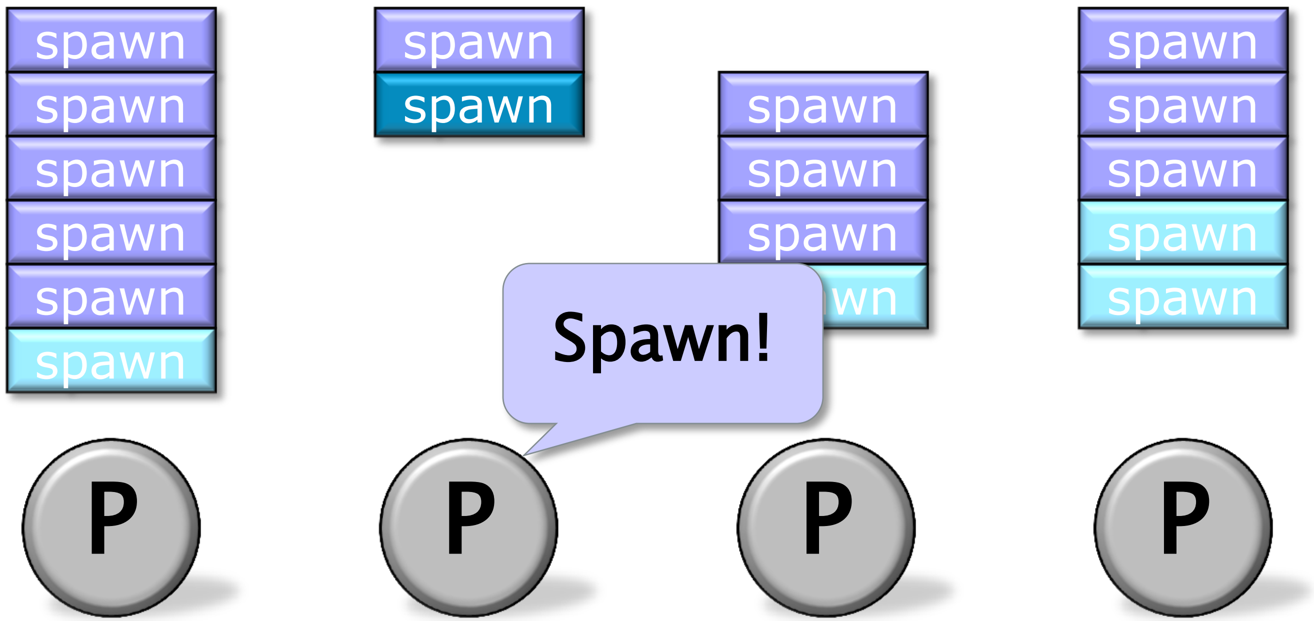
Optimization Notice

Work-stealing task scheduler



Optimization Notice

Work-stealing task scheduler



With sufficient parallelism, the steals are rare, and we get ***linear speedup*** (ignoring memory effects).

OpenMP Tactics to Unlearn
(Thanks to James Cownie for List)

1. Creating one work item per thread.
2. Anything involving omp_get_thread_num().
3. Fear of nested parallelism.

Problem with One Work Item Per Thread

Destroys composability

- No way to know if running as child or sibling of other parallel work.

Hurts load balancing.

- Gives scheduler no parallel slack.

Advice: Choose grain size based on amortizing scheduling overhead, not balancing load.

Problem with Using Thread Ids

Thus thread id can change in surprising ways.

- Id after spawn can be *different* than before spawn.
- Id after sync can be *different* than before spawns.

```
...
#include <cilk/cilk_api.h>

std::vector<int> A;

void bar() {
    int j = __cilkrts_get_worker_number()
    A[j]++;
}

int main() {
    A.resize (__cilkrts_get_nworkers()]);
    int i = __cilkrts_get_worker_number();
    cilk_spawn f();
    A[i]++;
    cilk_sync;
}
```

Race, because i==j!

Advice: Use hyperobjects (reducers and holders).

Embrace Nested Parallelism

Cilk was designed for nested parallelism.

Unused nested parallelism is inexpensive.

- Execution is serial when all threads are busy.

Performance Tools

Intel® Cilk™ View

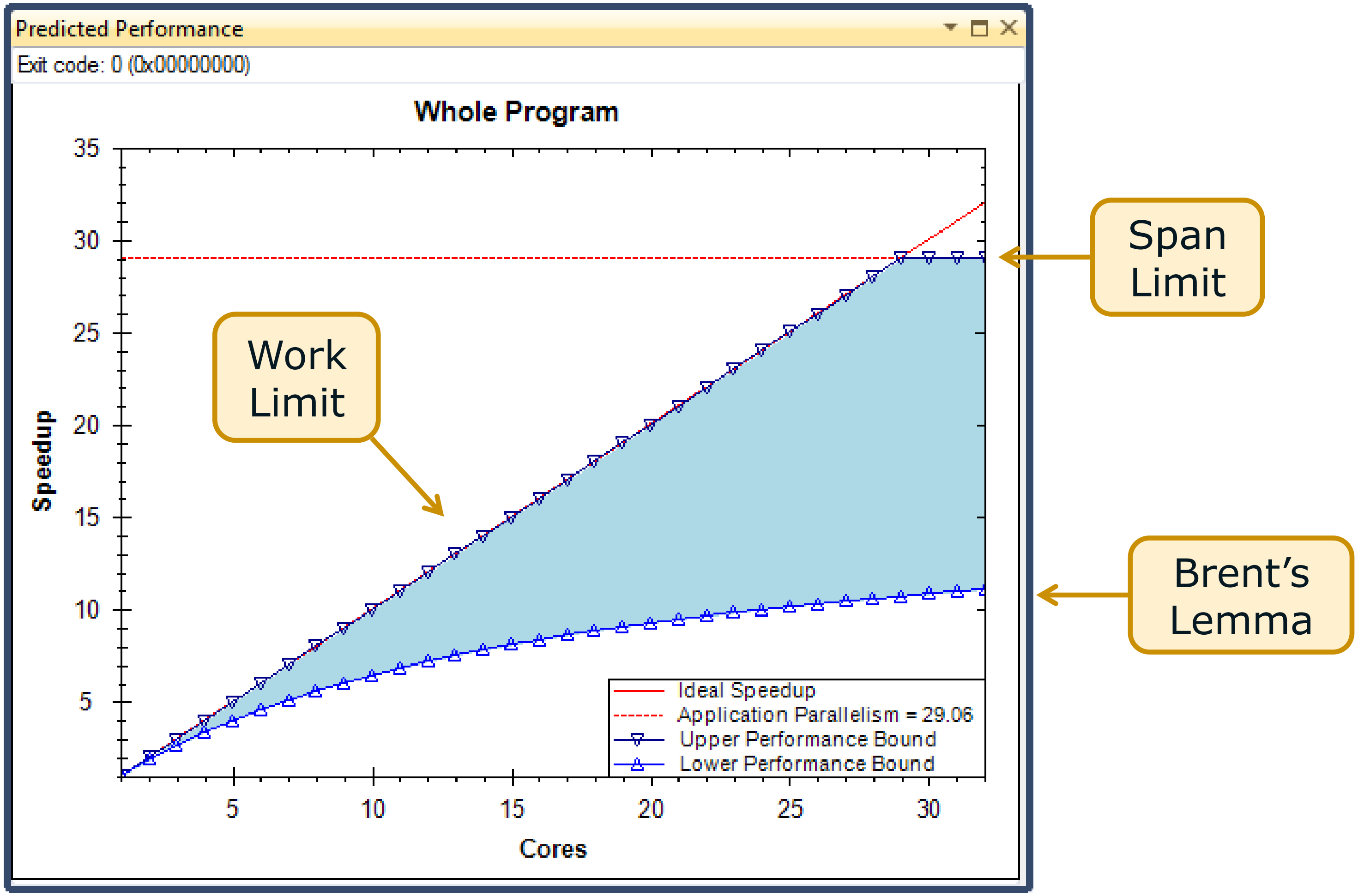
- Automatic work-span analysis for Cilk™ Plus

Intel® Amplifier

- General threading analysis
- Good for spotting hardware-related bottlenecks

Sample Cilk View Output

Uses *burdened span* that estimates scheduling costs.



Two Race Detectors for Cilk Plus

Intel® Cilk Screen

- "Happens before" on strands + "Lock set"
- Theoretically efficient implementation that strict fork-join nature of Cilk

Intel® Parallel Inspector

- "Happens before" on threads + "Lock set"
- Also detects potential deadlock
- Also has memory checker
- GUI integrates into Visual Studio

Both based on "Pin" dynamic instrumentation technology.
http://www.pintool.org/

Cilk Screen Example

```
void f() {
    int x[10];
    cilk_for( int i=0; i<10; ++i )
        x[i] = pseudo_random();
}
```

```
5 int pseudo_random() {
6     static int state = 1;
7     return state = a*state+b;
8 }
```

```
$ icc -g randomfill.cpp
$ cilkscreen a.out
Cilkscreen Race Detector V2.0.0, Build 2516

Race condition on location 0x600b84
  write access at 0x40062b: (/tmp/randomfill.cpp:7, pseudo_random+0x19)
  read access at 0x40061a: (/tmp/randomfill.cpp:7, pseudo_random+0x8)
    called by 0x2b2156f08b07: (__$U0+0xc7)
    called by 0x2b2156f08848: (cilk_for_recursive<unsigned int, void (*)(void*, unsigned int,
unsigned int)>+0x128)
    called by 0x2b2156f086b8: (__$U1+0xb8)
    called by 0x2b2156f082c5: (cilk_for_root<unsigned int, void (*)(void*, unsigned int,
unsigned int)>+0x135)
    called by 0x2b2156f0818a: (__cilkrts_cilk_for_32+0xa)
```

Philosophy of Cilk Plus

Division of Responsibilities	
Programmer	**Cilk Plus**
Specify what *can* run in parallel.	Make parallelism easy to express. Enable clean composition.
Provide much more *potential* parallelism than system can use.	Throttle *actual* parallelism. • Make unused parallelism cheap. • Balance load.
Express SIMD opportunities.	Make SIMD easy to express. Generate SIMD code.
Avoid races.	Synchronize strands of execution.
Minimize use of locks.	Provide hyperobjects.
Promote locality via cache-oblivious style.	Depth-first serial execution.

URLs

Cilk Plus home page

- http://cilkplus.org

Cilk Plus Forum

- http://software.intel.com/en-us/forums/intel-cilk-plus/

Cilk Plus Specifications

- http://software.intel.com/en-us/articles/intel-cilk-plus-specification/

Intel® Cilk™ Plus Software Development Kit

- http://software.intel.com/en-us/articles/intel-cilk-plus-software-development-kit/
  - Cilk Screen Race Detector
  - Cilk View Scalability Analyzer

GCC 4.7 Branch

- http://gcc.gnu.org/svn/gcc/branches/cilkplus/

Intel ® Parallel Inspector

- http://software.intel.com/en-us/articles/intel-parallel-inspector/

Legal Disclaimer & Optimization Notice

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

intel®

Where to get Cilk

- Cilk Arts was Charles Leiserson's company to commercialize Cilk
- Acquired by Intel in 2009
- In September 2010 released by Intel as Intel Cilk Plus
  - adds support for reductions
  - simplifies the language
  - debugger integration
- Spec published, and Intel is encouraging other vendors to support the language