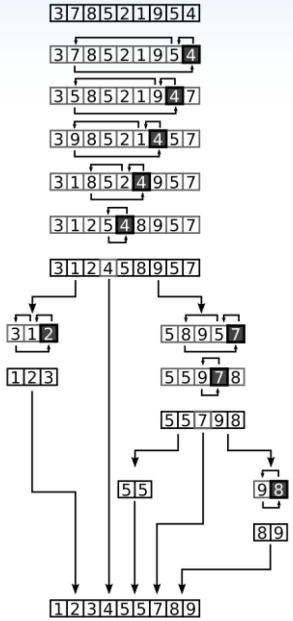
6.506: Algorithm Engineering



Lecture 4
The Cilk Runtime System

Alexandros-Stavros Iliopoulos

February 16, 2023

Cilk Programming

Cilk allows programmers to make software run faster using parallel processors.

Serial fib

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

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

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

Running time T_S.

Parallelized fib using Cilk

```
int fib(int n) {
  if (n < 2) {
    return n;
  } else {
    int x, y;
    cilk_scope {
        x = cilk_spawn fib(n-1);
        y = fib(n-2);
    }
    return (x + y);
  }
}
```

Running time T_P on P processors.

Scheduling in Cilk

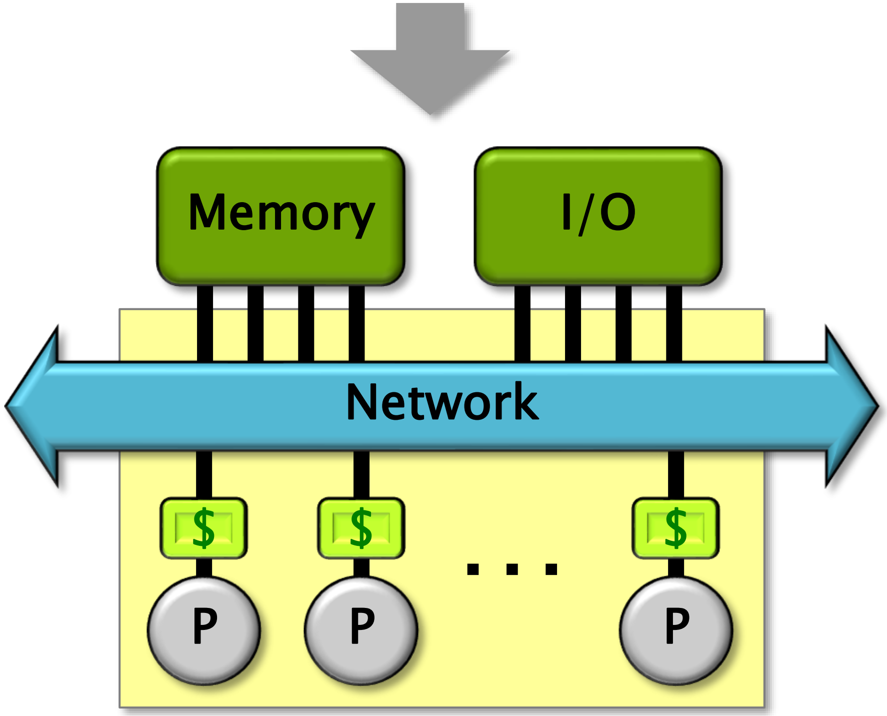
- The Cilk concurrency platform allows the programmer to express logical parallelism in an application.

```
int fib(int n) {
  if (n < 2) return n;
  int x, y;
  cilk_scope {
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
  }
  return (x + y);
}
```

Scheduling in Cilk

- The Cilk concurrency platform allows the programmer to express logical parallelism in an application.
- The Cilk scheduler maps the executing program onto the processor cores dynamically at runtime.

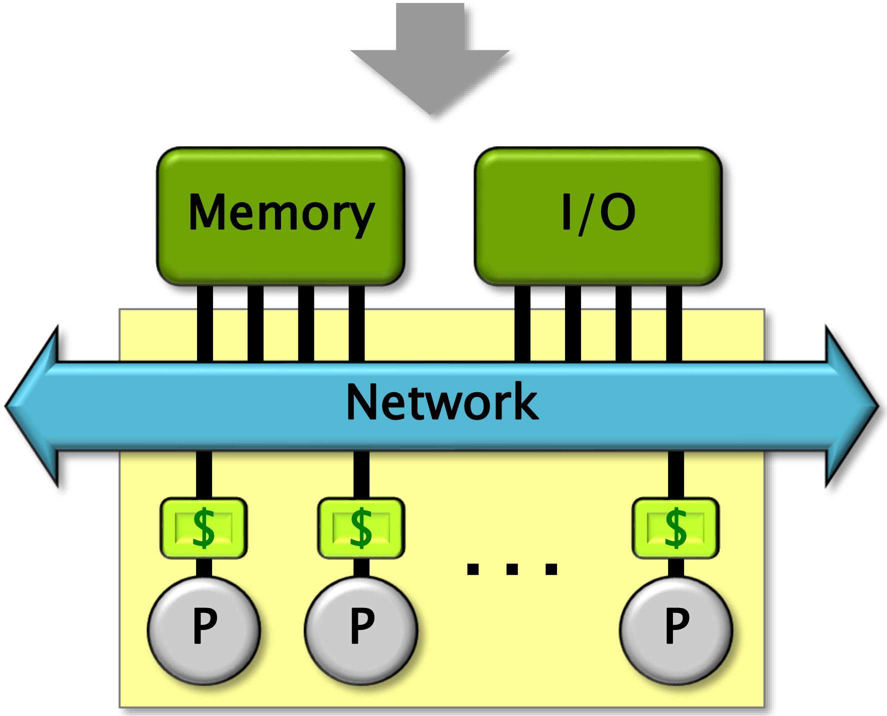
```
int fib(int n) {
  if (n < 2) return n;
  int x, y;
  cilk_scope {
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
  }
  return (x + y);
}
```



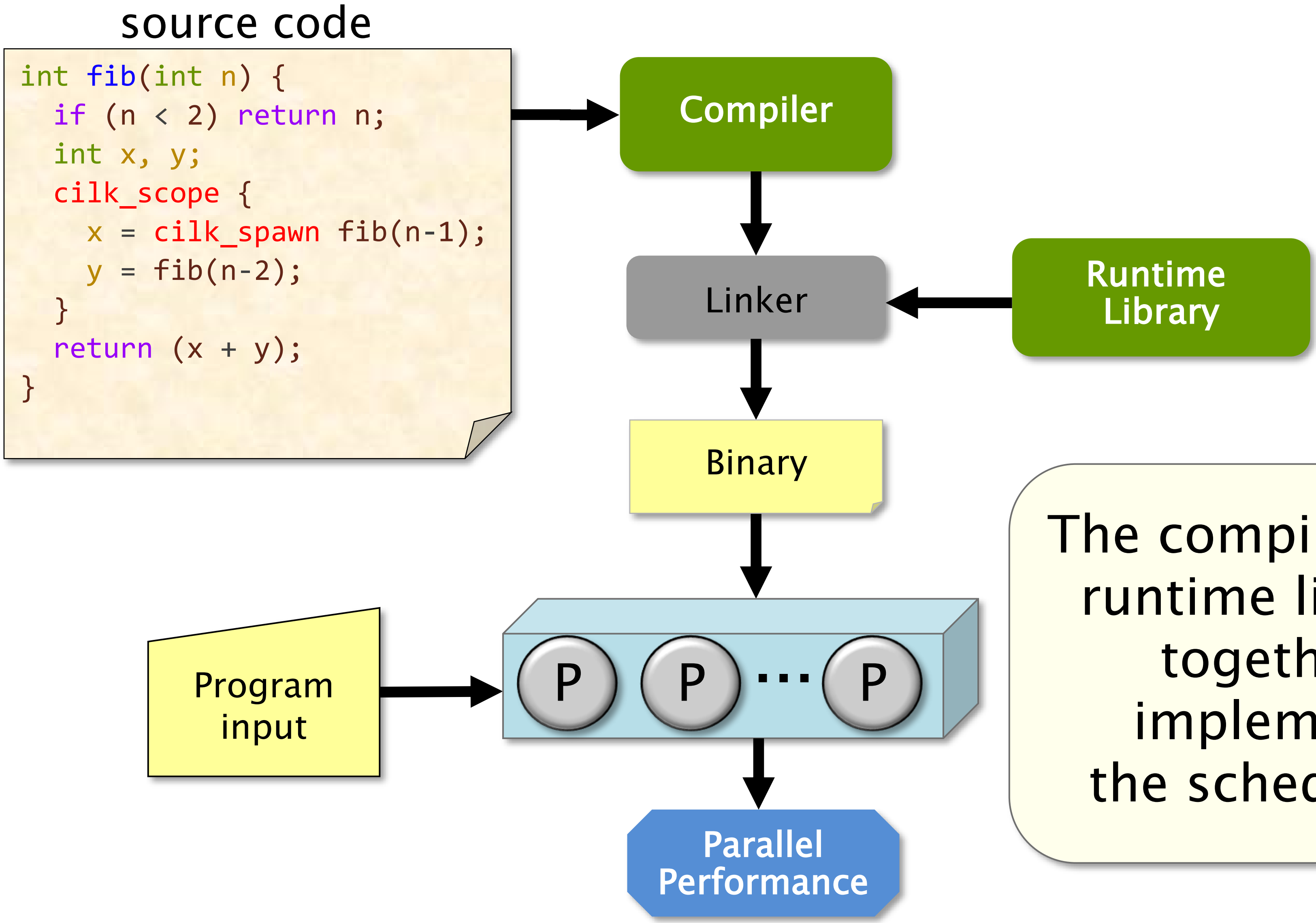
Scheduling in Cilk

- The Cilk concurrency platform allows the programmer to express logical parallelism in an application.
- The Cilk scheduler maps the executing program onto the processor cores dynamically at runtime.
- Cilk's *work-stealing scheduler* is provably efficient.

```
int fib(int n) {
  if (n < 2) return n;
  int x, y;
  cilk_scope {
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
  }
  return (x + y);
}
```

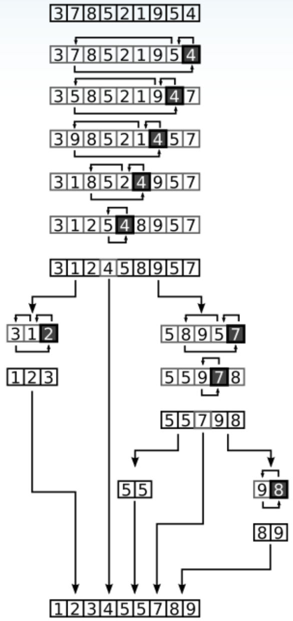


Cilk Platform



```
int fib(int n) {
  if (n < 2) return n;
  int x, y;
  cilk_scope {
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
  }
  return (x + y);
}
```

The compiler and runtime library together implement the scheduler.



Work Stealing and the Work-First Principle

Serial Execution & Stack Frames

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

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

  return (x + y);
}
```

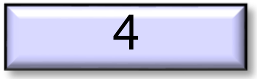
Example:

`fib(4)`

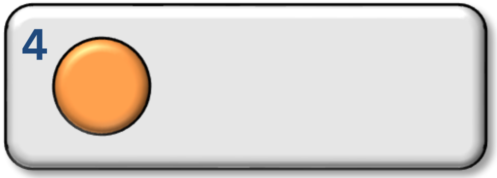
Serial Execution & Stack Frames

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

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

  return (x + y);
}
```

Call stack



Execution trace



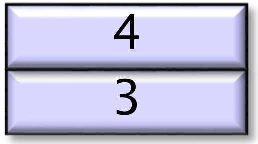
Example:

fib(4)

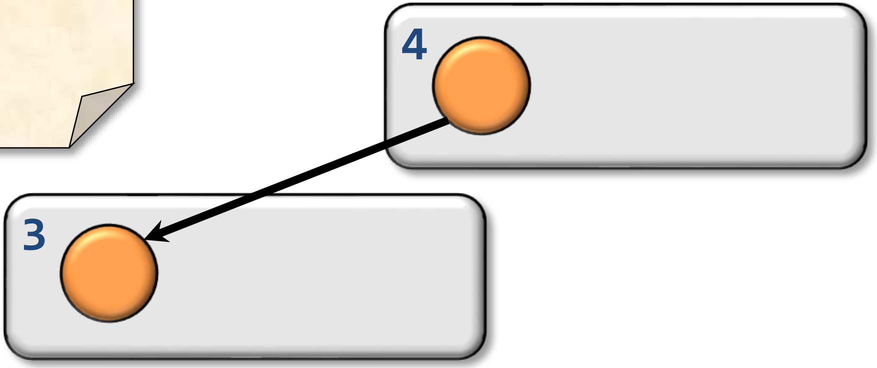
Serial Execution & Stack Frames

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

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

  return (x + y);
}
```

Call stack



Execution trace



Example:

`fib(4)`

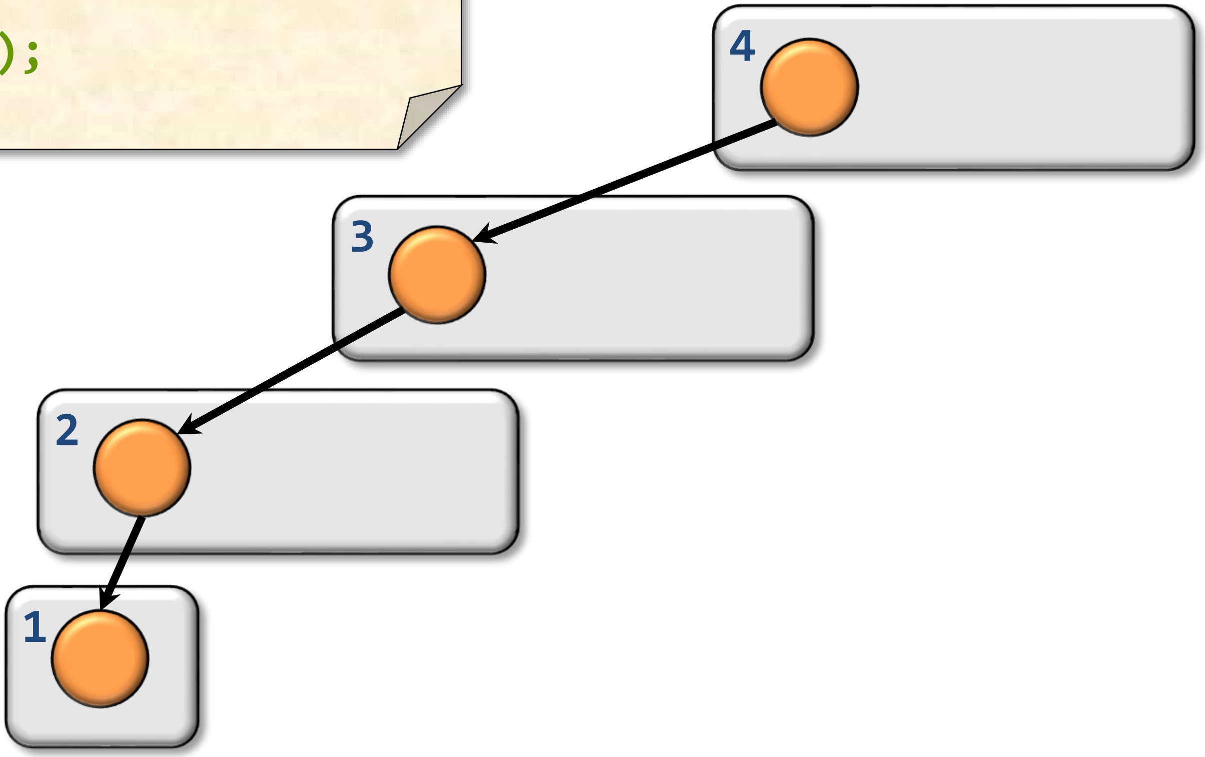
Serial Execution & Stack Frames

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

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

  return (x + y);
}
```

Call stack

4
3
2
1

Execution trace



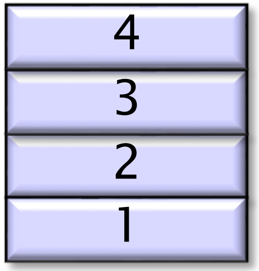
Example:

`fib(4)`

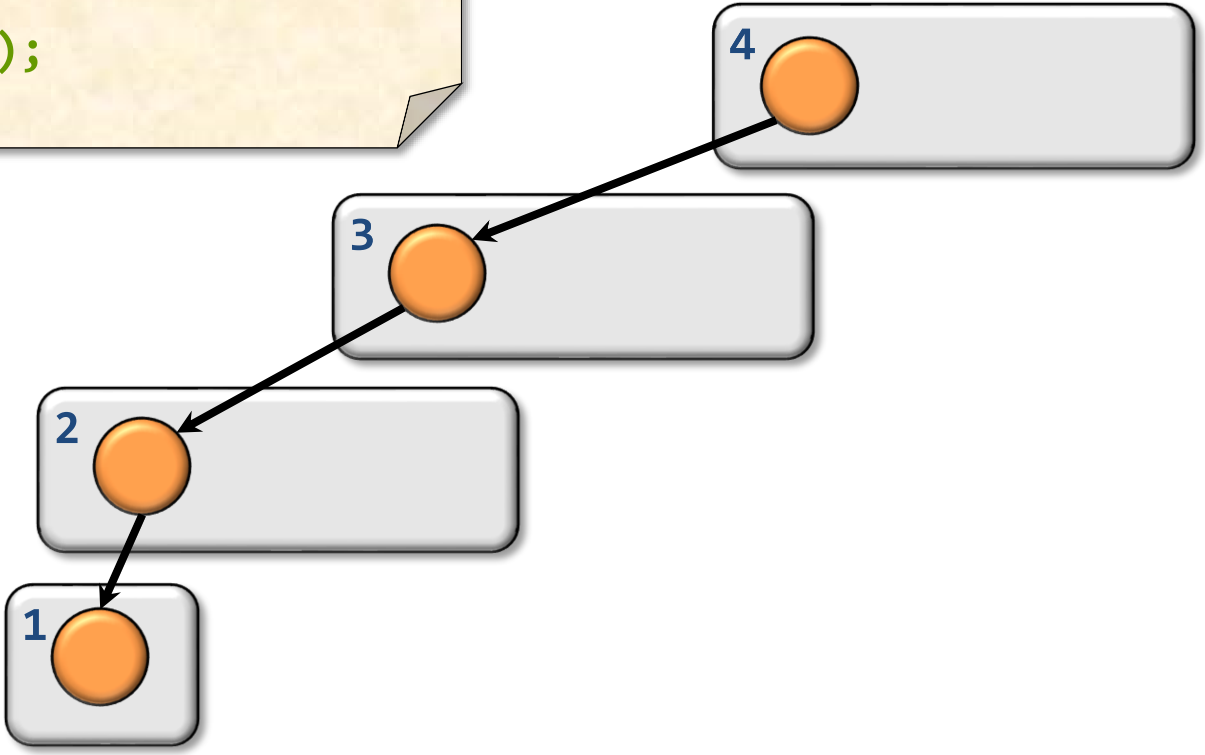
Serial Execution & Stack Frames

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

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

  return (x + y);
}
```

Call stack



Execution trace



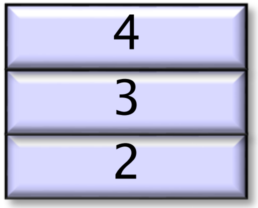
Example:

`fib(4)`

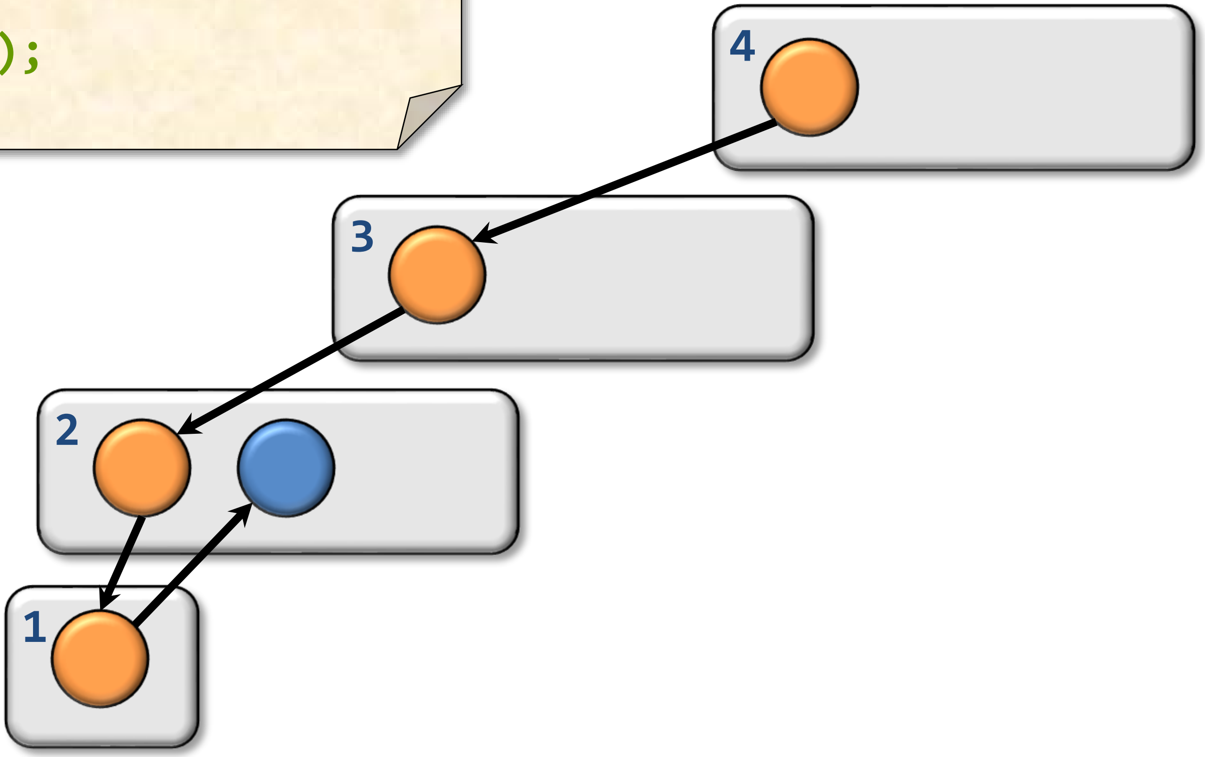
Serial Execution & Stack Frames

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

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

  return (x + y);
}
```

Call stack



Execution trace



Example:

`fib(4)`

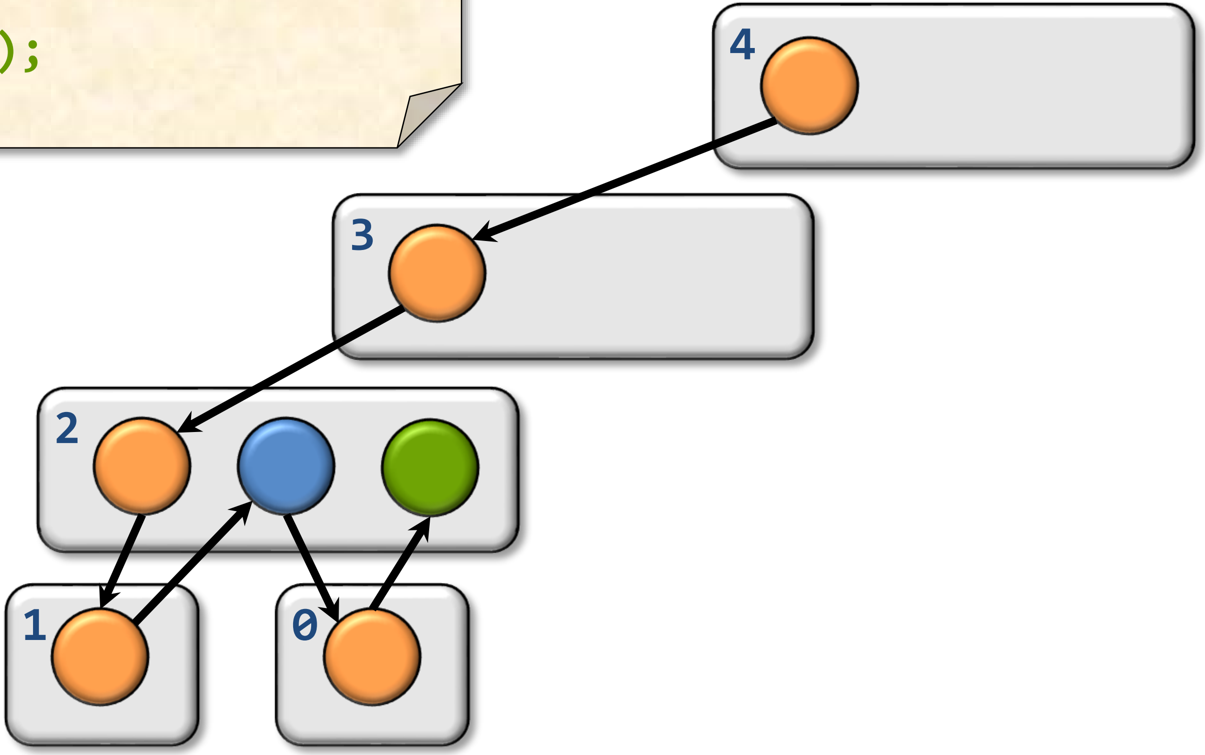
Serial Execution & Stack Frames

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

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

  return (x + y);
}
```

Call stack

4
3
2

Execution trace



Example:

`fib(4)`

Serial Execution & Stack Frames

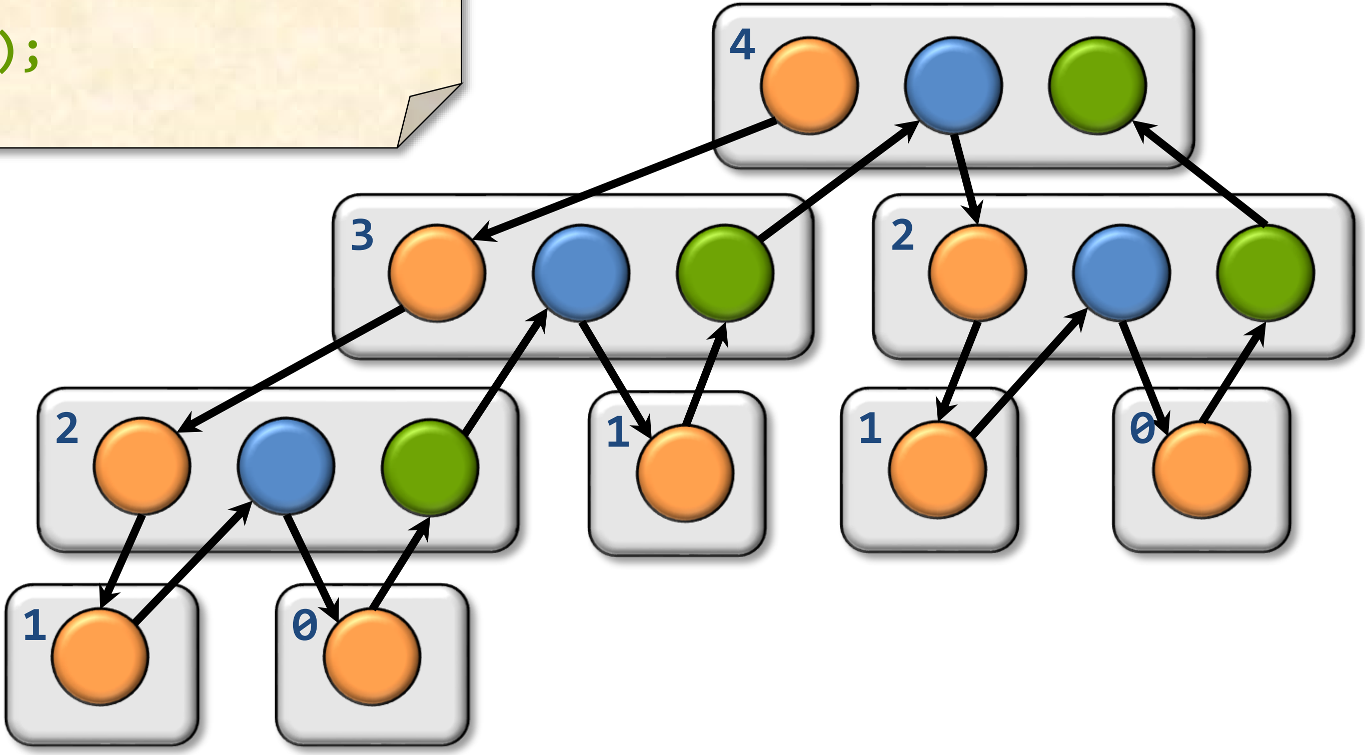
```
int fib(int n) {
  if (n < 2) return n;
  int x, y;

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

  return (x + y);
}
```

Example:

`fib(4)`

Call stack

Execution trace



Serial Execution & Stack Frames

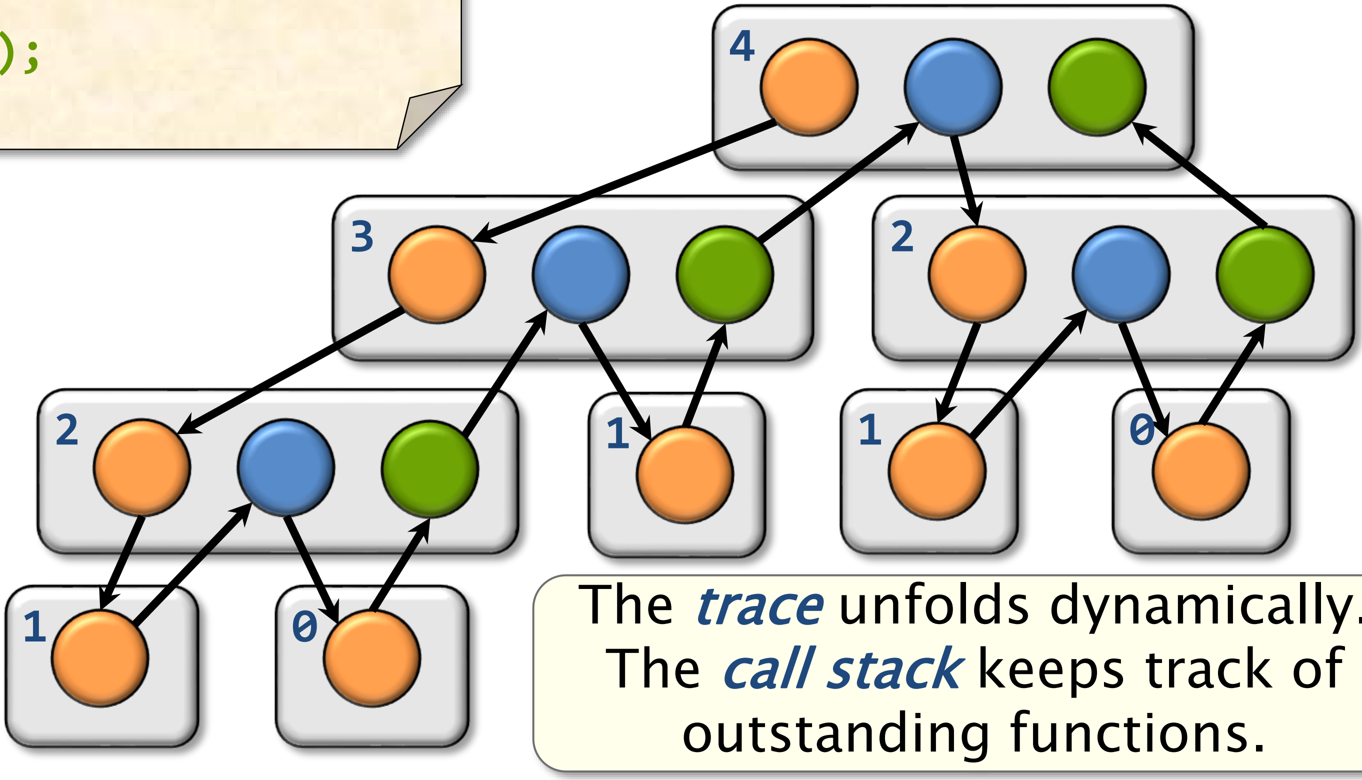
```
int fib(int n) {
  if (n < 2) return n;
  int x, y;

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

  return (x + y);
}
```

Call stack

Example:

`fib(4)`

Execution trace

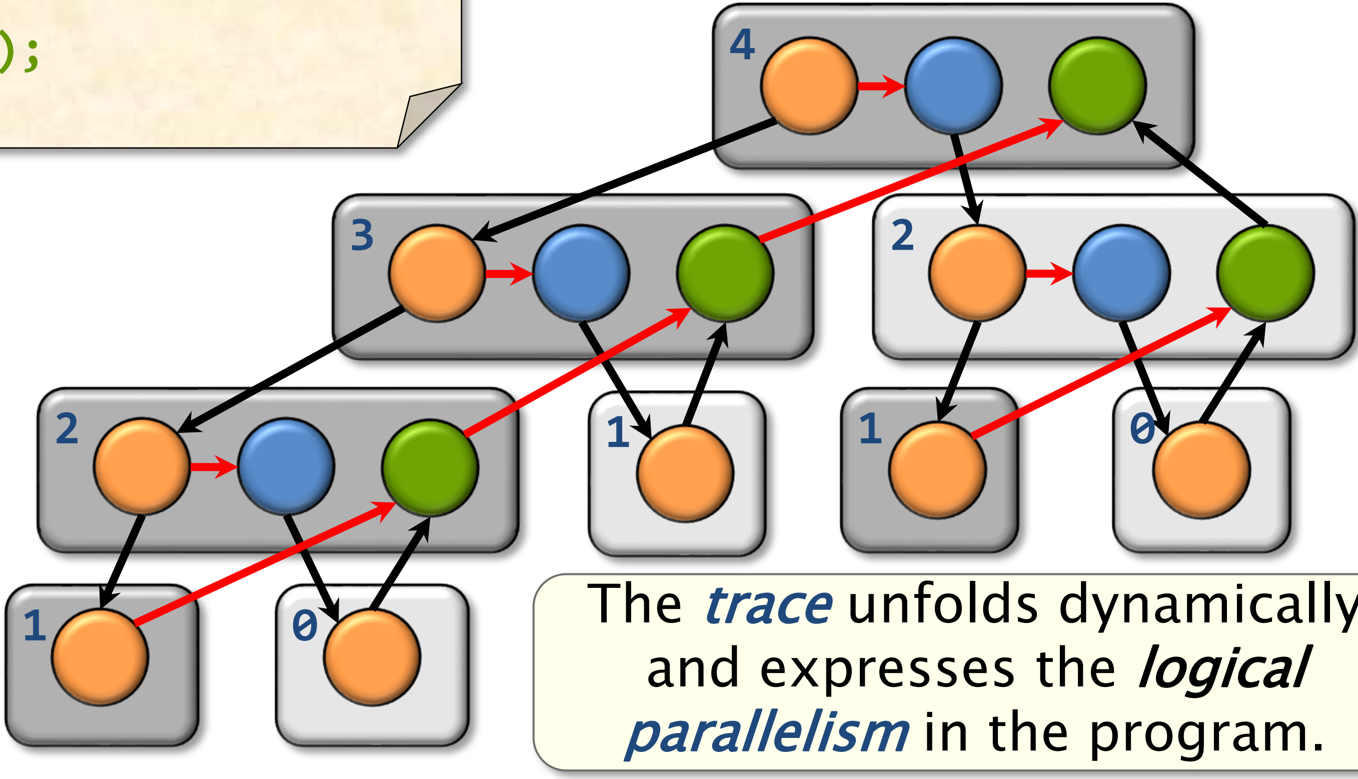


The *trace* unfolds dynamically. The *call stack* keeps track of outstanding functions.

Parallel Execution

```
int fib(int n) {
  if (n < 2) return n;
  int x, y;
  cilk_scope {
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
  }
  return (x + y);
}
```

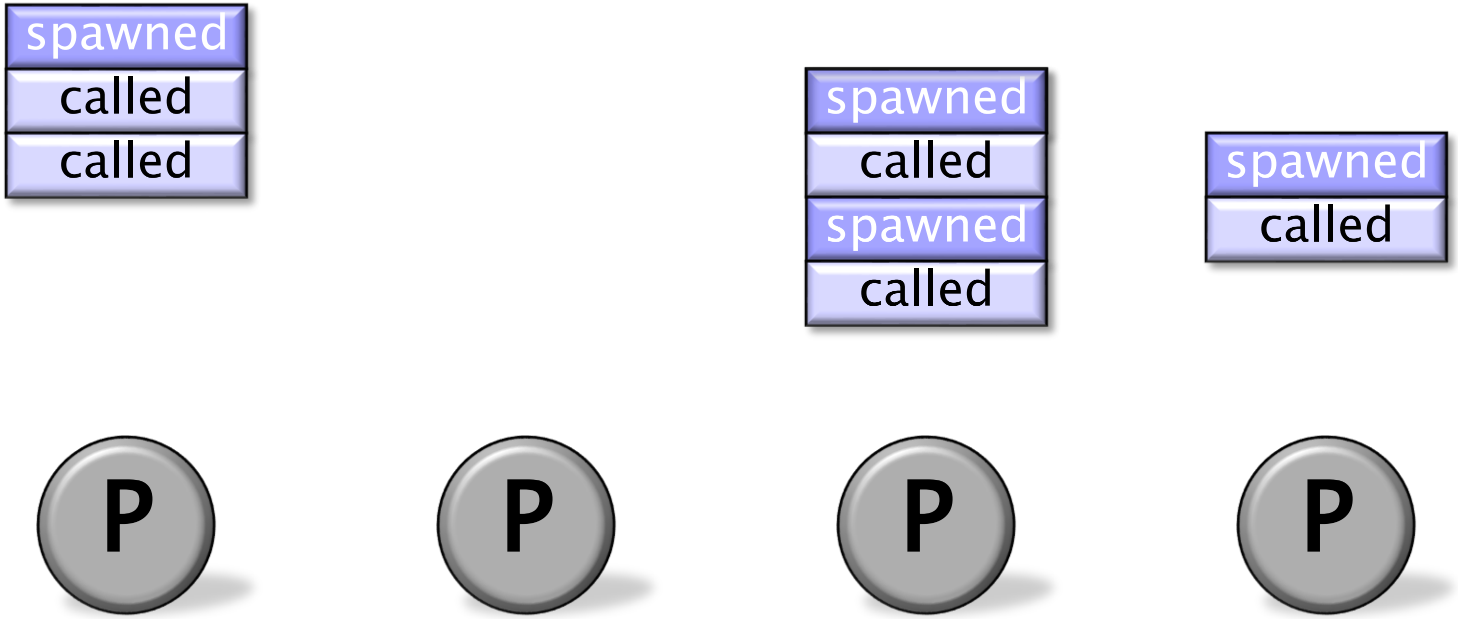
Example:
fib(4)



The *trace* unfolds dynamically and expresses the *logical parallelism* in the program.

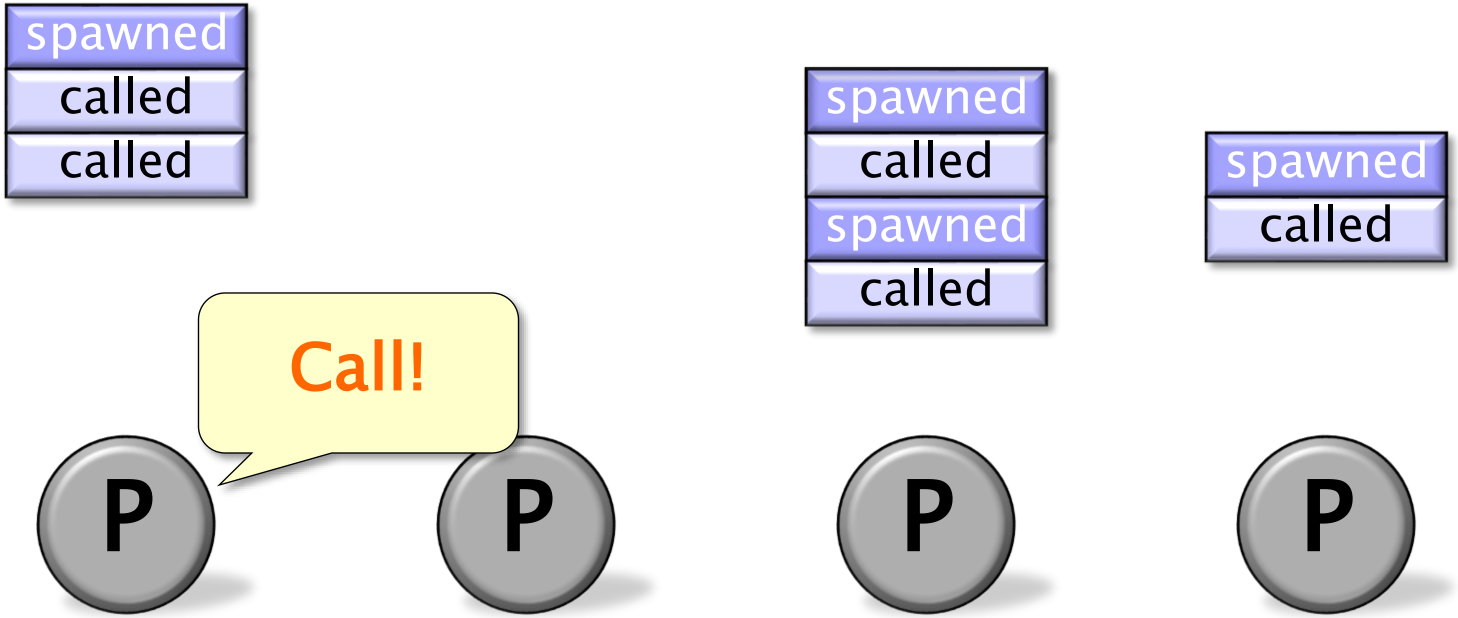
Work Stealing

Each worker (processor) maintains a ***work deque*** of ready strands, and it manipulates the bottom of the deque like a call stack.



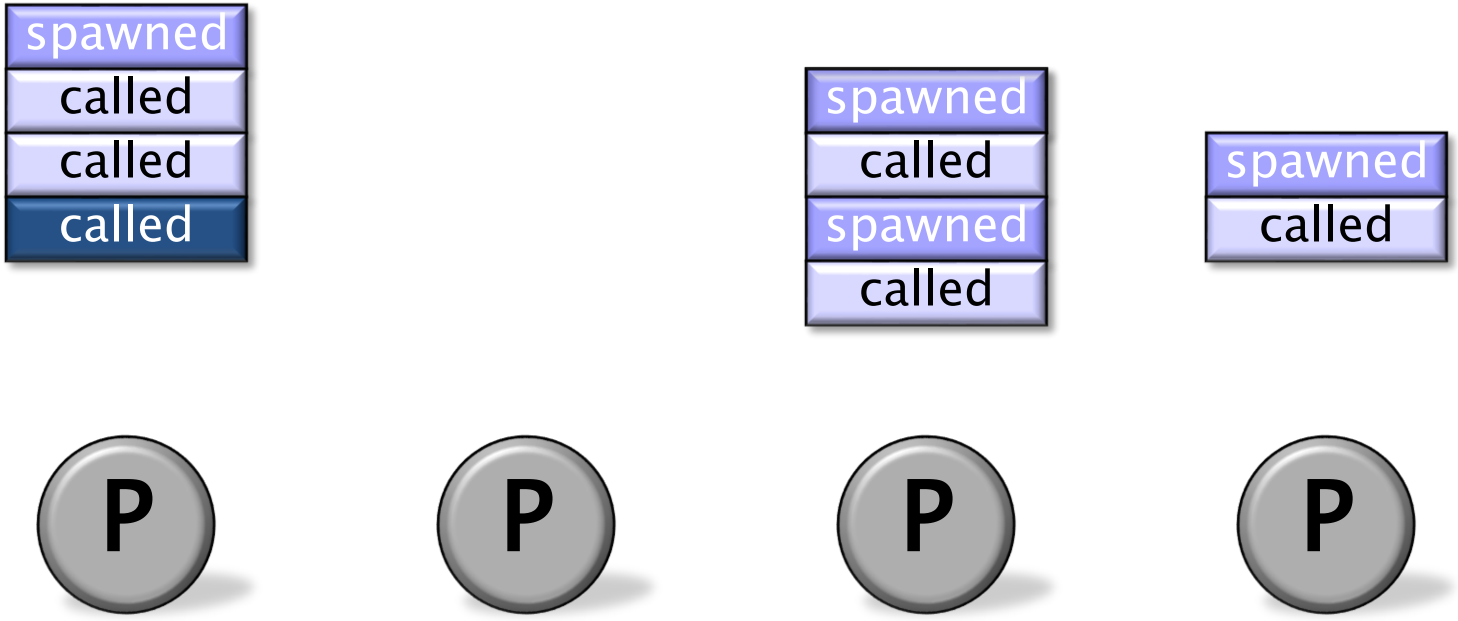
Work Stealing

Each worker (processor) maintains a ***work deque*** of ready strands, and it manipulates the bottom of the deque like a call stack.



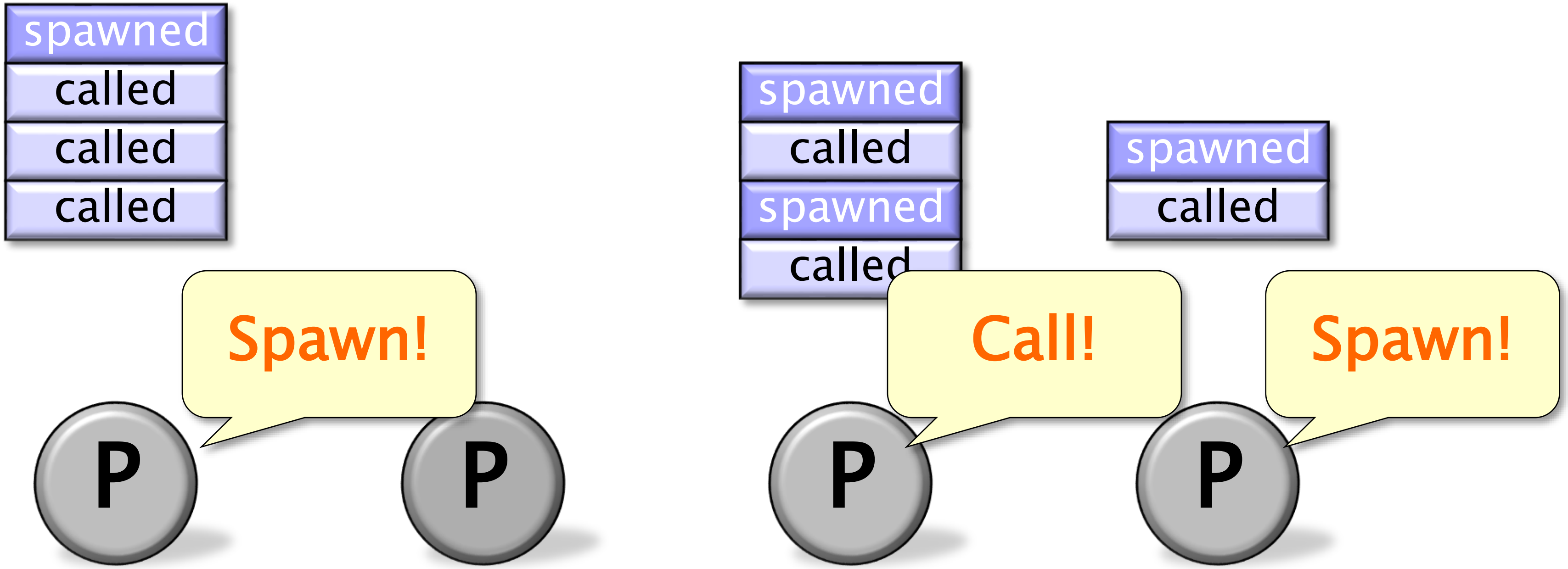
Work Stealing

Each worker (processor) maintains a ***work deque*** of ready strands, and it manipulates the bottom of the deque like a call stack.



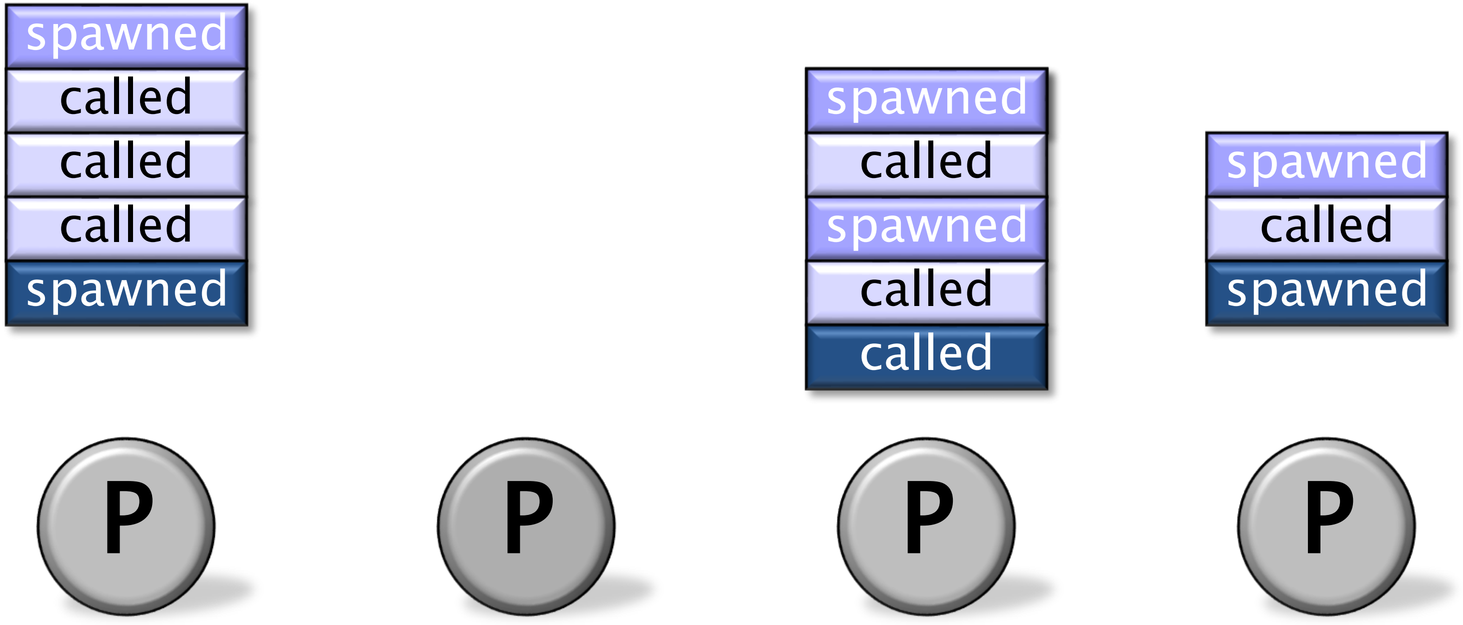
Work Stealing

Each worker (processor) maintains a ***work deque*** of ready strands, and it manipulates the bottom of the deque like a stack.



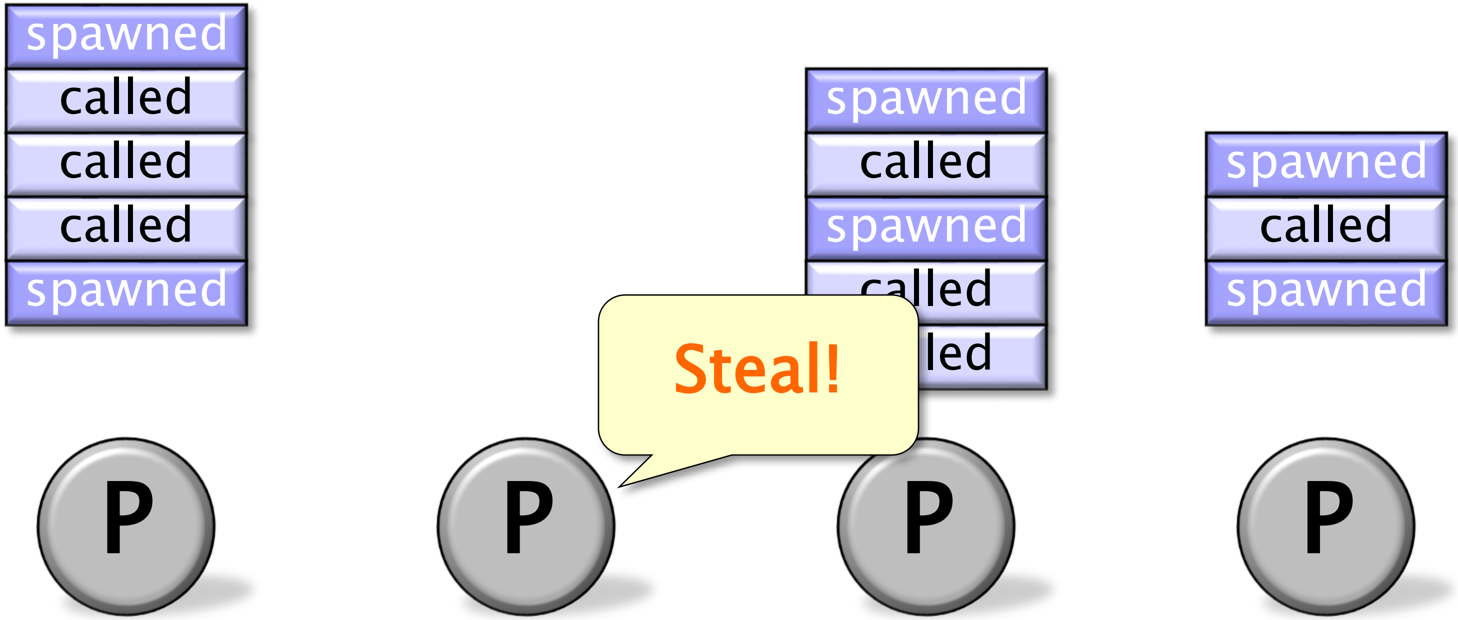
Work Stealing

Each worker (processor) maintains a *work deque* of ready strands, and it manipulates the bottom of the deque like a stack.



Work Stealing

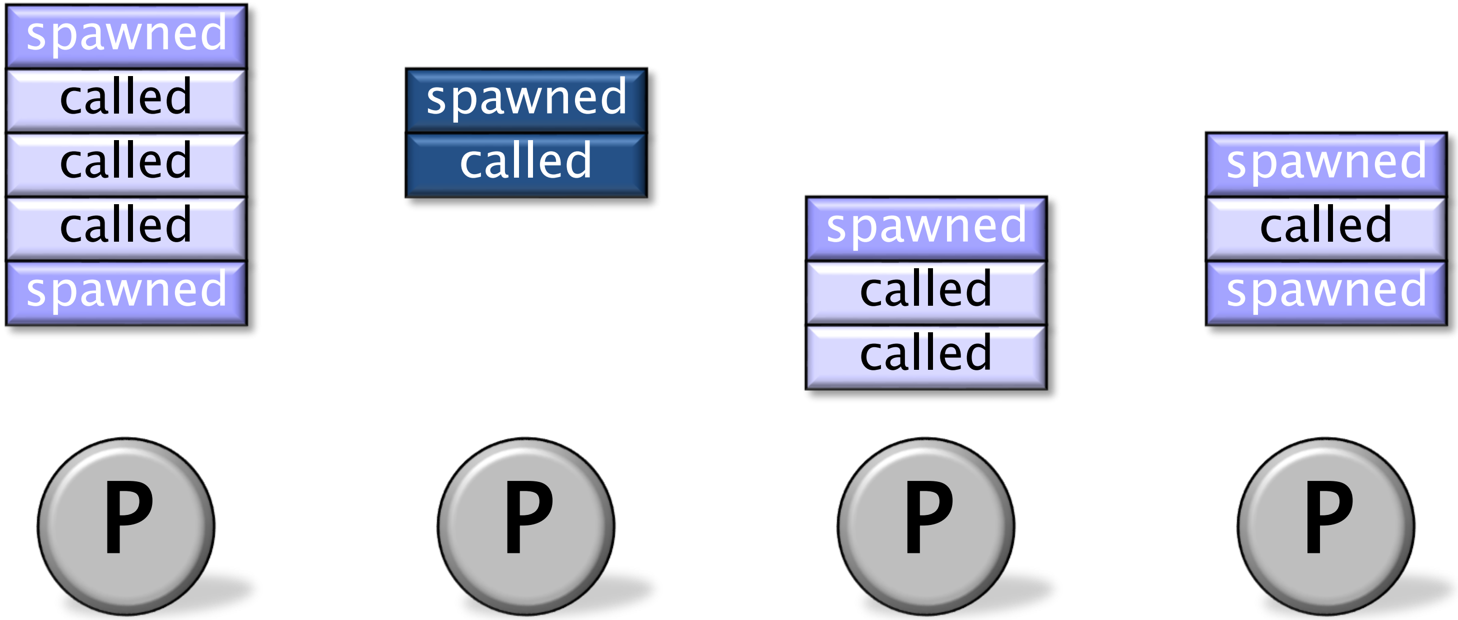
Each worker (processor) maintains a ***work deque*** of ready strands, and it manipulates the bottom of the deque like a stack.



When a worker runs out of work, it steals from the top of a random victim's deque.

Work Stealing

Each worker (processor) maintains a ***work deque*** of ready strands, and it manipulates the bottom of the deque like a stack.



When a worker runs out of work, it steals from the top of a random victim's deque.

Parallel Speedup

T_S — work of a serial program

Suppose the serial program is parallelized.

T_1 — work of the parallel program

T_∞ — span of the parallel program

T_P — running time of the parallel program on P cores

Parallel scalability = T_1/T_P

Parallel speedup = T_S/T_P

Work-Stealing Bounds

Theorem. The Cilk work-stealing scheduler achieves expected running time

$$T_P \approx T_1/P + O(T_\infty)$$

on P processors.

Work-Stealing Bounds

Theorem. The Cilk work-stealing scheduler achieves expected running time

$$T_P \approx T_1/P + O(T_\infty)$$

on P processors.

Time workers spend working.

Work-Stealing Bounds

Theorem. The Cilk work-stealing scheduler achieves expected running time

$$T_P \approx T_1/P + O(T_\infty)$$

on P processors.

Time workers spend **working**.

Time workers spend **stealing**.

Work-Stealing Bounds

Theorem. The Cilk work-stealing scheduler achieves expected running time

$$T_P \approx T_1/P + O(T_\infty)$$

on P processors.

Time workers spend working.

Time workers spend stealing.

If the program has ample parallelism, i.e., $T_1/T_\infty \gg P$, then the first term dominates, and $T_P \approx T_1/P$.

Parallel Speedup

T_S — work of a serial program

Suppose the serial program is parallelized.

T_1 — work of the parallel program

T_∞ — span of the parallel program

T_P — running time of the parallel program on P cores

Parallel scalability $= T_1/T_P$

Parallel speedup $= T_S/T_P$

To achieve linear speedup on P processors over the serial program, i.e., $T_P \approx T_S/P$, we need :

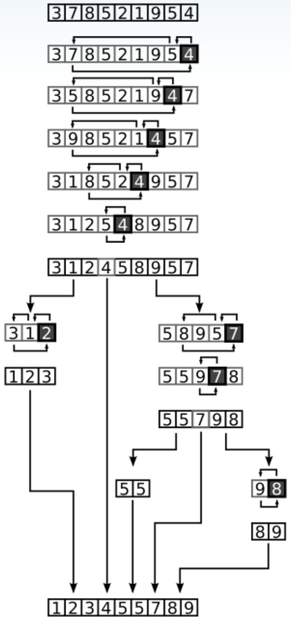
1. Ample parallelism: $T_1/T_\infty \gg P$.
2. High work efficiency: $T_S/T_1 \approx 1$.

The Work-First Principle

To optimize the execution of programs with sufficient parallelism, the implementation of the Cilk scheduler aims to maintain high work efficiency by abiding by the work-first principle:

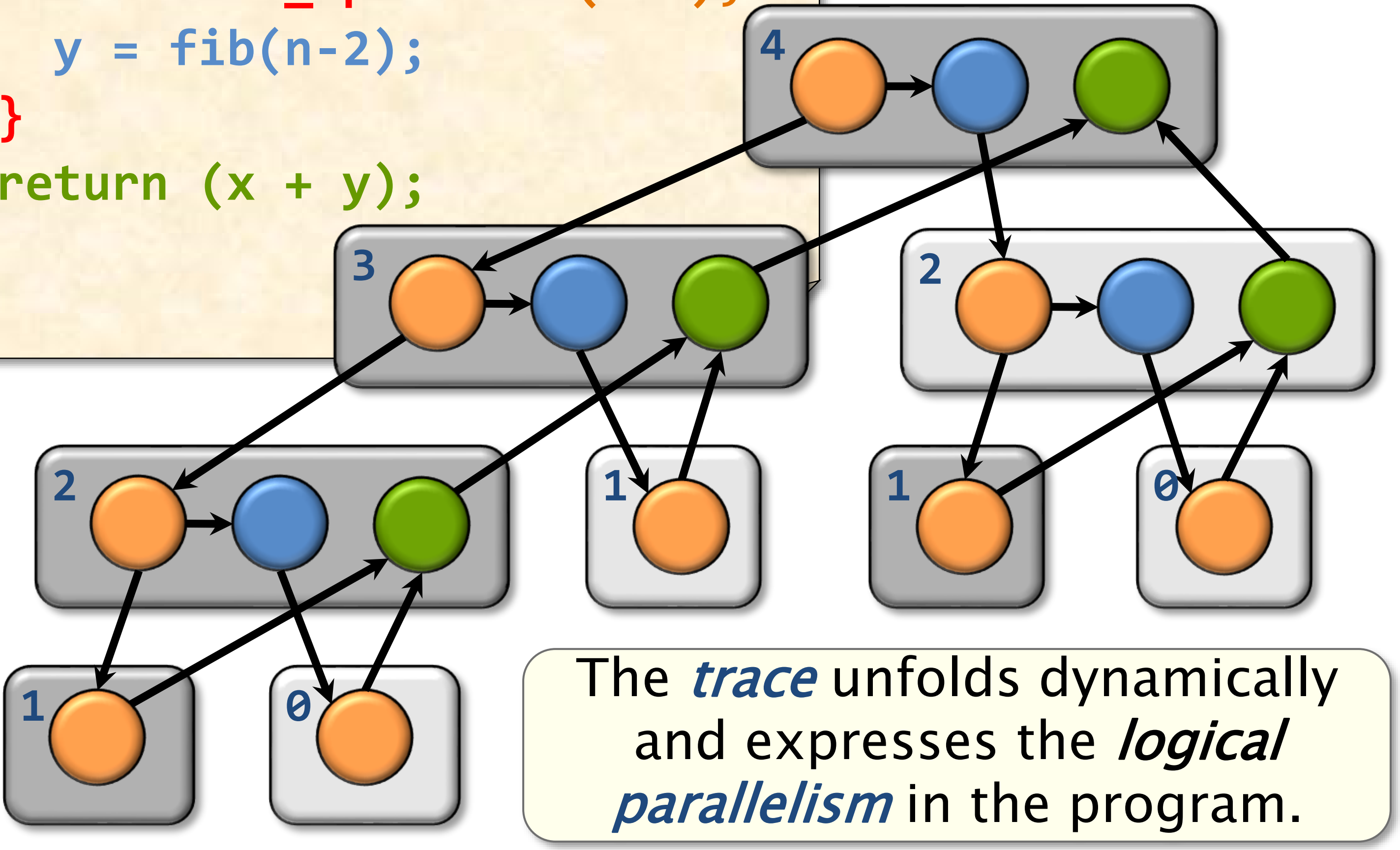
Optimize for ordinary serial execution, at the expense of some additional overhead in steals.

CORE FUNCTIONALITIES FOR WORK STEALING



Cilk's Execution Model

```
int fib(int n) {
  if (n < 2) return n;
  int x, y;
  cilk_scope {
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
  }
  return (x + y);
}
```



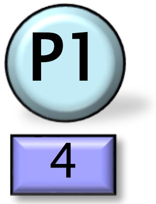
Example:
fib(4)

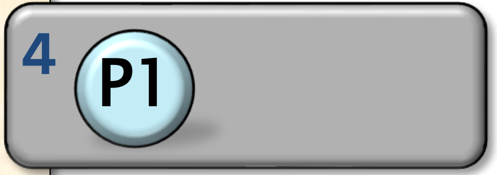
The *trace* unfolds dynamically and expresses the *logical parallelism* in the program.

Workers Mirror Serial Execution

P1 %rip →

```
int fib(int n) {
  if (n < 2) return n;
  int x, y;
  cilk_scope {
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
  }
  return (x + y);
}
```





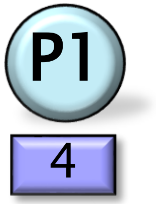
Example:

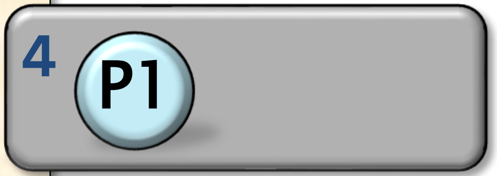
fib(4)

Workers Mirror Serial Execution

P1 %rip →

```
int fib(int n) {
  if (n < 2) return n;
  int x, y;
  cilk_scope {
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
  }
  return (x + y);
}
```

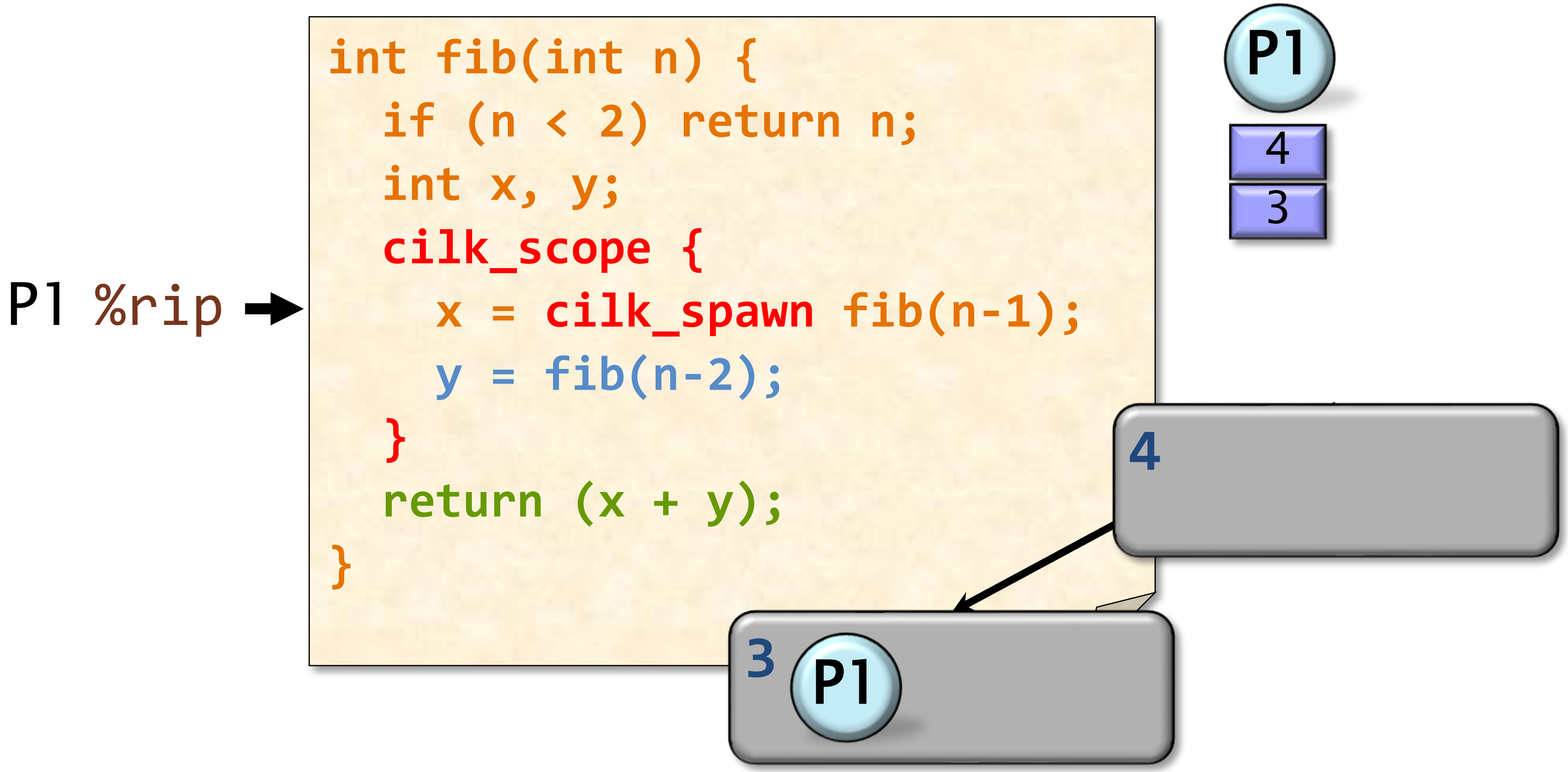




Example:

fib(4)

Workers Mirror Serial Execution



```
int fib(int n) {
  if (n < 2) return n;
  int x, y;
  cilk_scope {
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
  }
  return (x + y);
}
```

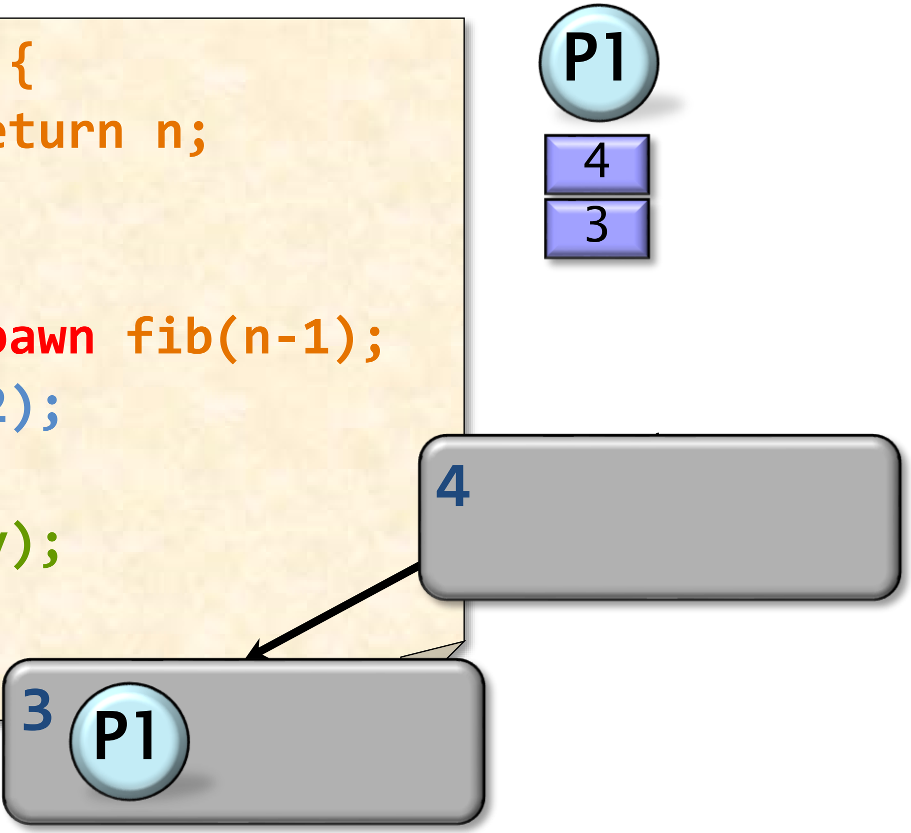
Example:

fib(4)

Workers Mirror Serial Execution

P1 %rip →

```
int fib(int n) {
  if (n < 2) return n;
  int x, y;
  cilk_scope {
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
  }
  return (x + y);
}
```



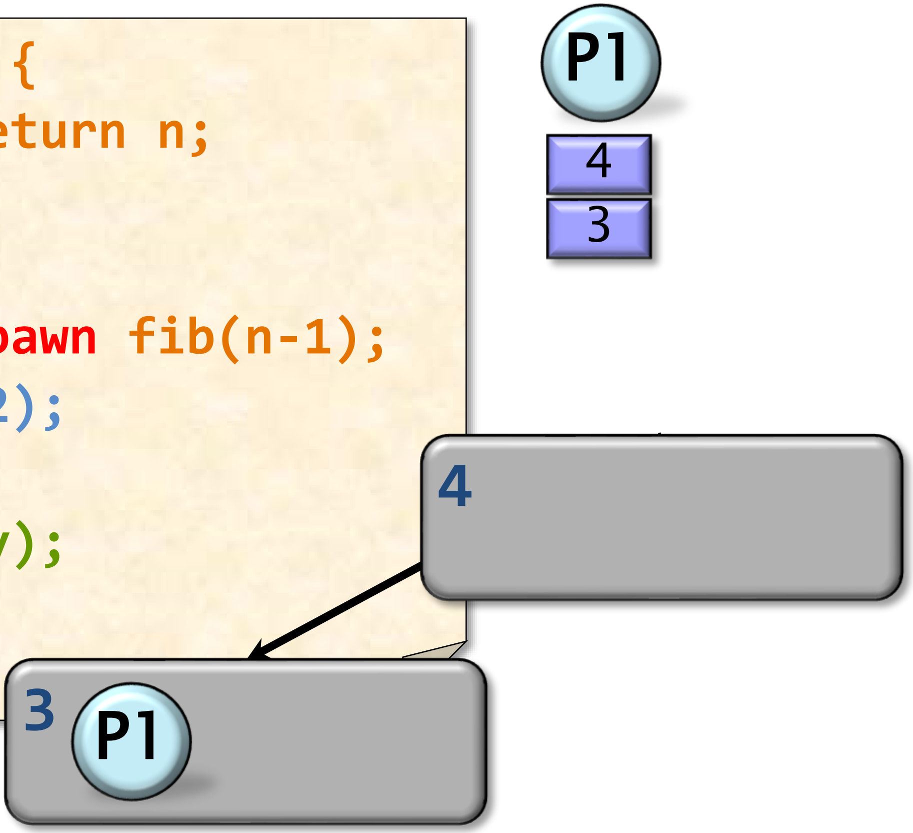
Example:

fib(4)

Workers Mirror Serial Execution

P1 %rip →

```
int fib(int n) {
  if (n < 2) return n;
  int x, y;
  cilk_scope {
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
  }
  return (x + y);
}
```



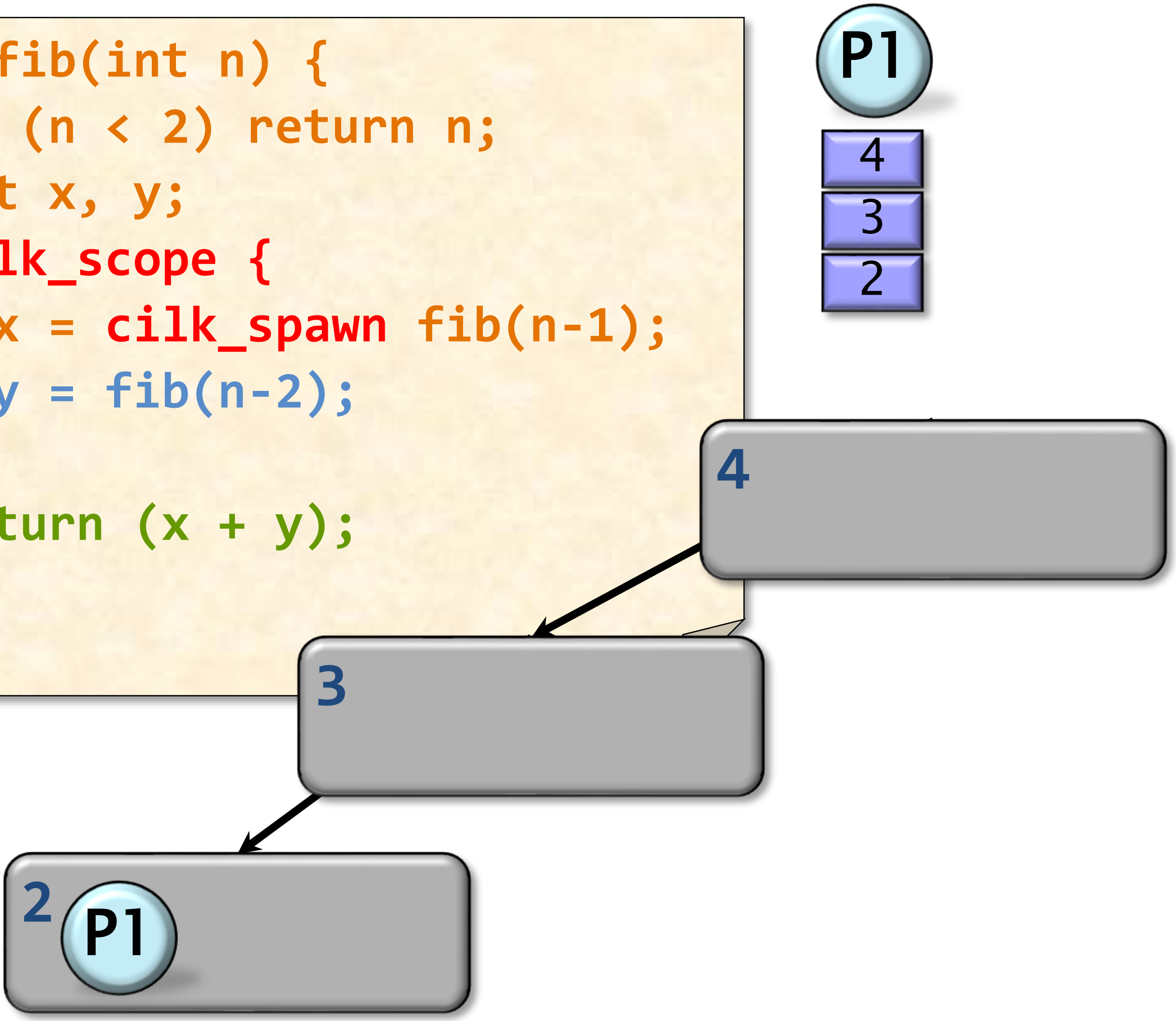
Example:

fib(4)

Workers Mirror Serial Execution

P1 %rip →

```
int fib(int n) {
  if (n < 2) return n;
  int x, y;
  cilk_scope {
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
  }
  return (x + y);
}
```

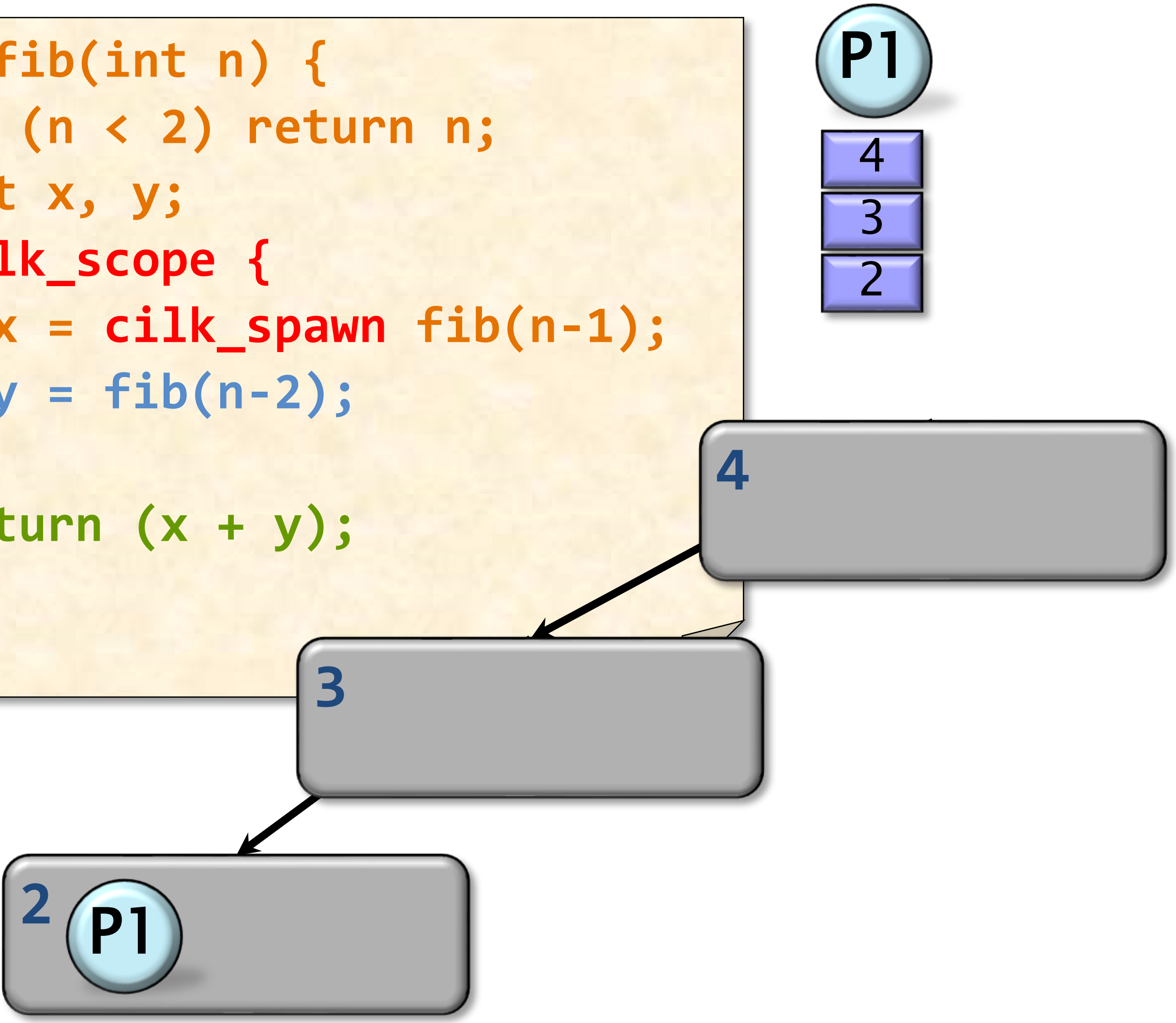


Example:
fib(4)

Workers Mirror Serial Execution

P1 %rip →

```
int fib(int n) {
  if (n < 2) return n;
  int x, y;
  cilk_scope {
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
  }
  return (x + y);
}
```

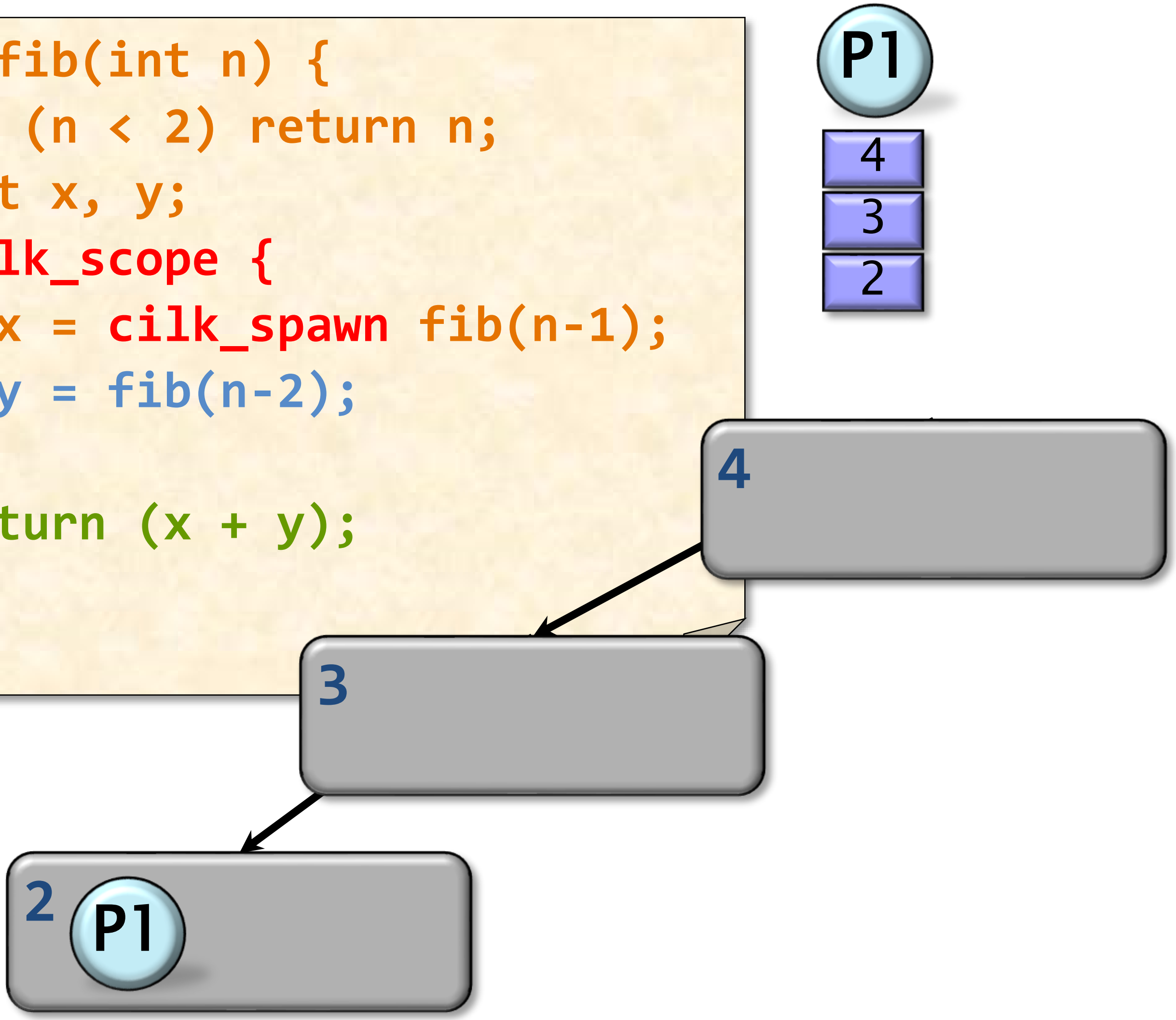


Example:
fib(4)

Workers Mirror Serial Execution

P1 %rip →

```
int fib(int n) {
  if (n < 2) return n;
  int x, y;
  cilk_scope {
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
  }
  return (x + y);
}
```



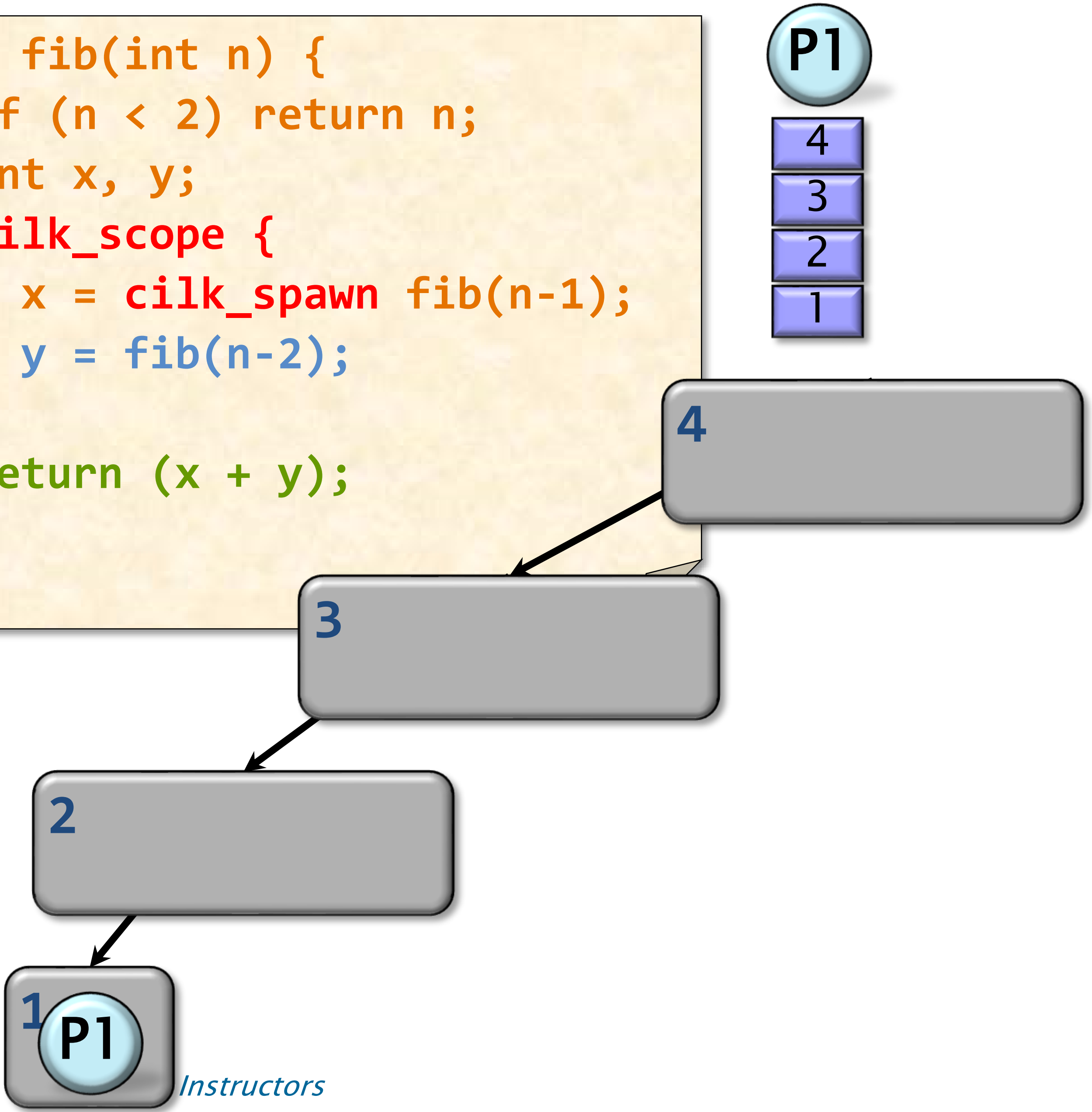
Example:

fib(4)

Workers Mirror Serial Execution

P1 %rip →

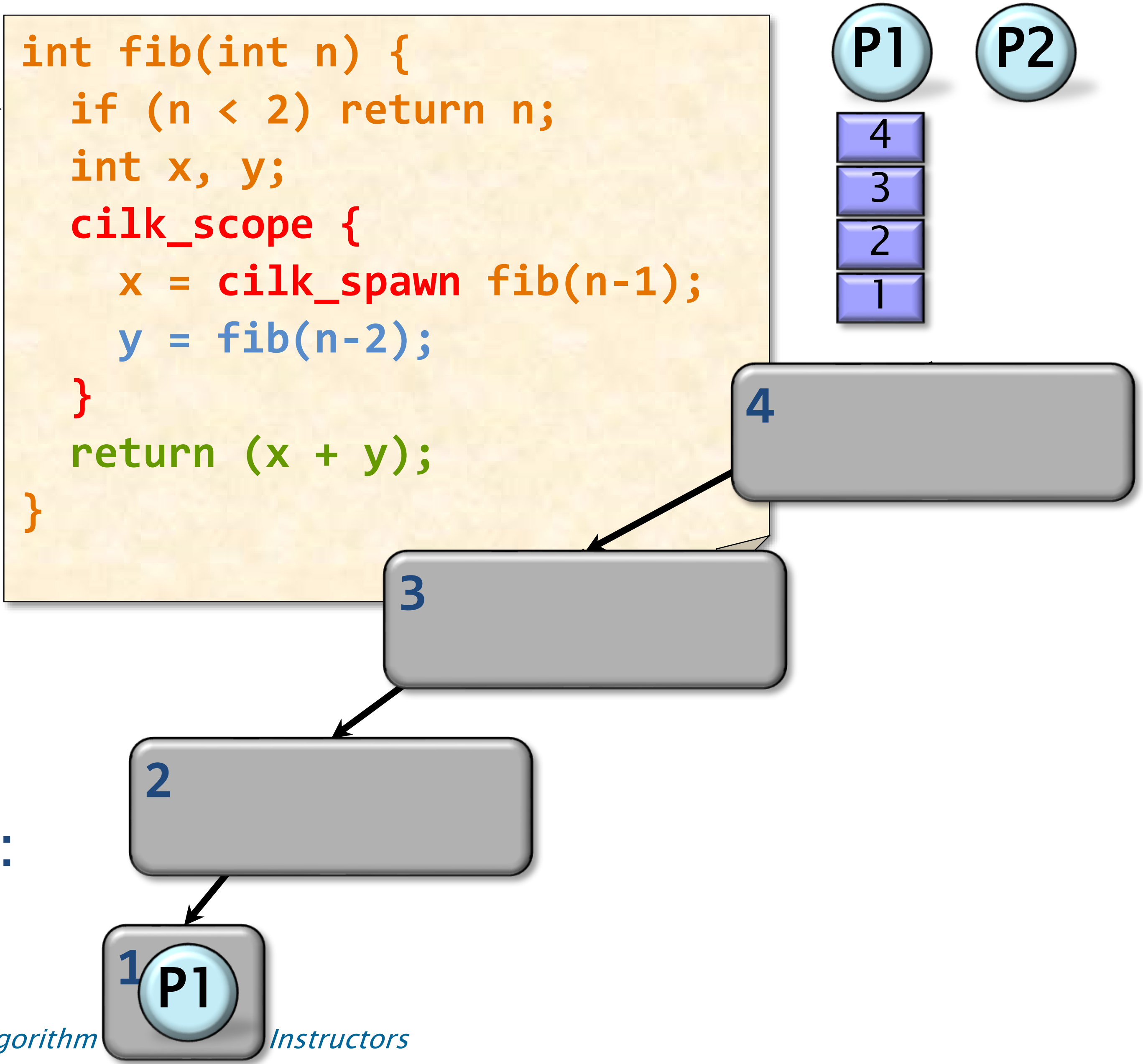
```
int fib(int n) {
  if (n < 2) return n;
  int x, y;
  cilk_scope {
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
  }
  return (x + y);
}
```



Example:
fib(4)

Successful Steals Create Parallelism

P1 %rip →



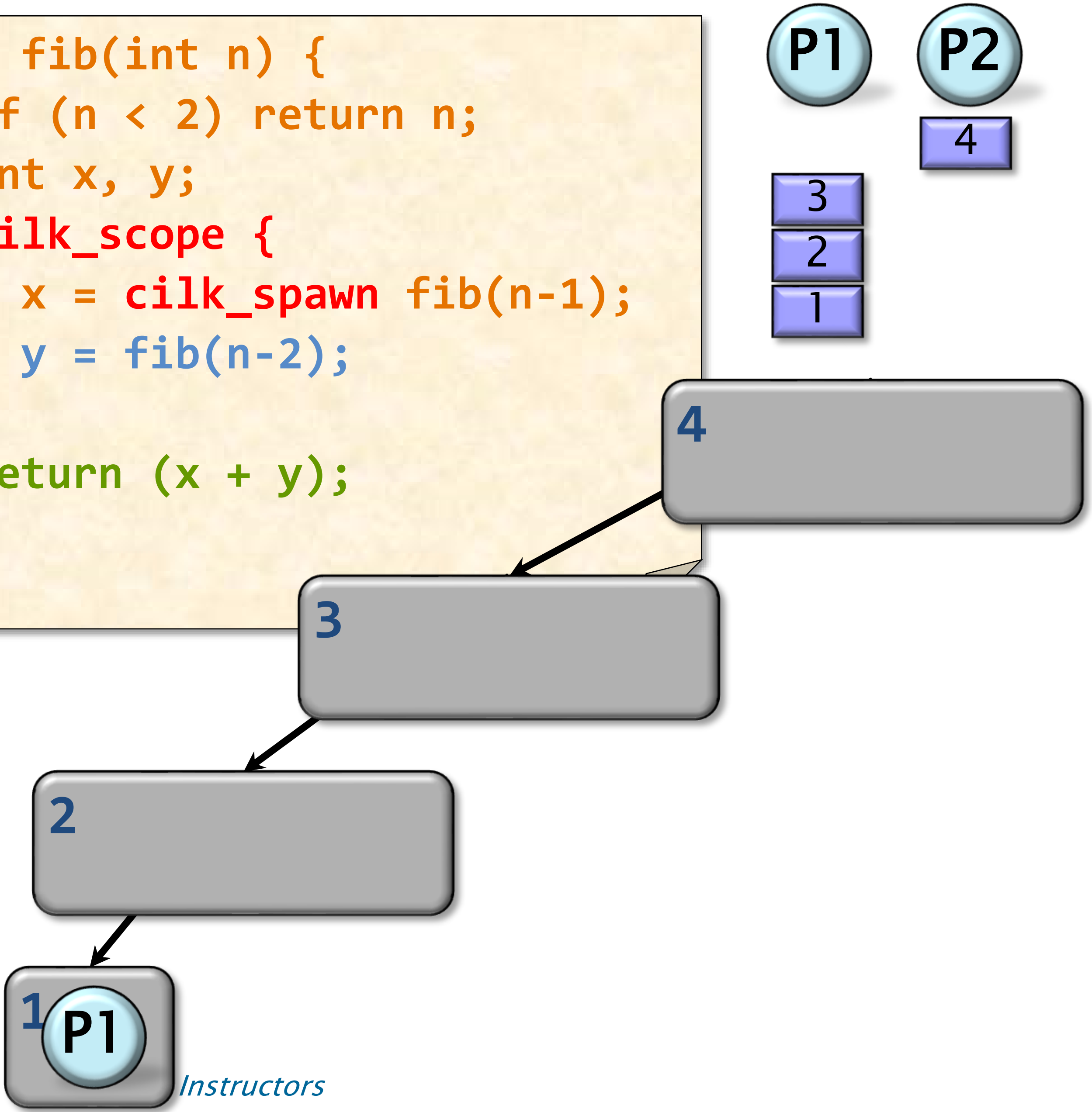
```
int fib(int n) {
  if (n < 2) return n;
  int x, y;
  cilk_scope {
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
  }
  return (x + y);
}
```

Example:
fib(4)

Successful Steals Create Parallelism

P1 %rip →

```
int fib(int n) {
  if (n < 2) return n;
  int x, y;
  cilk_scope {
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
  }
  return (x + y);
}
```



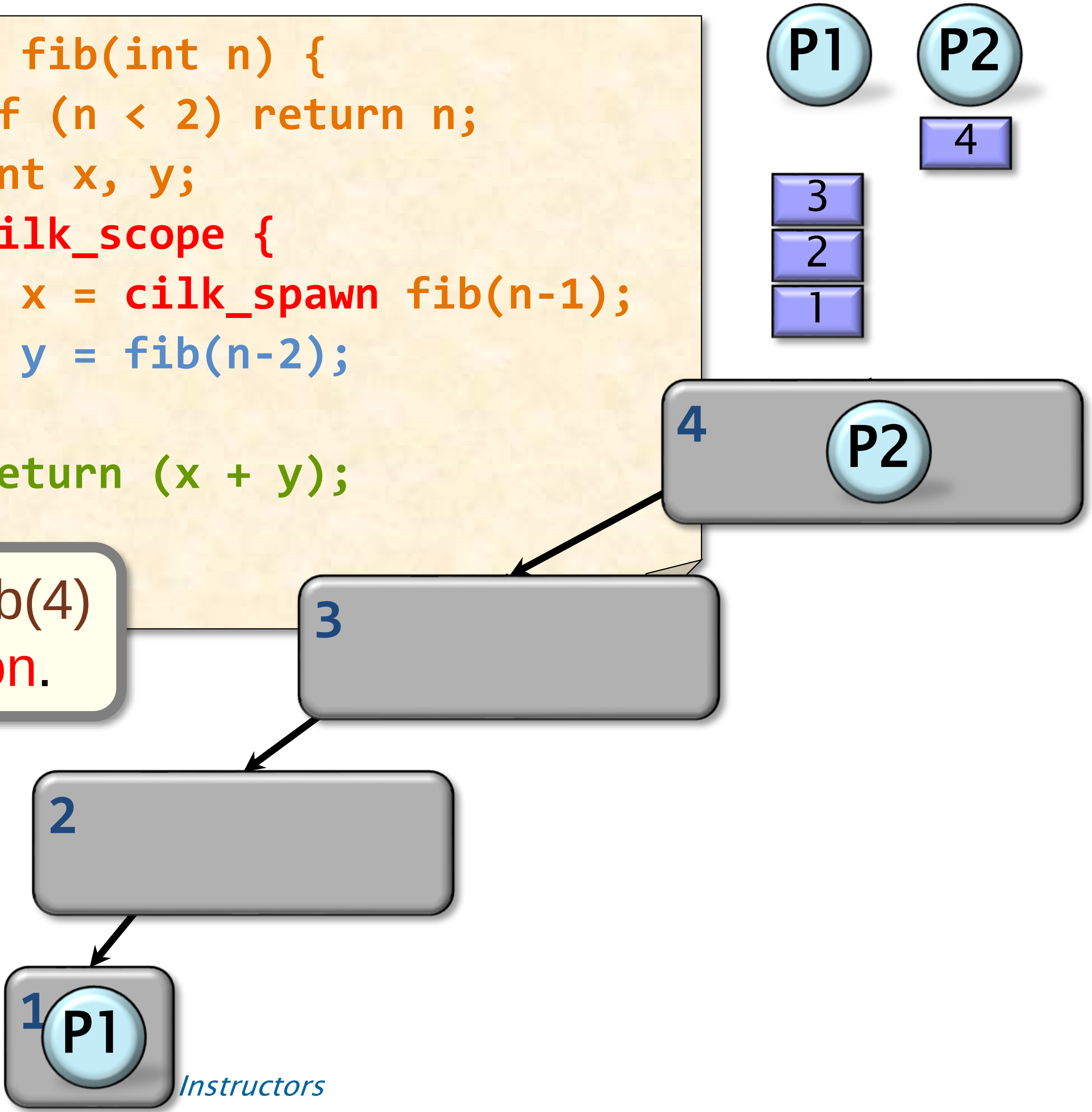
Example:
fib(4)

Successful Steals Create Parallelism

P1 %rip →

P2 %rip →

```
int fib(int n) {
  if (n < 2) return n;
  int x, y;
  cilk_scope {
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
  }
  return (x + y);
}
```



P2 resumes fib(4) mid-execution.

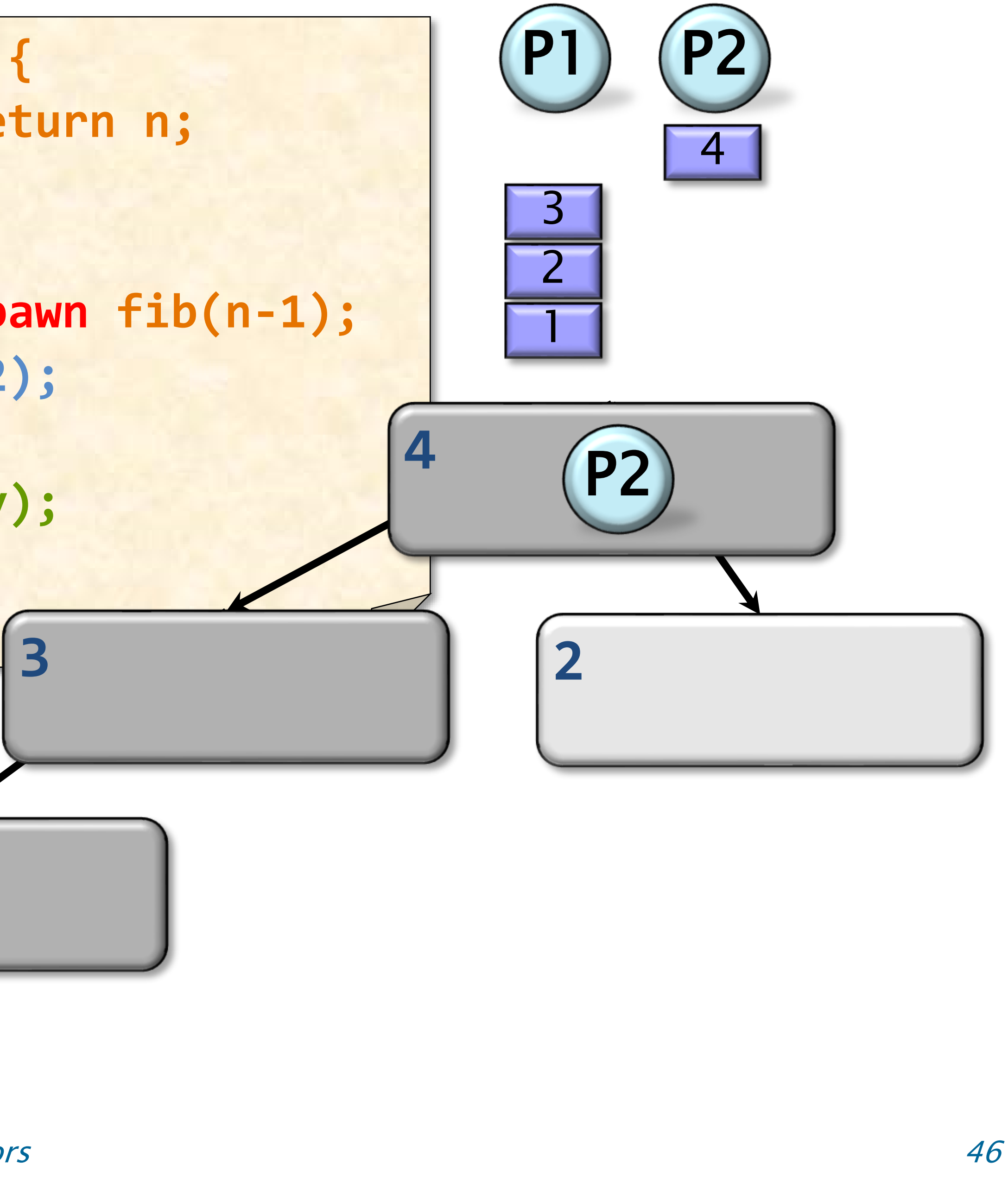
Example:
fib(4)

Successful Steals Create Parallelism

P1 %rip →

P2 %rip →

```
int fib(int n) {
  if (n < 2) return n;
  int x, y;
  cilk_scope {
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
  }
  return (x + y);
}
```



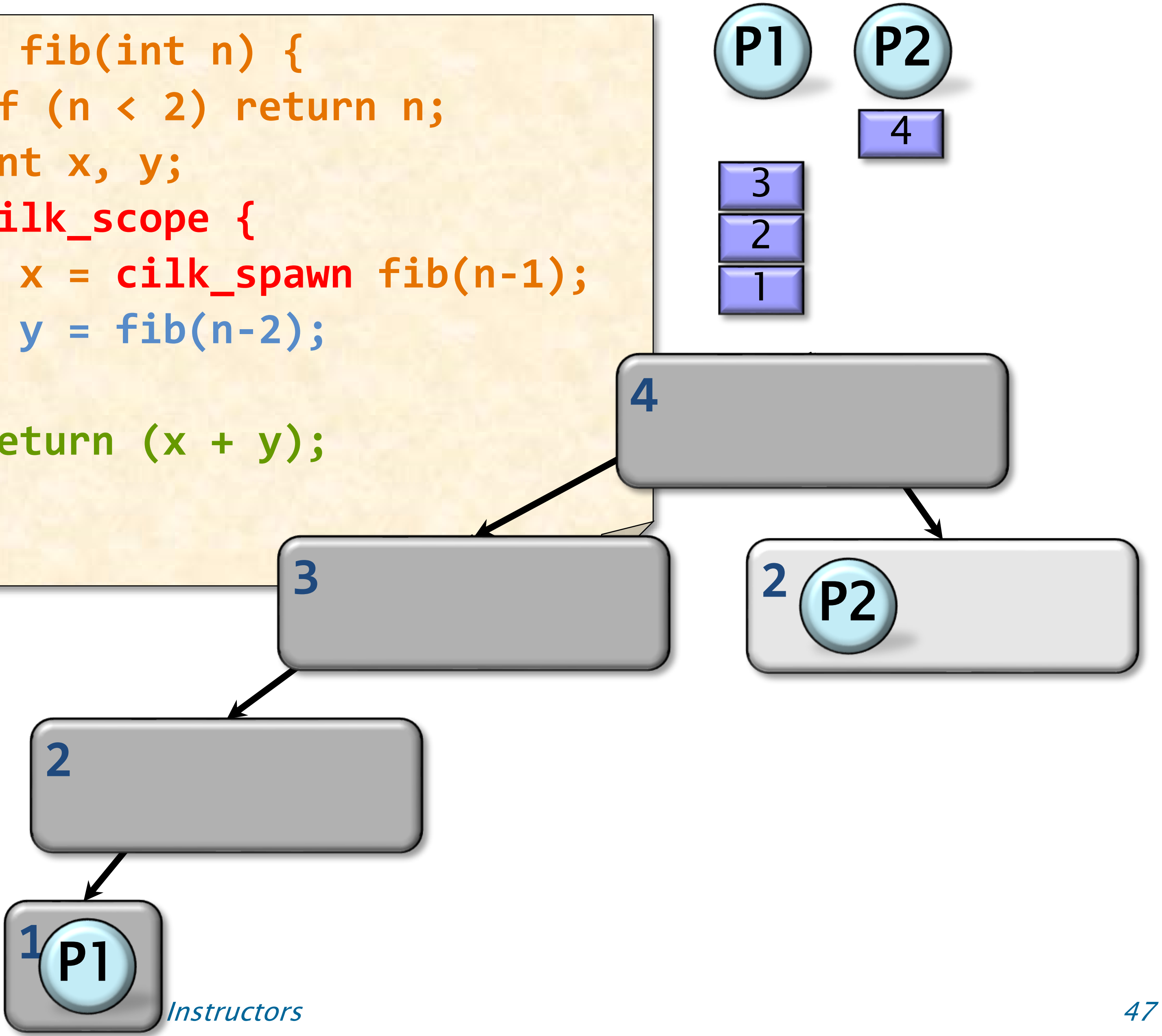
Example:
fib(4)

Successful Steals Create Parallelism

P1 %rip →

P2 %rip →

```
int fib(int n) {
  if (n < 2) return n;
  int x, y;
  cilk_scope {
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
  }
  return (x + y);
}
```



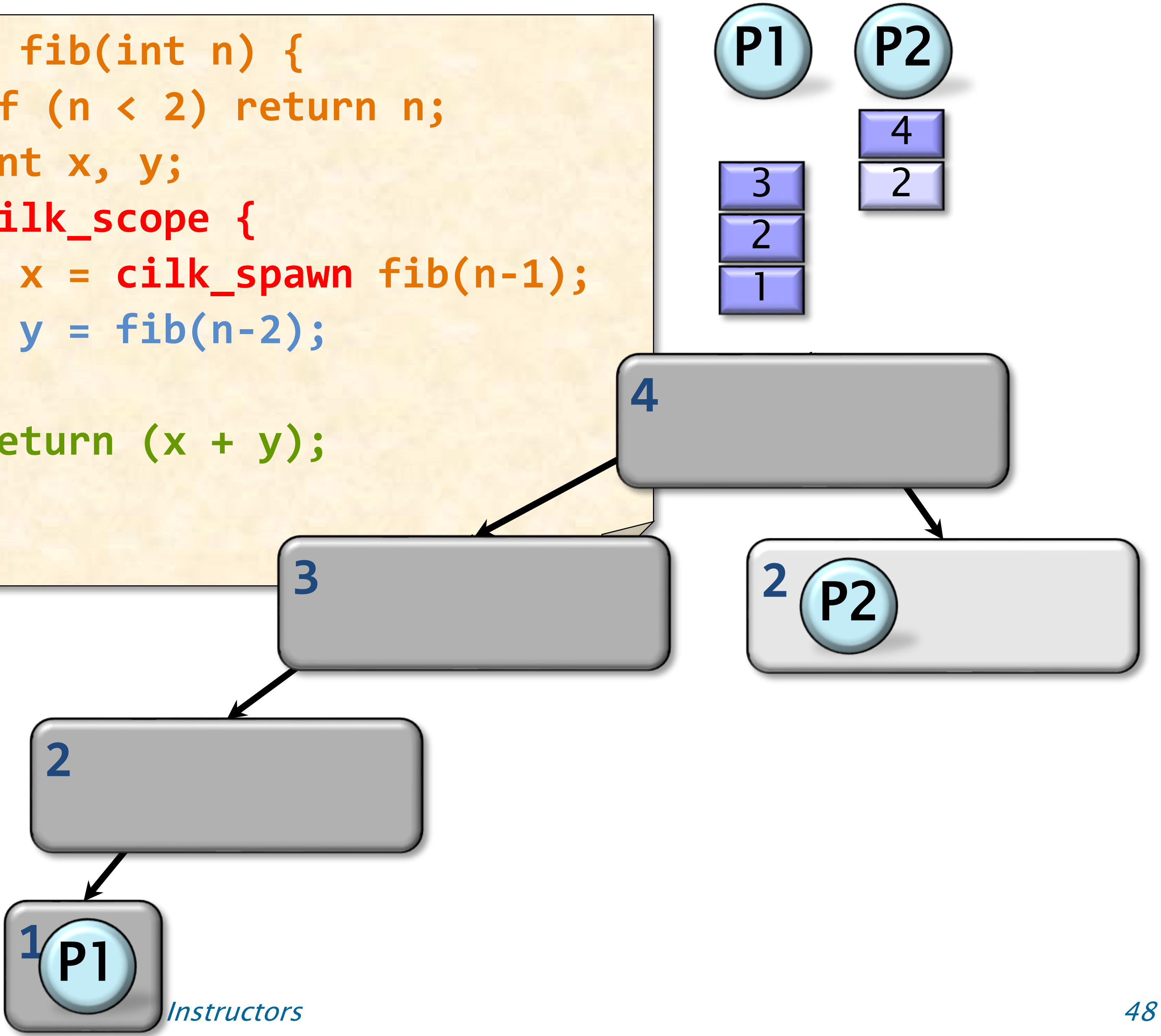
Example:
fib(4)

Successful Steals Create Parallelism

P2 %rip →

P1 %rip →

```
int fib(int n) {
  if (n < 2) return n;
  int x, y;
  cilk_scope {
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
  }
  return (x + y);
}
```



Example:
fib(4)

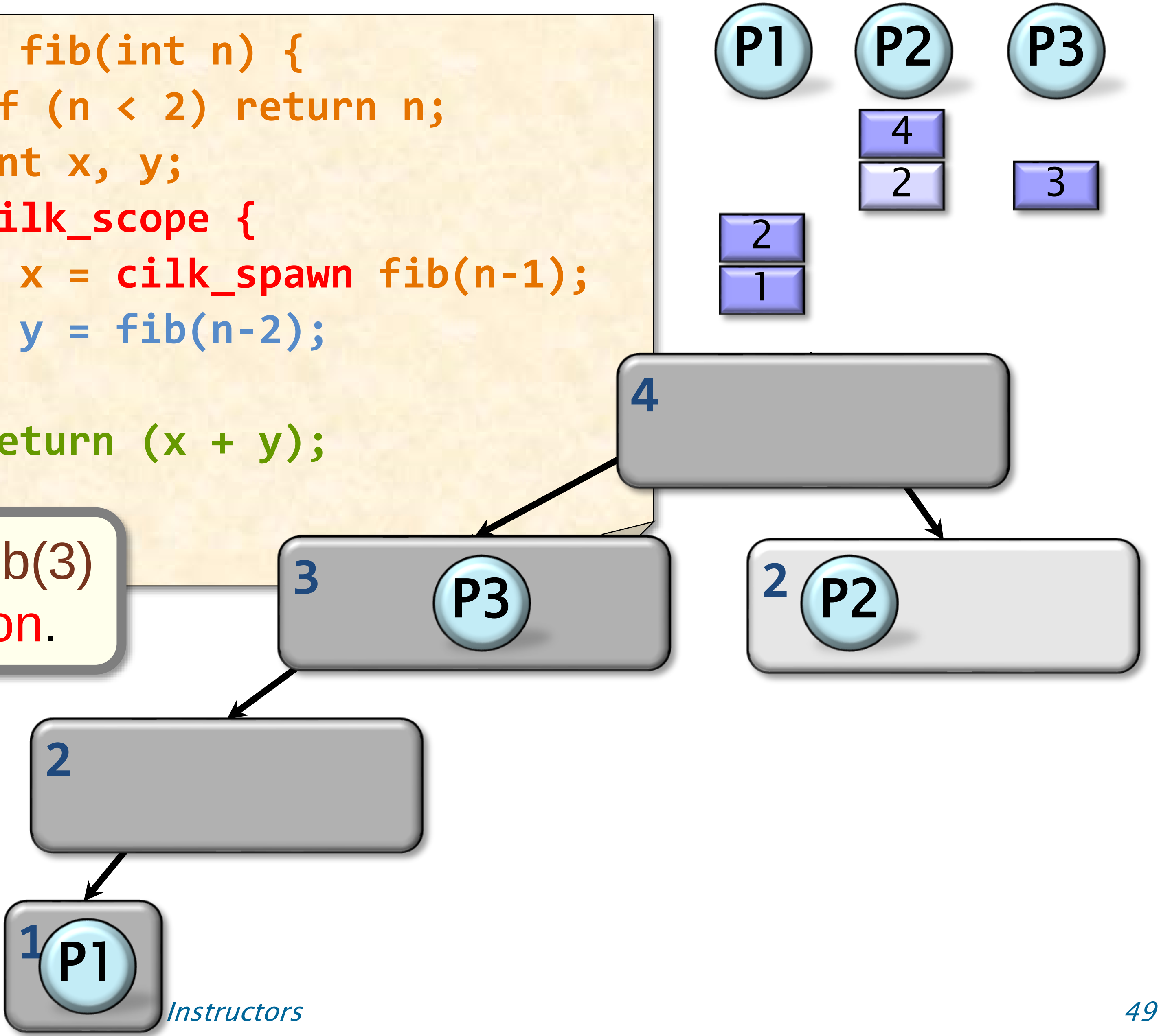
Successful Steals Create Parallelism

P2 %rip →
P1 %rip →

P3 %rip →

```
int fib(int n) {
  if (n < 2) return n;
  int x, y;
  cilk_scope {
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
  }
  return (x + y);
}
```

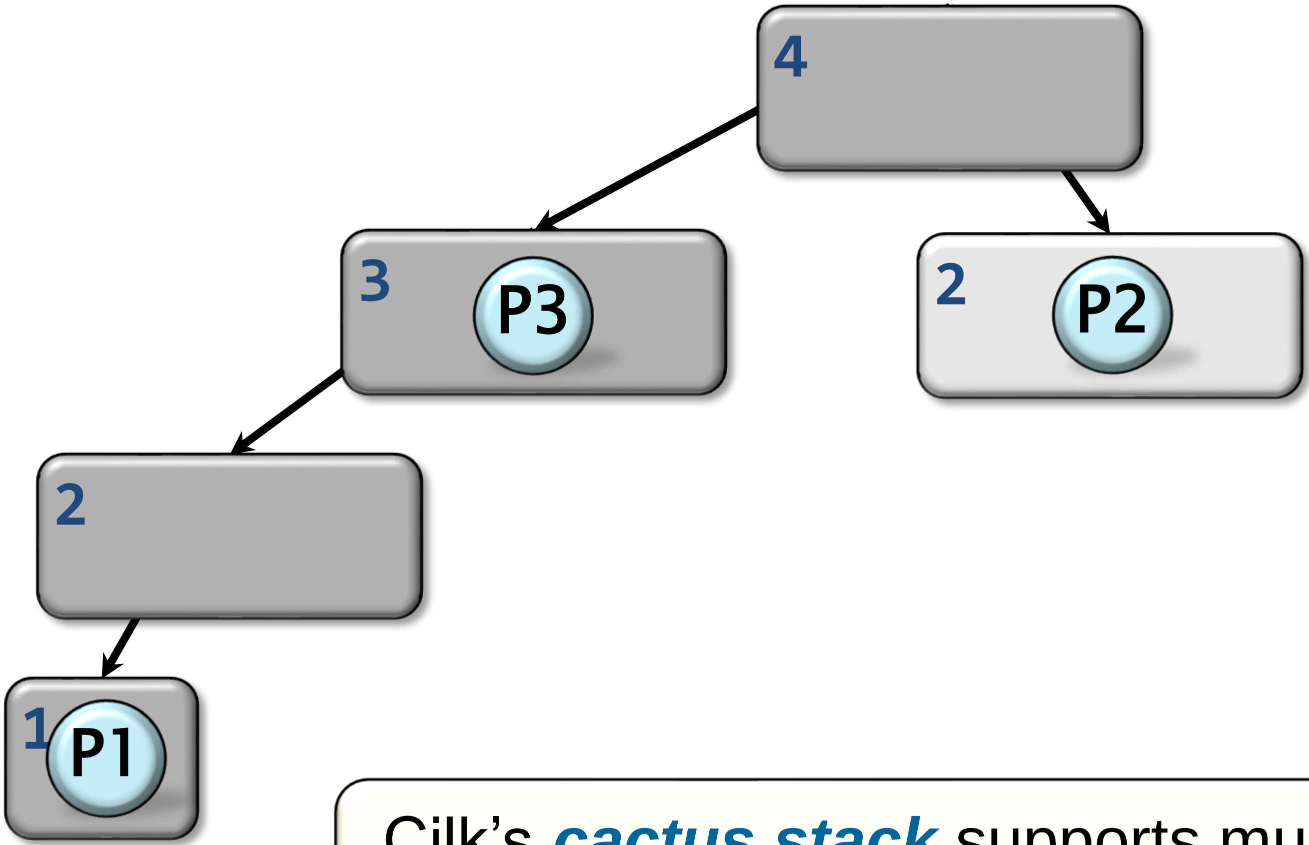
P3 resumes fib(3) mid-execution.

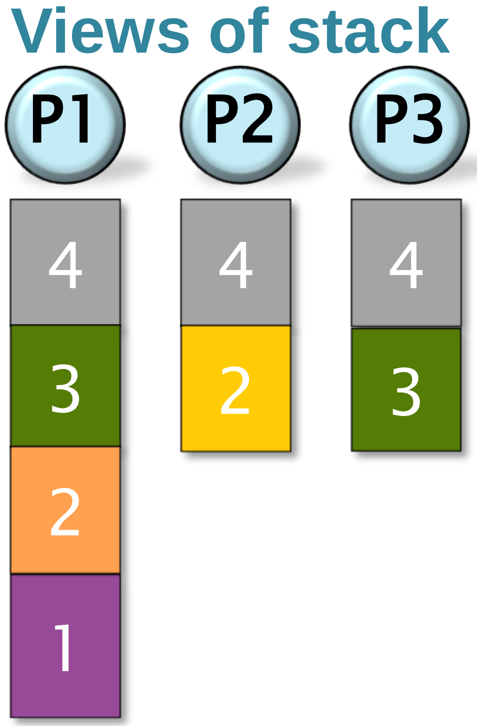


Example:
fib(4)

Cactus Stack

Cilk supports C's **rule for pointers**: A pointer to stack space can be passed from parent to child, but not from child to parent.



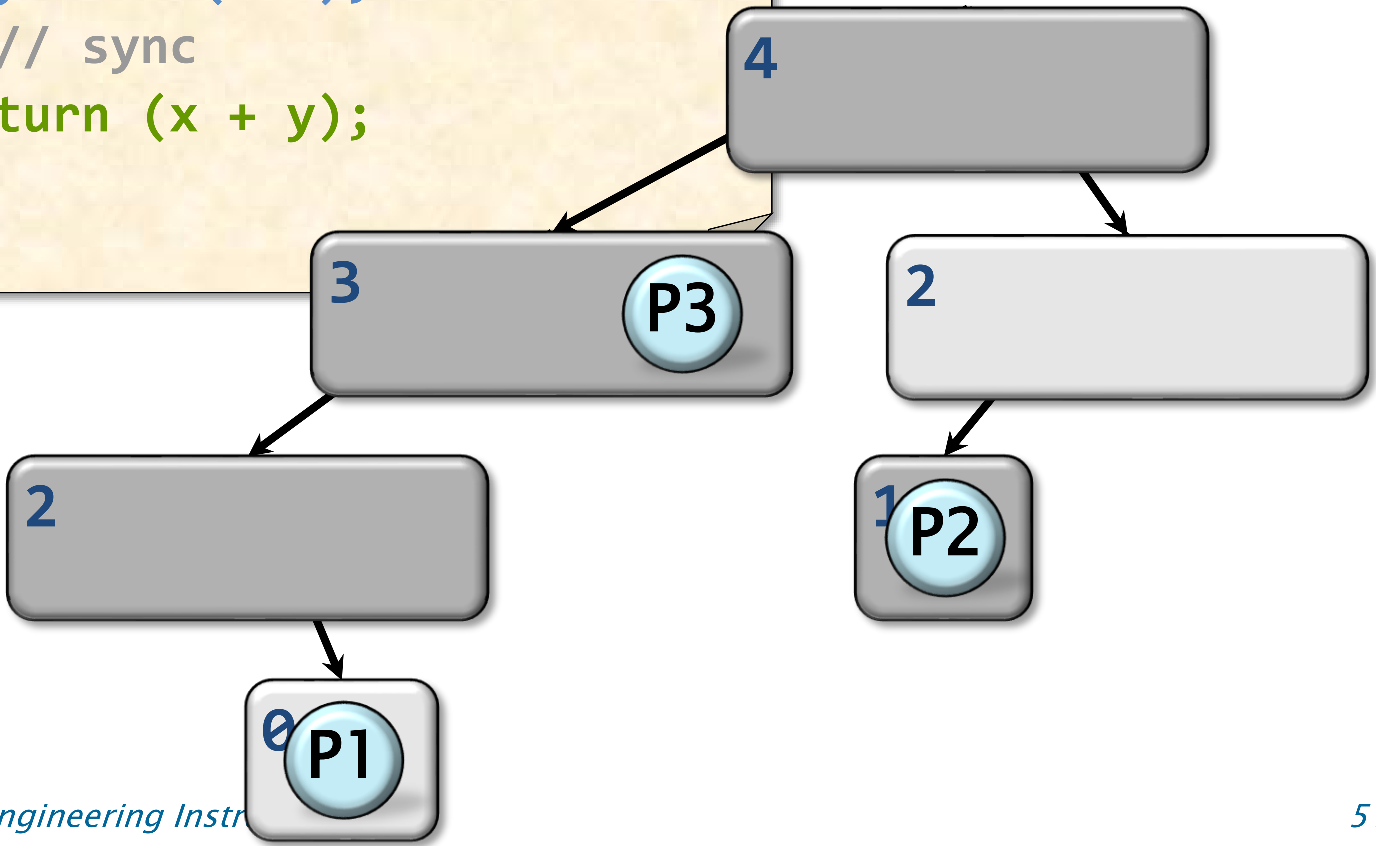


Cilk's ***cactus stack*** supports multiple views in parallel.

Syncs (cilk_scope)

P2 %rip →
P1 %rip →
P3 %rip →

```
int fib(int n) {
  if (n < 2) return n;
  int x, y;
  cilk_scope {
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
  } // sync
  return (x + y);
}
```



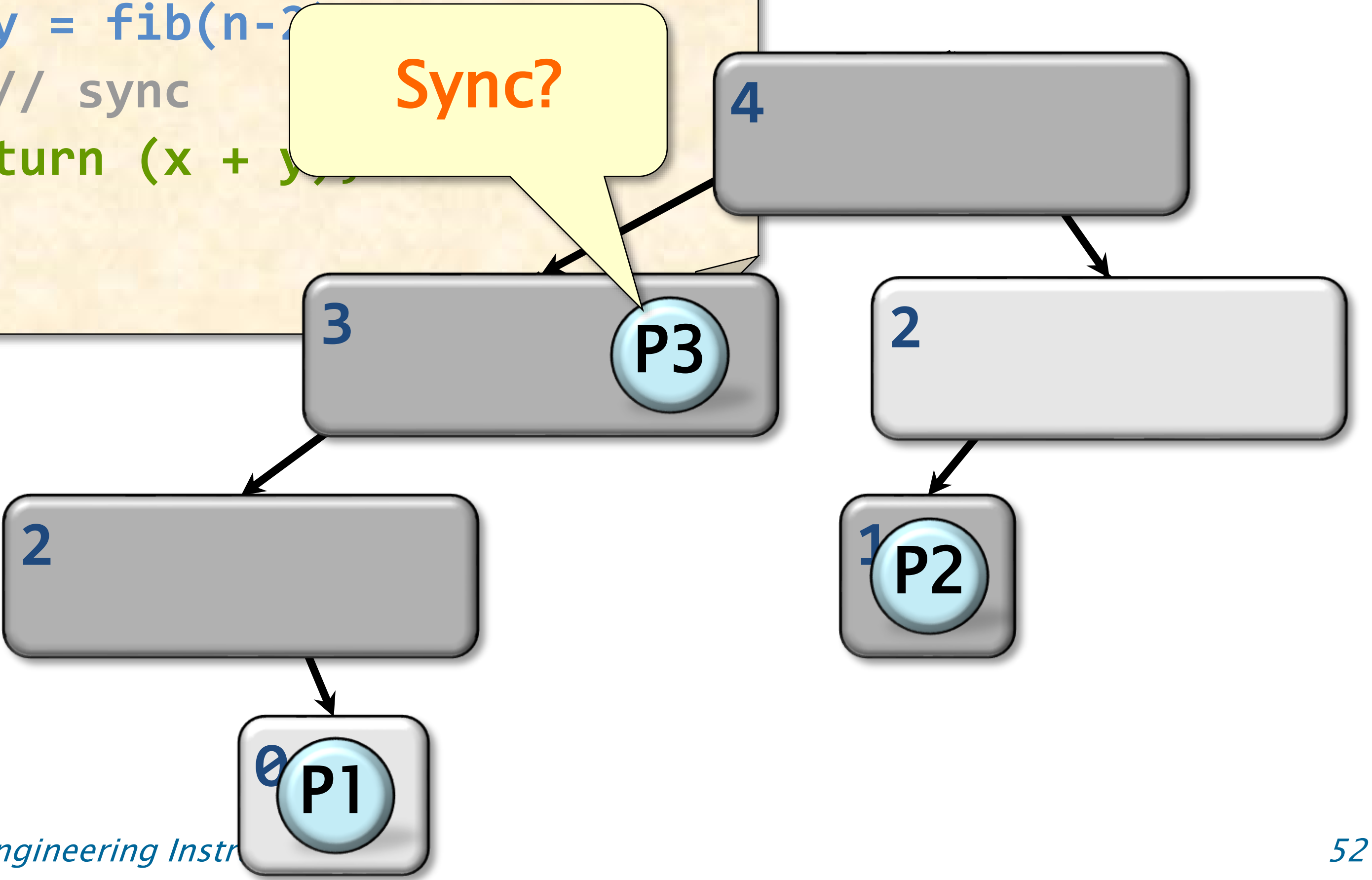
Example:
fib(4)

Syncs (cilk_scope)

P2 %rip →
P1 %rip →

P3 %rip →

```
int fib(int n) {
  if (n < 2) return n;
  int x, y;
  cilk_scope {
    x = cilk_spawn fib(n-1);
    y = fib(n-2
  } // sync
  return (x + y);
}
```



Example:
fib(4)

Syncs (cilk_scope)

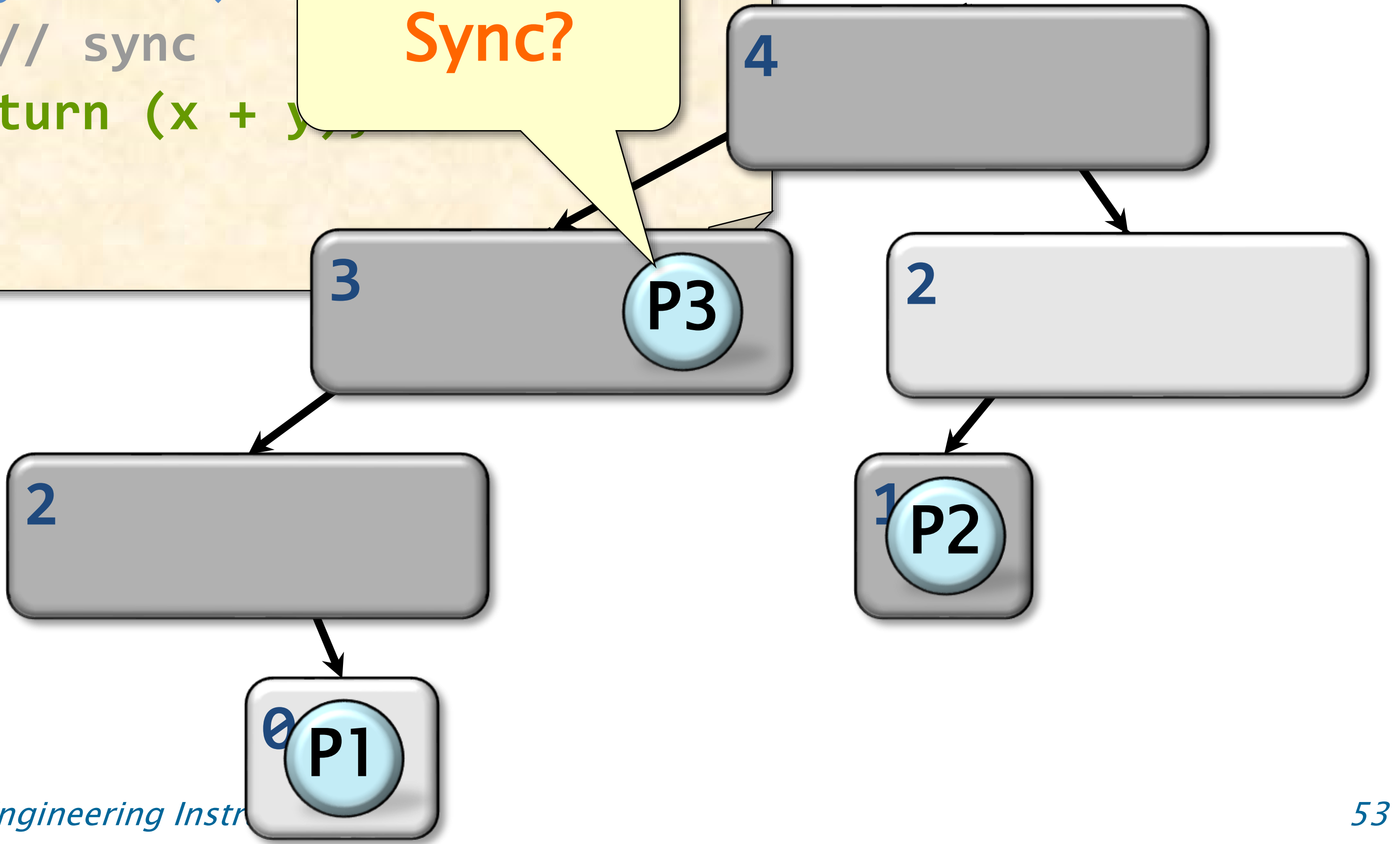
P2 %rip →

P1 %rip →

P3 %rip →

```
int fib(int n) {
  if (n < 2) return n;
  int x, y;
  cilk_scope {
    x = cilk_spawn fib(n-1);
    y = fib(n-2
  } // sync
  return (x + y);
}
```

Sync?



Example:

fib(4)

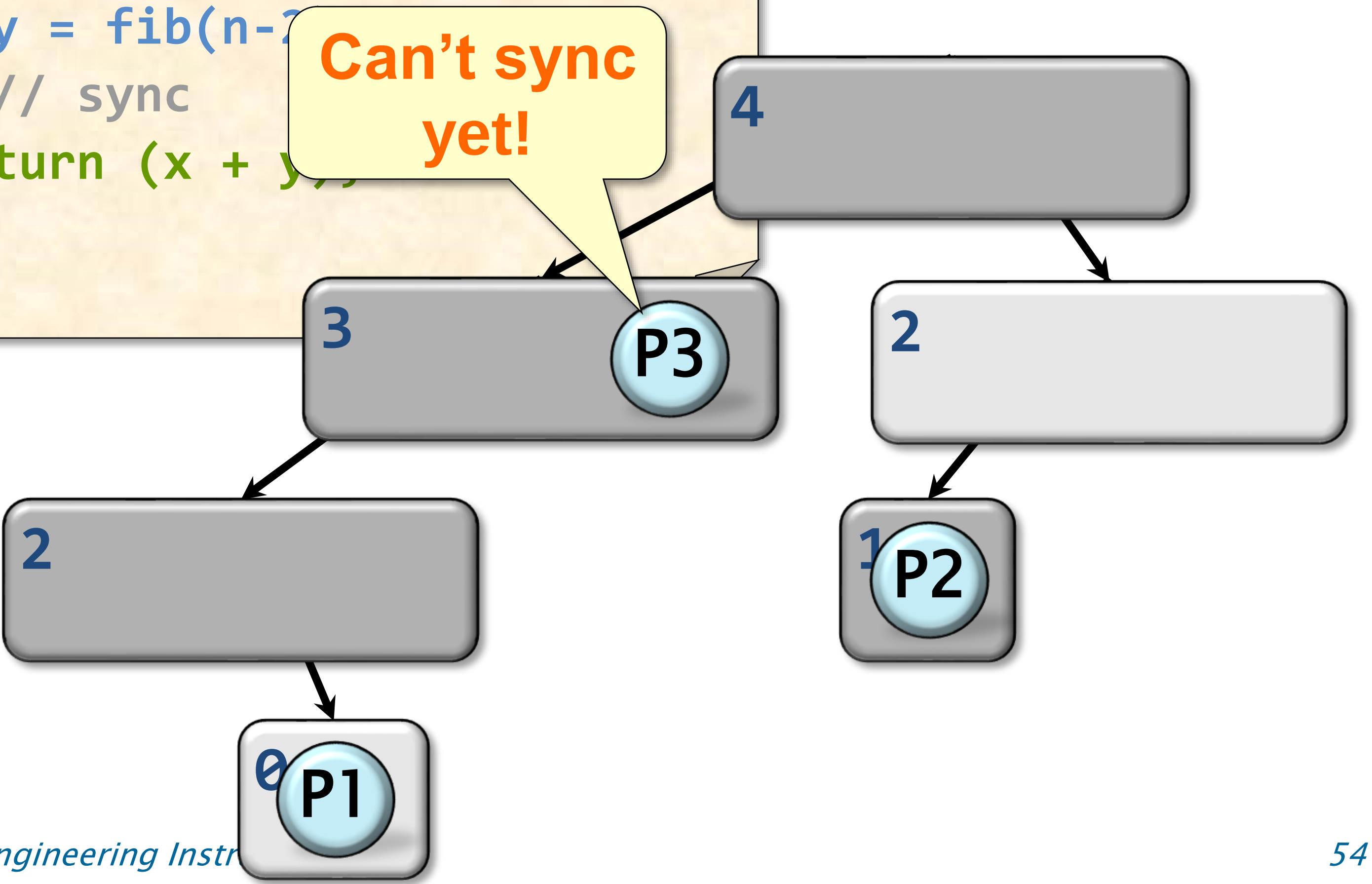
Syncs (cilk_scope)

P2 %rip →

P1 %rip →

P3 %rip →

```
int fib(int n) {
  if (n < 2) return n;
  int x, y;
  cilk_scope {
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
  } // sync
  return (x + y);
}
```



Can't sync yet!

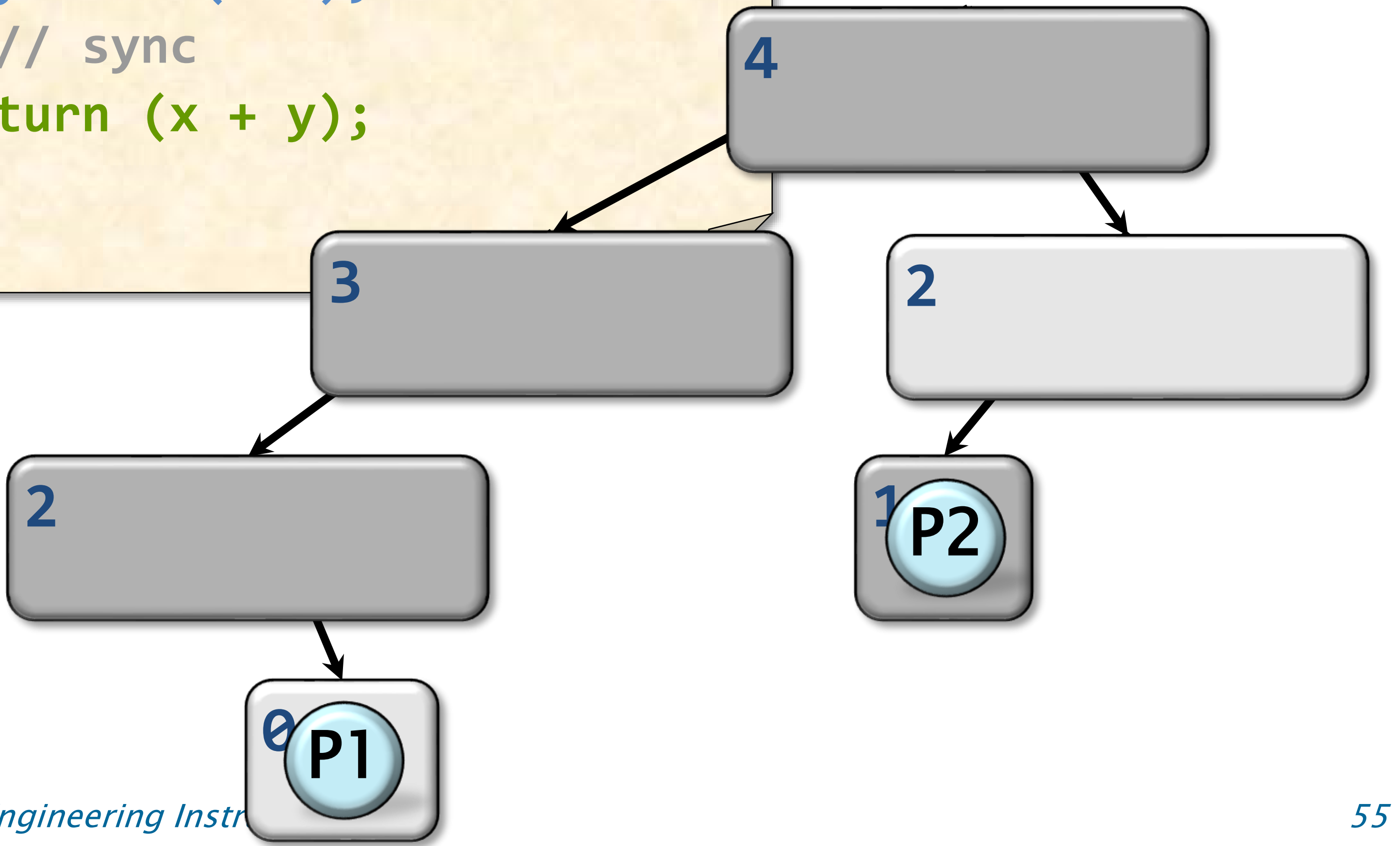
Example: fib(4)

Syncs (cilk_scope)

P2 %rip →

P1 %rip →

```
int fib(int n) {
  if (n < 2) return n;
  int x, y;
  cilk_scope {
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
  } // sync
  return (x + y);
}
```

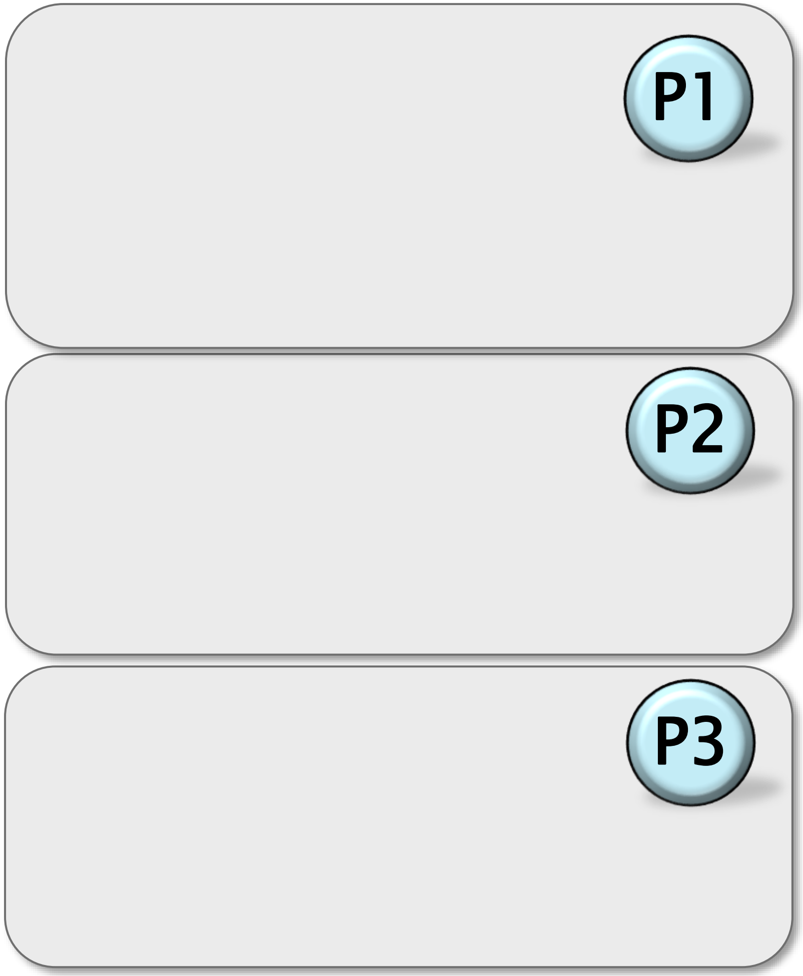


Example:

fib(4)

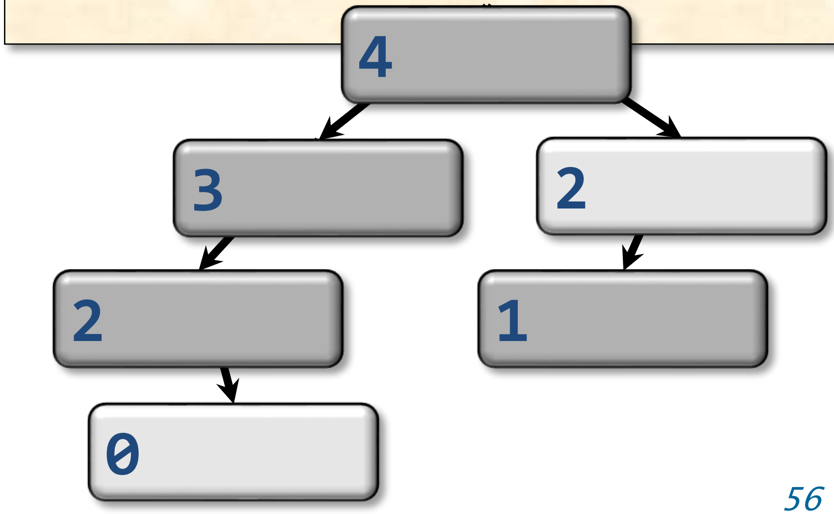
Putting Everything Together

Workers

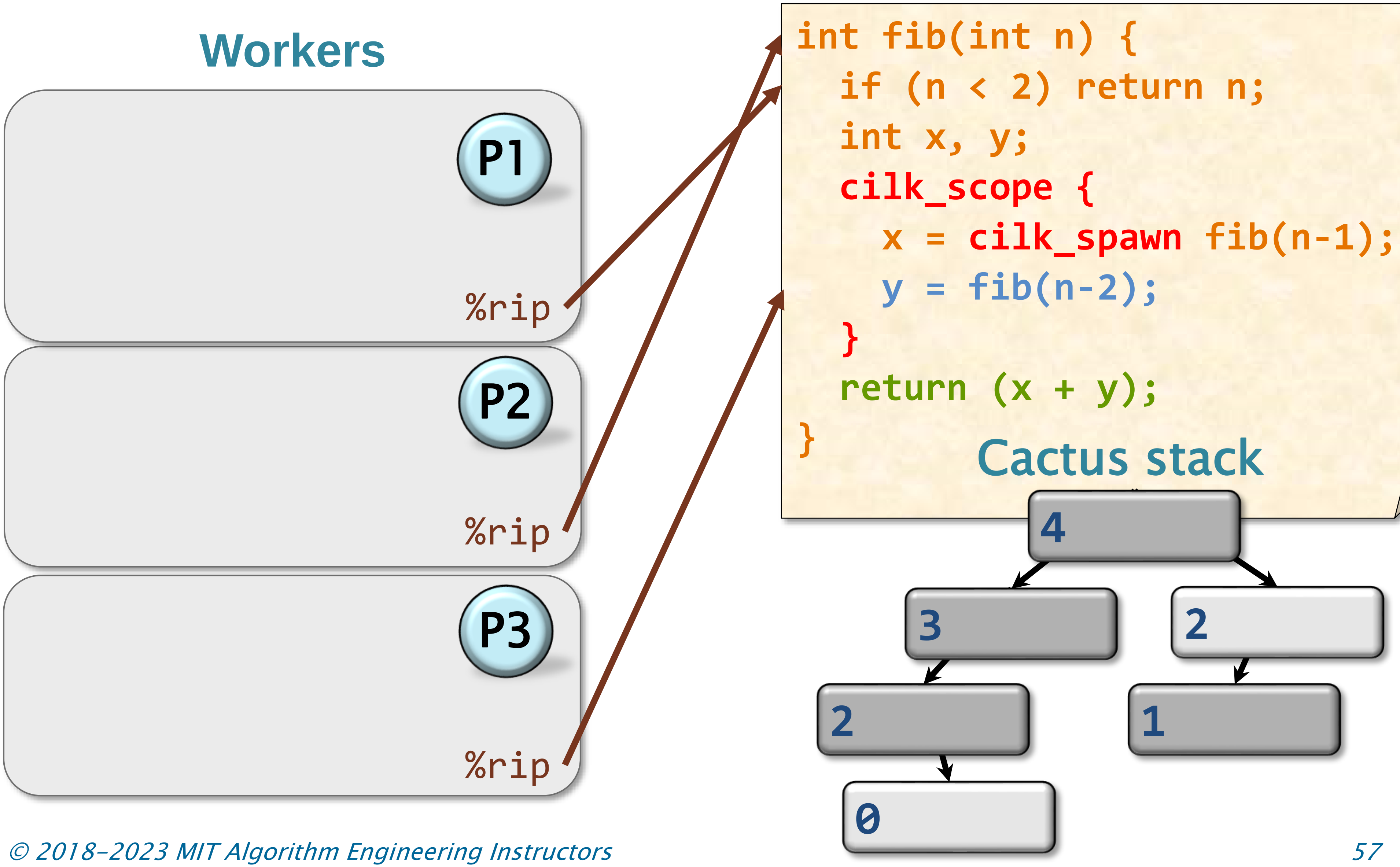


```
int fib(int n) {
  if (n < 2) return n;
  int x, y;
  cilk_scope {
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
  }
  return (x + y);
}
```

Cactus stack

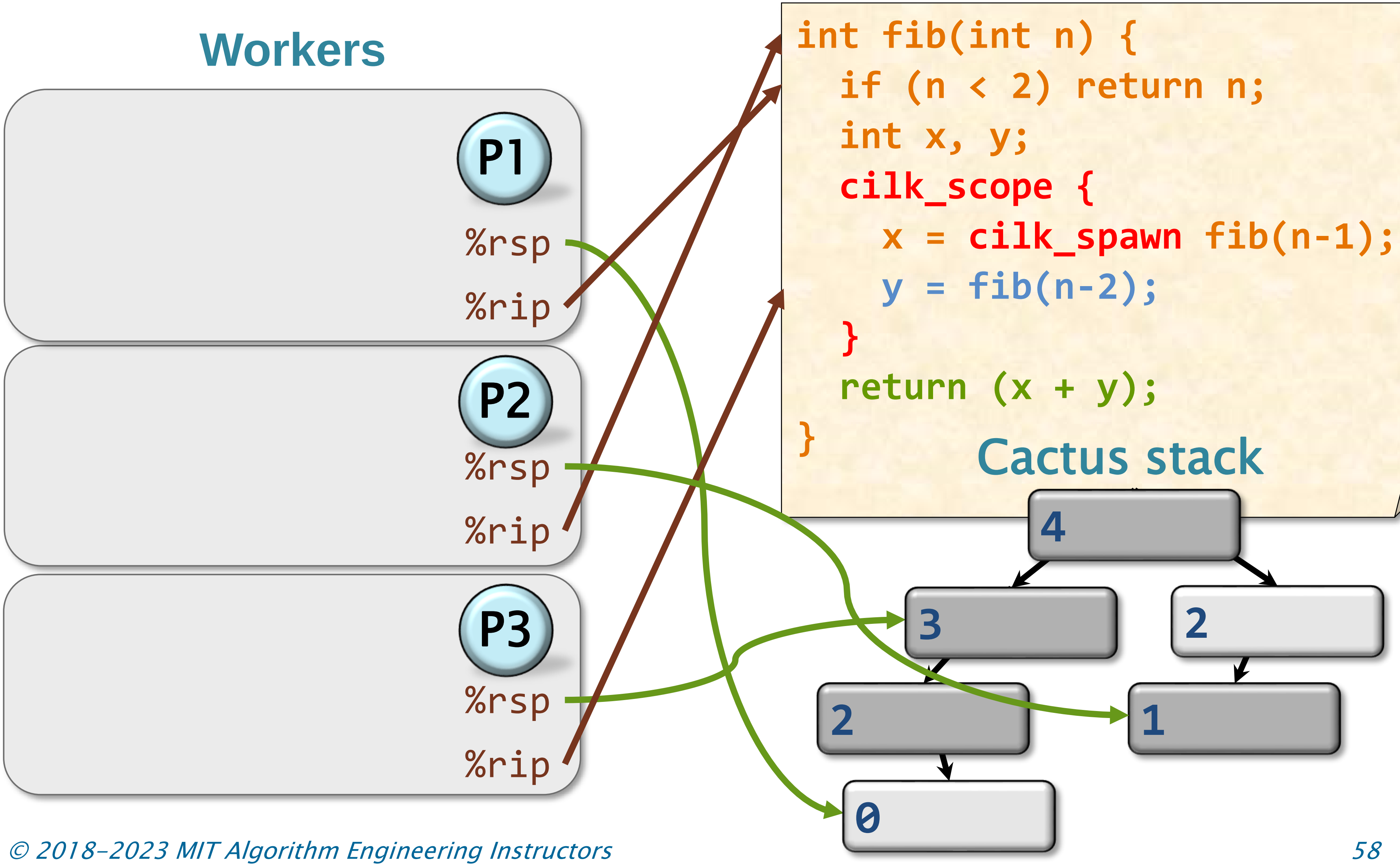


Putting Everything Together



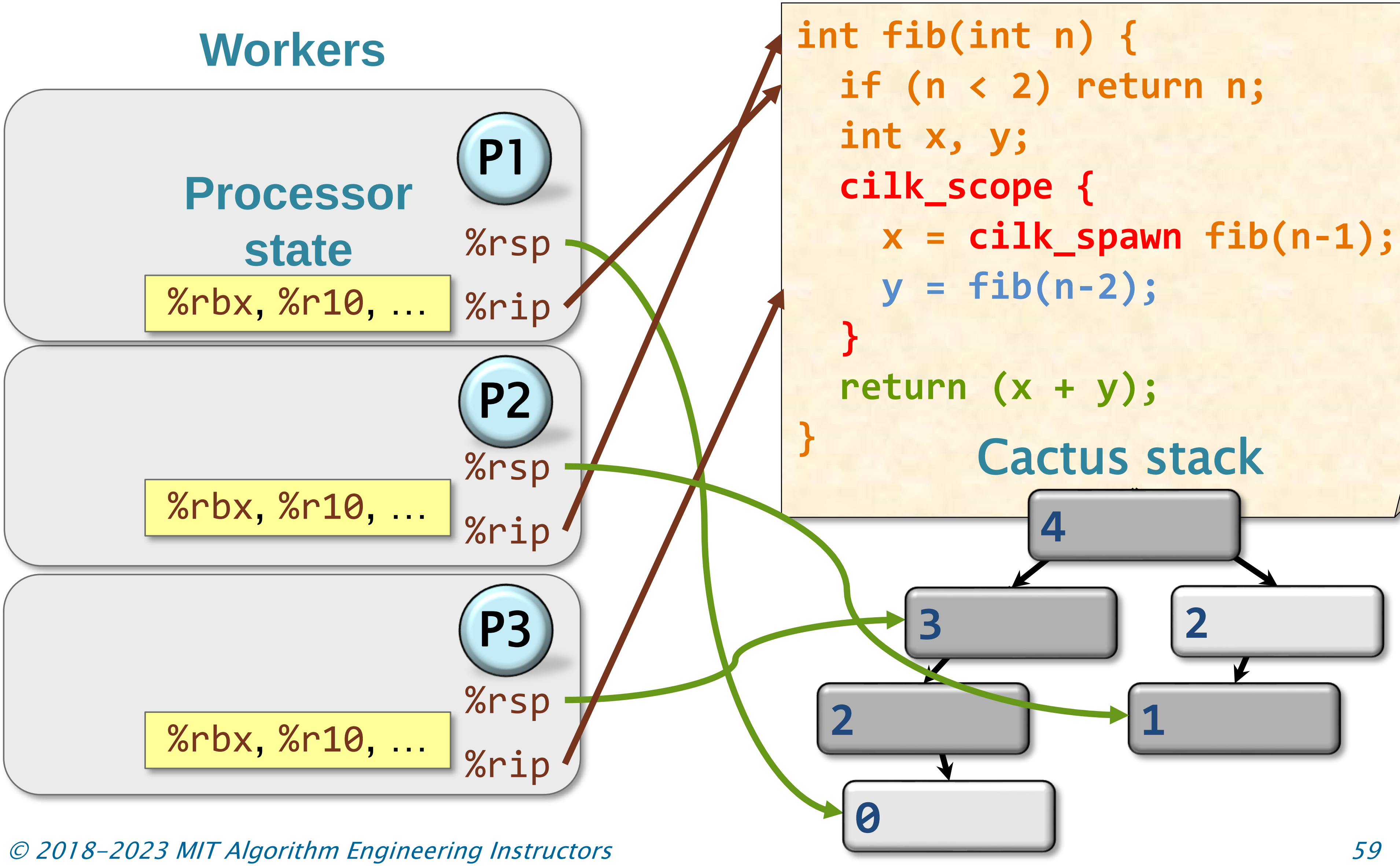
```
int fib(int n) {
  if (n < 2) return n;
  int x, y;
  cilk_scope {
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
  }
  return (x + y);
}
```

Putting Everything Together



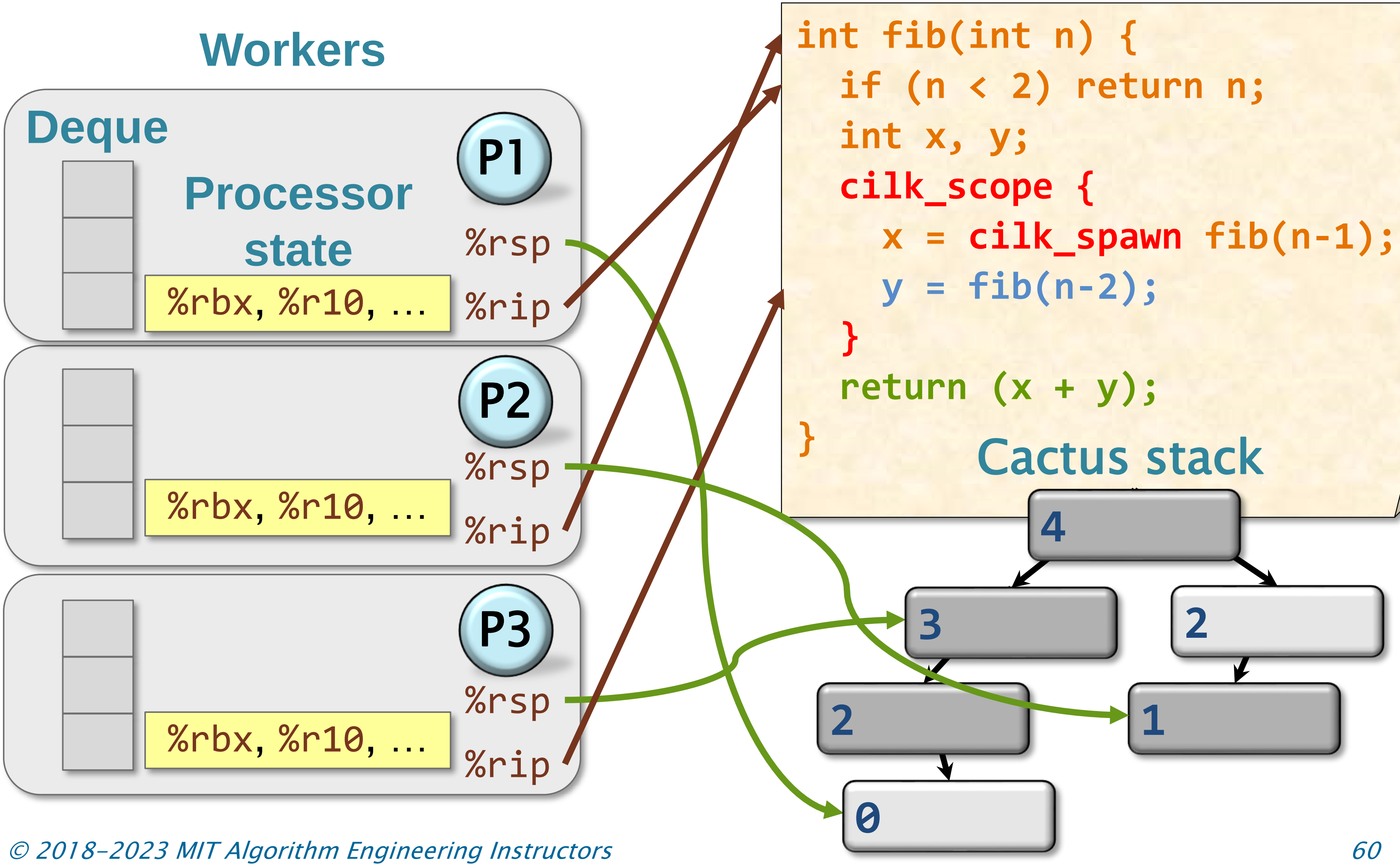
```
int fib(int n) {
  if (n < 2) return n;
  int x, y;
  cilk_scope {
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
  }
  return (x + y);
}
```

Putting Everything Together



```
int fib(int n) {
  if (n < 2) return n;
  int x, y;
  cilk_scope {
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
  }
  return (x + y);
}
```

Putting Everything Together



```
int fib(int n) {
  if (n < 2) return n;
  int x, y;
  cilk_scope {
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
  }
  return (x + y);
}
```

Required Functionalities

- Each worker needs to keep track of its own execution context, including work that it is responsible for / available to be stolen.
- After a successful steal, a worker can resume the stolen function mid-execution.
- Upon a sync, a worker needs to know whether there is any spawned subroutine still executing on another worker.

Cilk Runtime Data Structures

The Cilk runtime utilizes three basic data structures as workers execute work:

- *Worker deques* to keep track of subroutines which are being executed or available to steal.
- A *Cilk stack frame structure** to represent each spawning function (*Cilk* function) and store its execution context.
- A *full-frame tree* to represent function instances that have ever been stolen (to support true parallel execution).

*henceforth simply referred to as the frame

Division of Labor

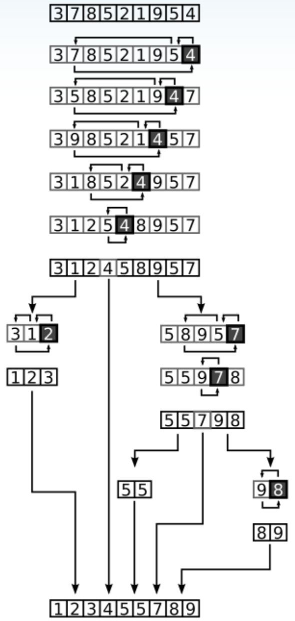
The work-first principle guides the division of the Cilk runtime between the compiler and the runtime library.

Compiler

- Manages a handful of light-weight data structures (e.g., Cilk stack frames and deques).
- Implements optimized fast paths for execution of functions when no steals have occurred (i.e., no actual parallelism has been realized).

Runtime library

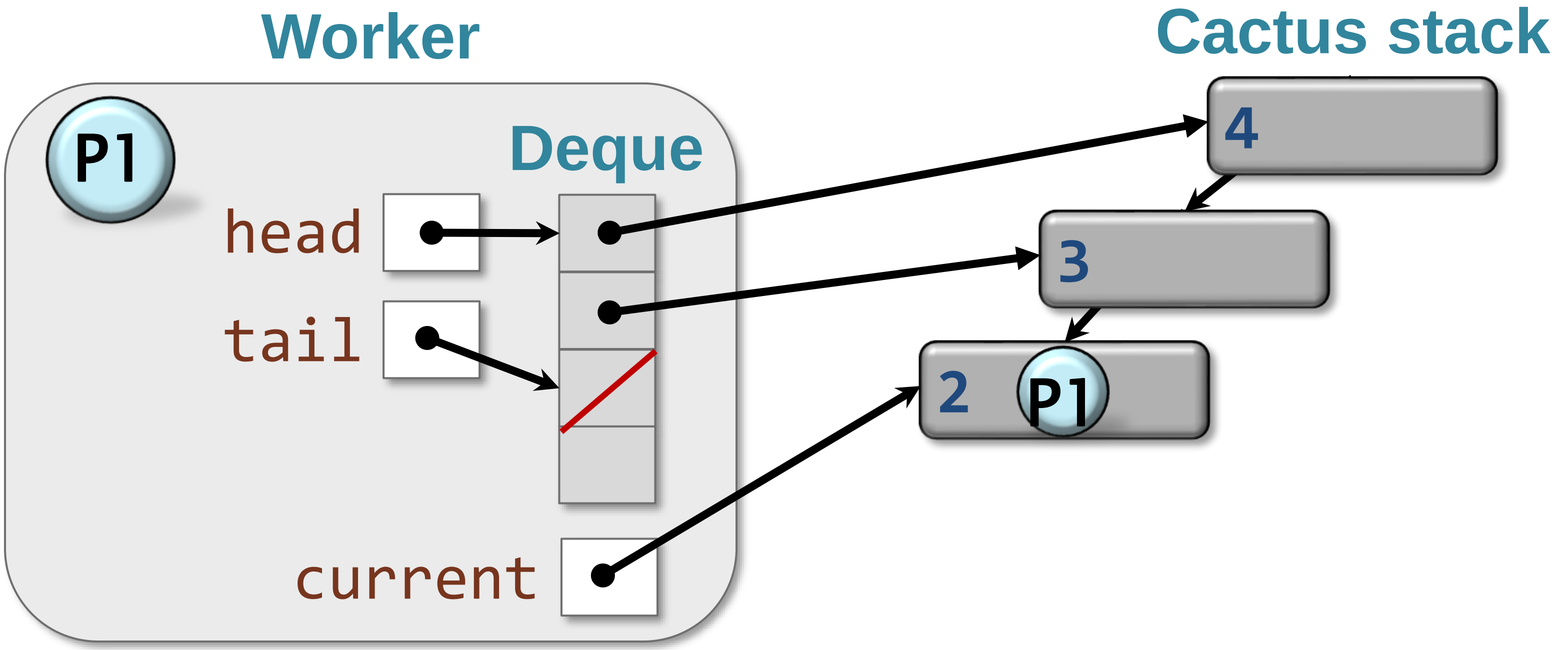
- Manages the more heavy-weight data structures (e.g., the full-frame tree).
- Handles slow paths of execution (e.g., when a steal occurs).



SPAWNS AND STEALS: DEQUES & CILK STACK FRAMES

Deque of Frames

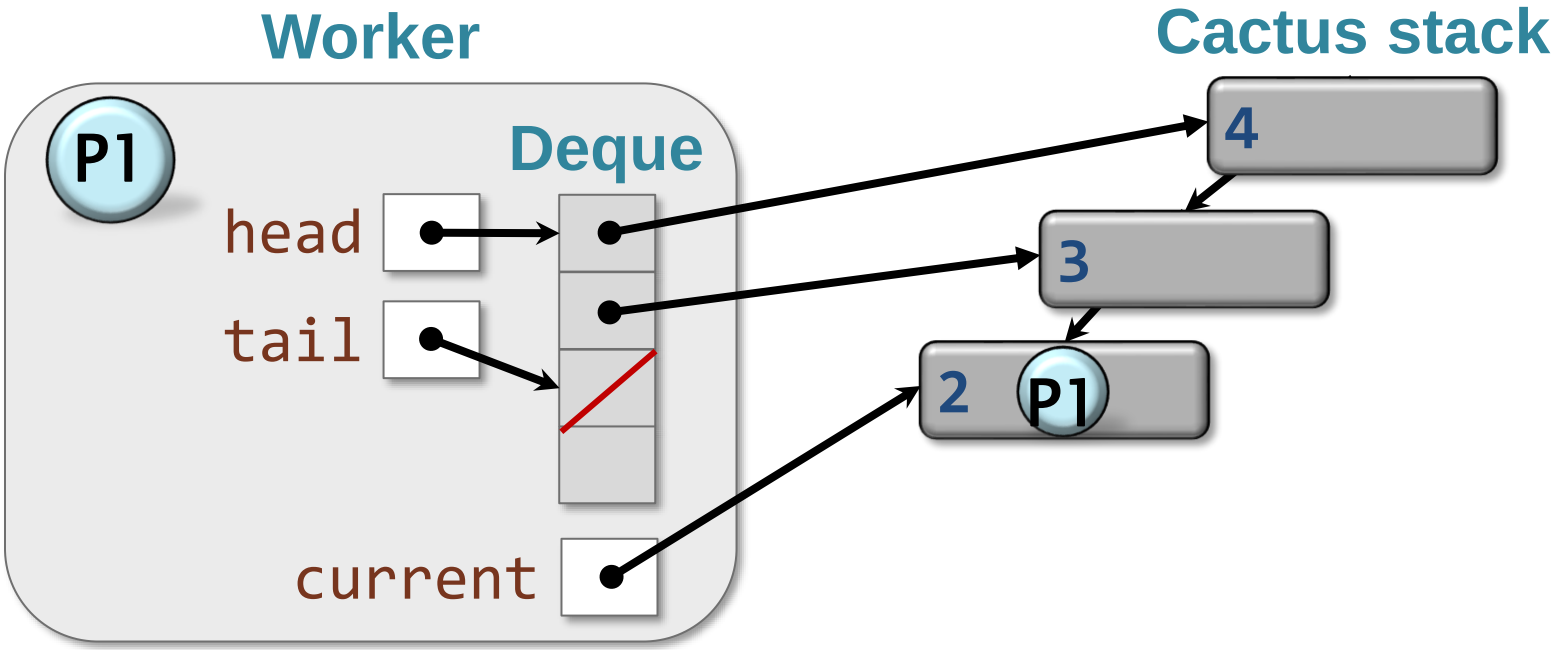
Each Cilk worker maintains a deque of references to Cilk Stack frames* containing work available to be stolen.



*We'll discuss what these references are in a few slides.

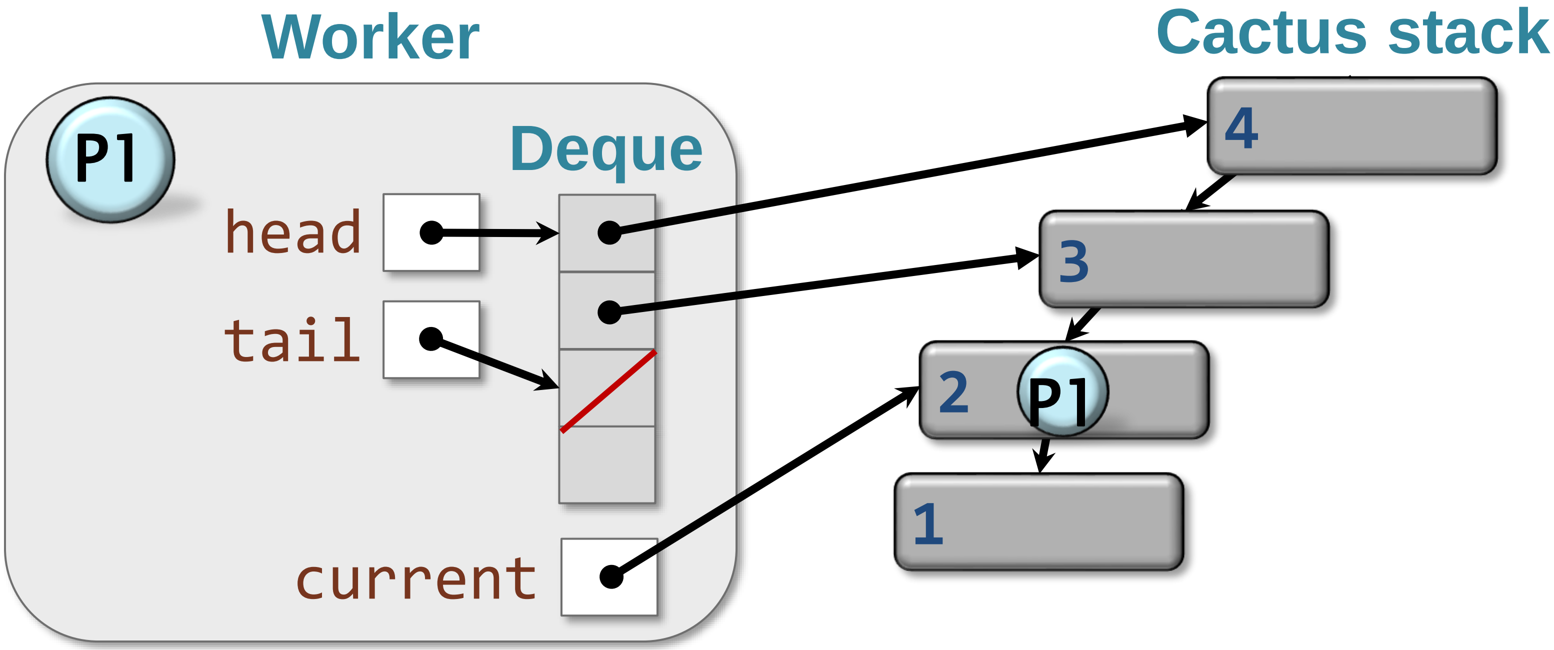
Spawn

When spawning, the current frame is pushed onto the bottom of the deque.



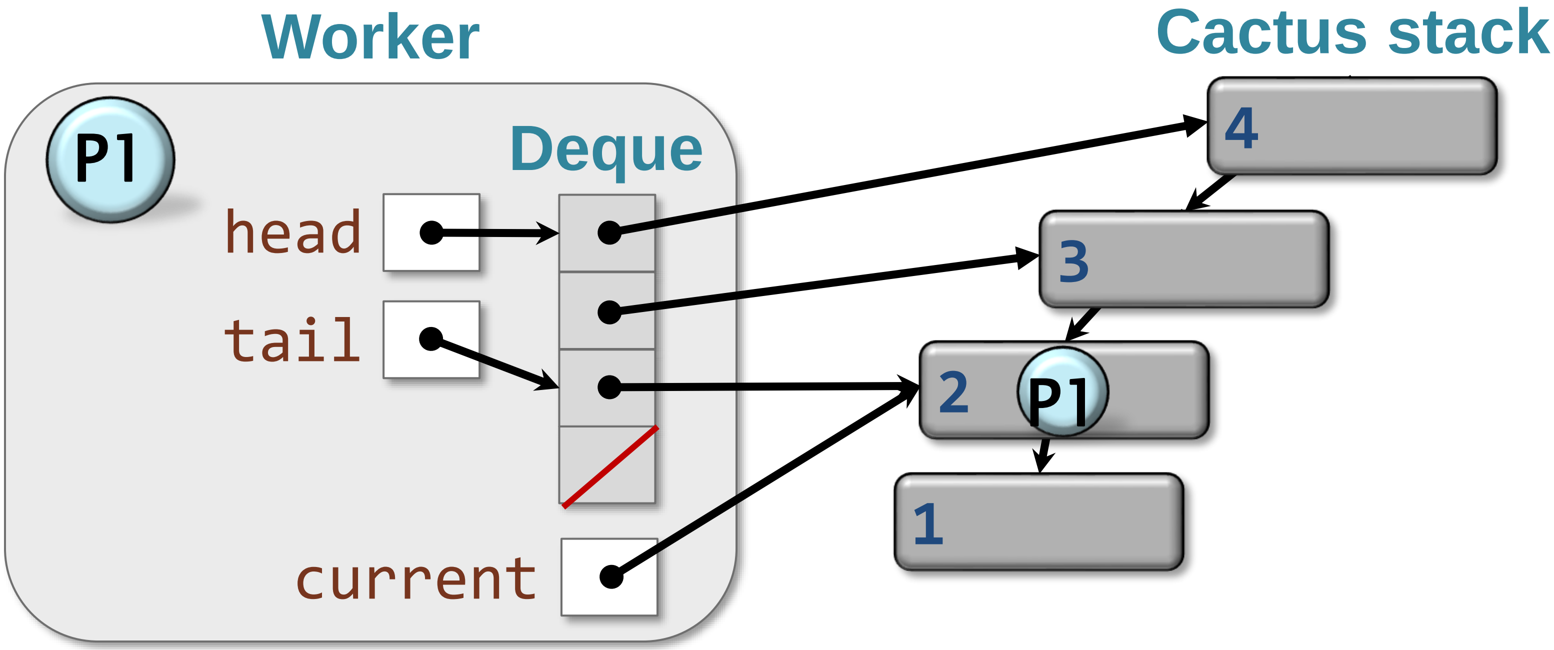
Spawn

When spawning, the current frame is pushed onto the bottom of the deque.



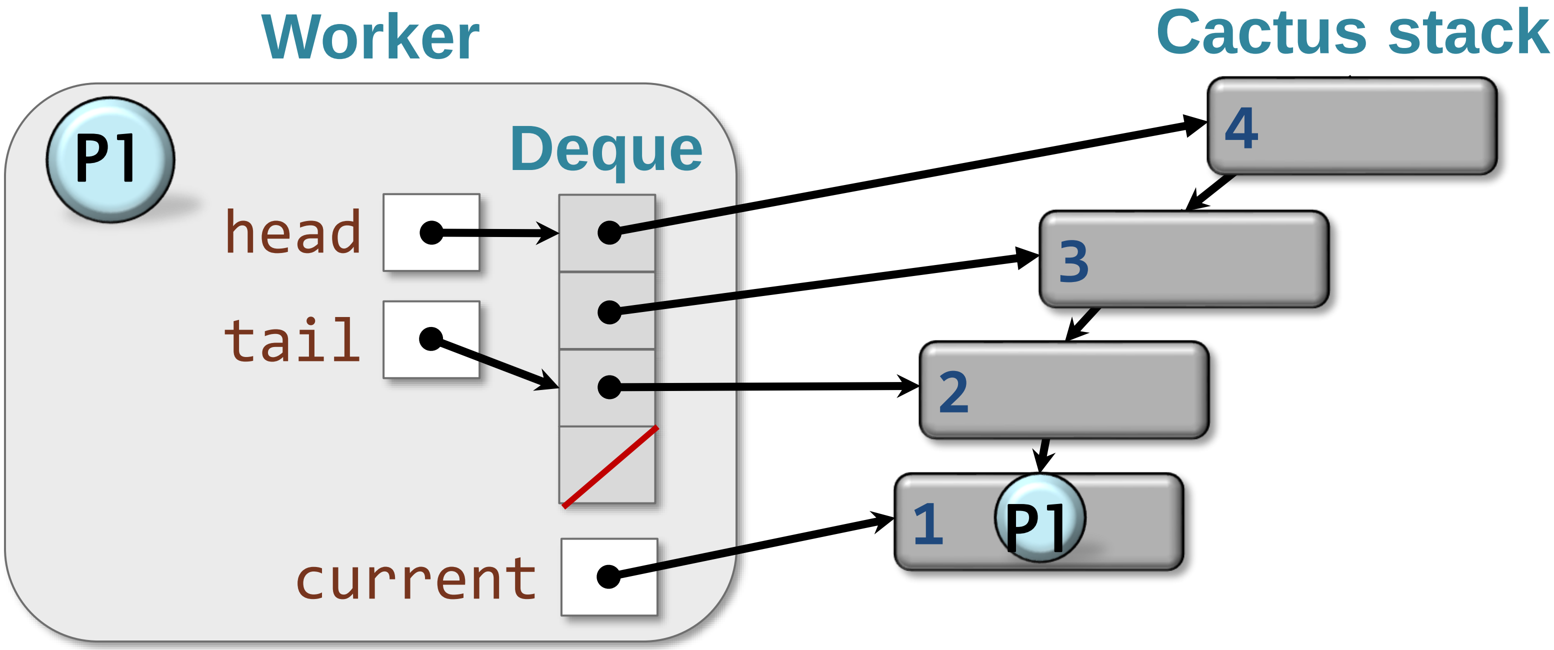
Spawn

When spawning, the current frame is pushed onto the bottom of the deque.



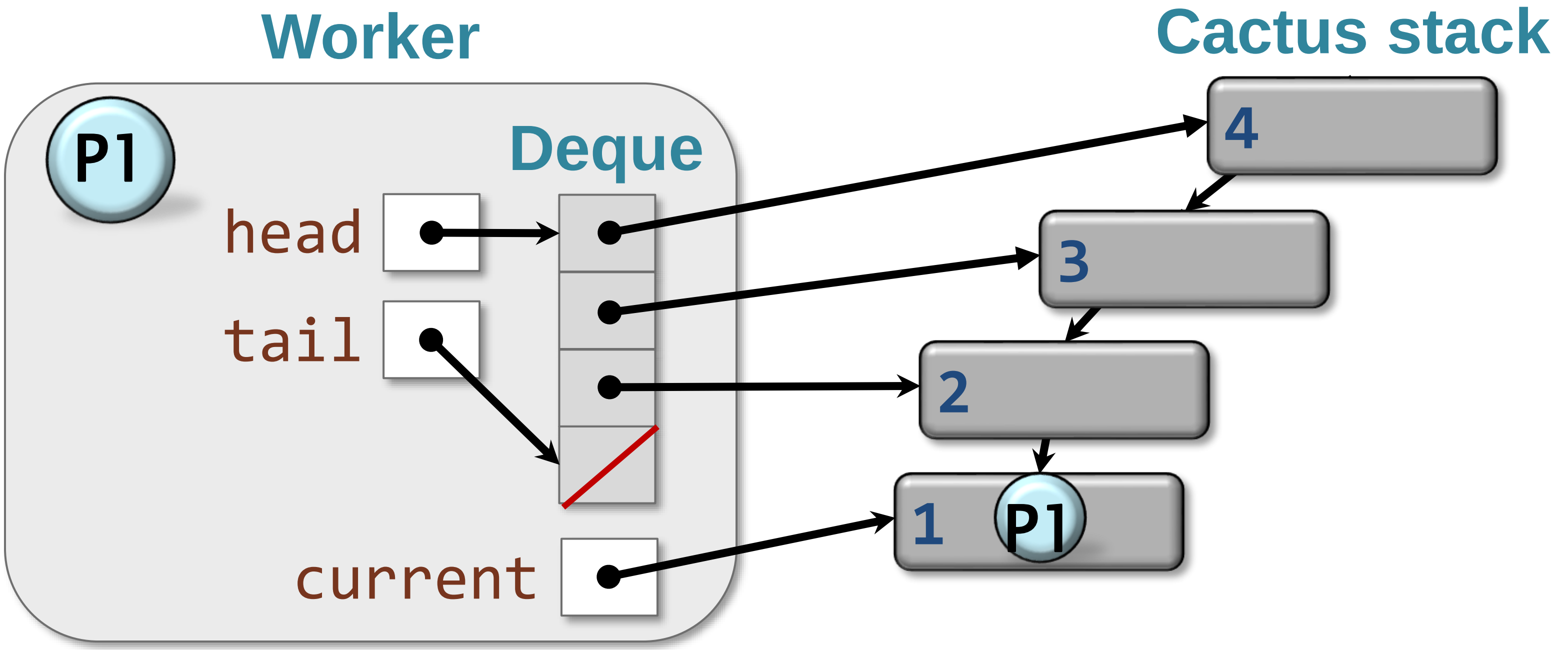
Spawn

When spawning, the current frame is pushed onto the bottom of the deque.



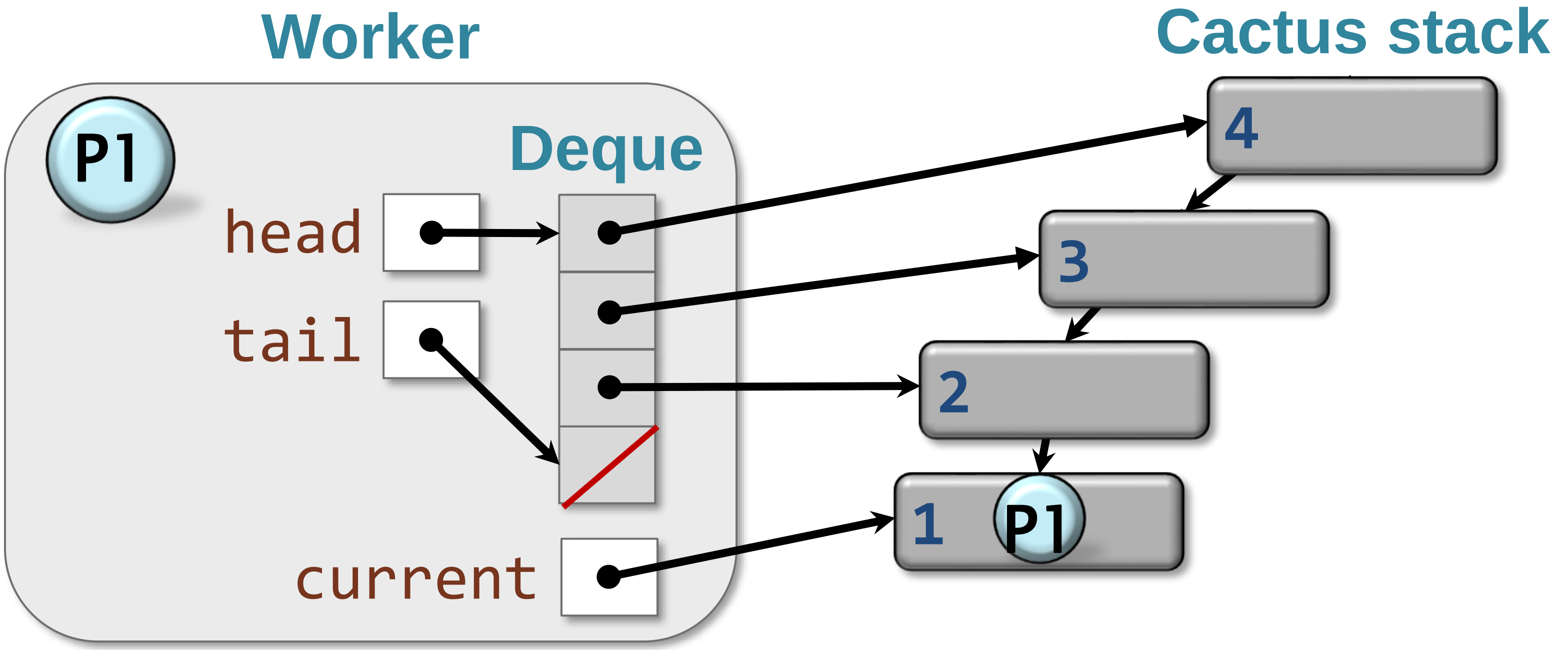
Spawn

When spawning, the current frame is pushed onto the bottom of the deque.



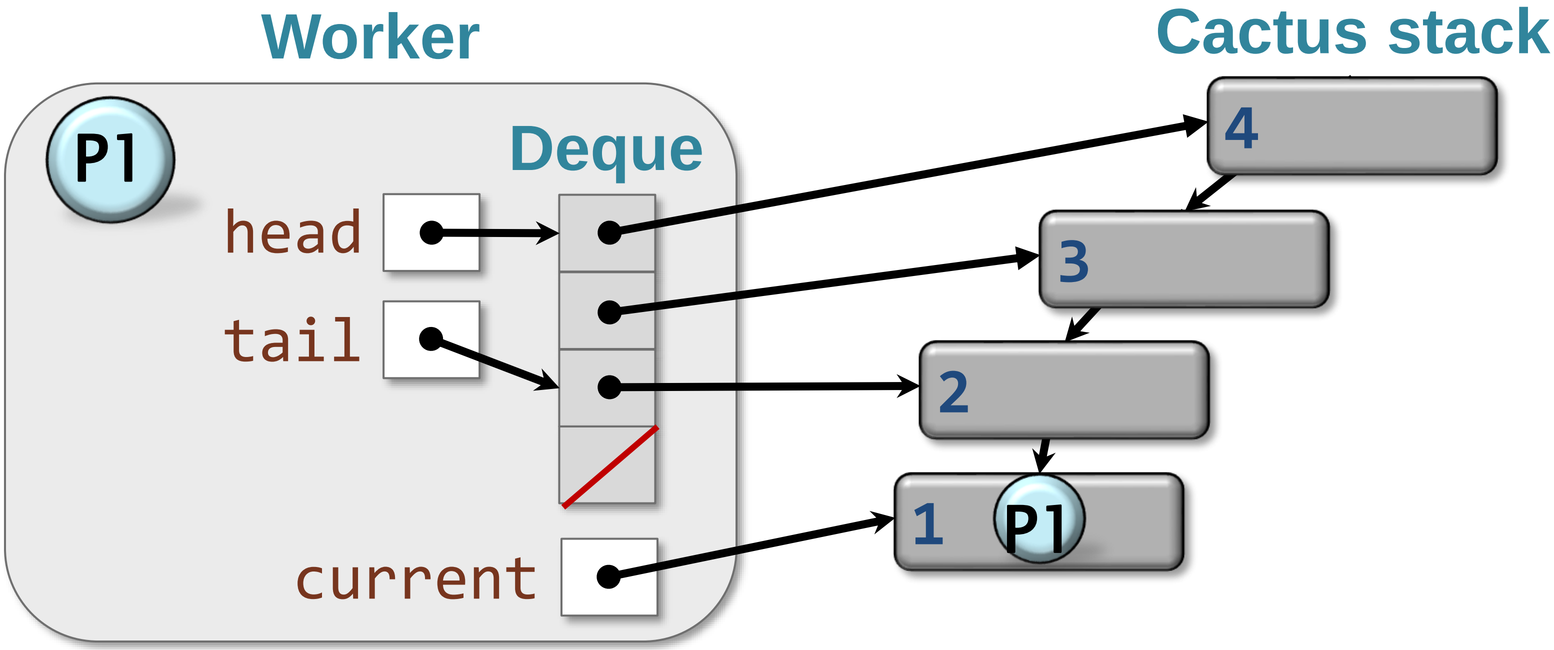
Return from Spawn

When returning from a spawn, the current frame is popped from the bottom of the deque.



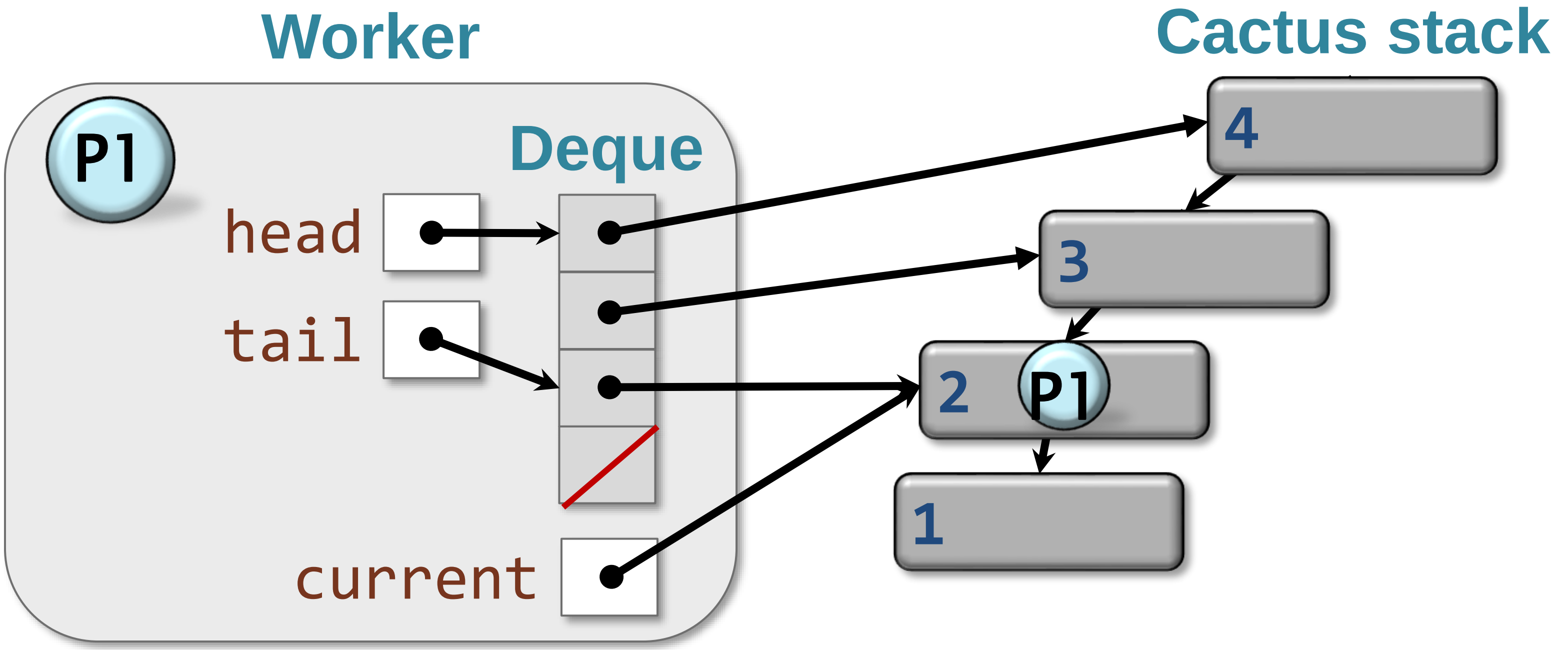
Return from Spawn

When returning from a spawn, the current frame is popped from the bottom of the deque.



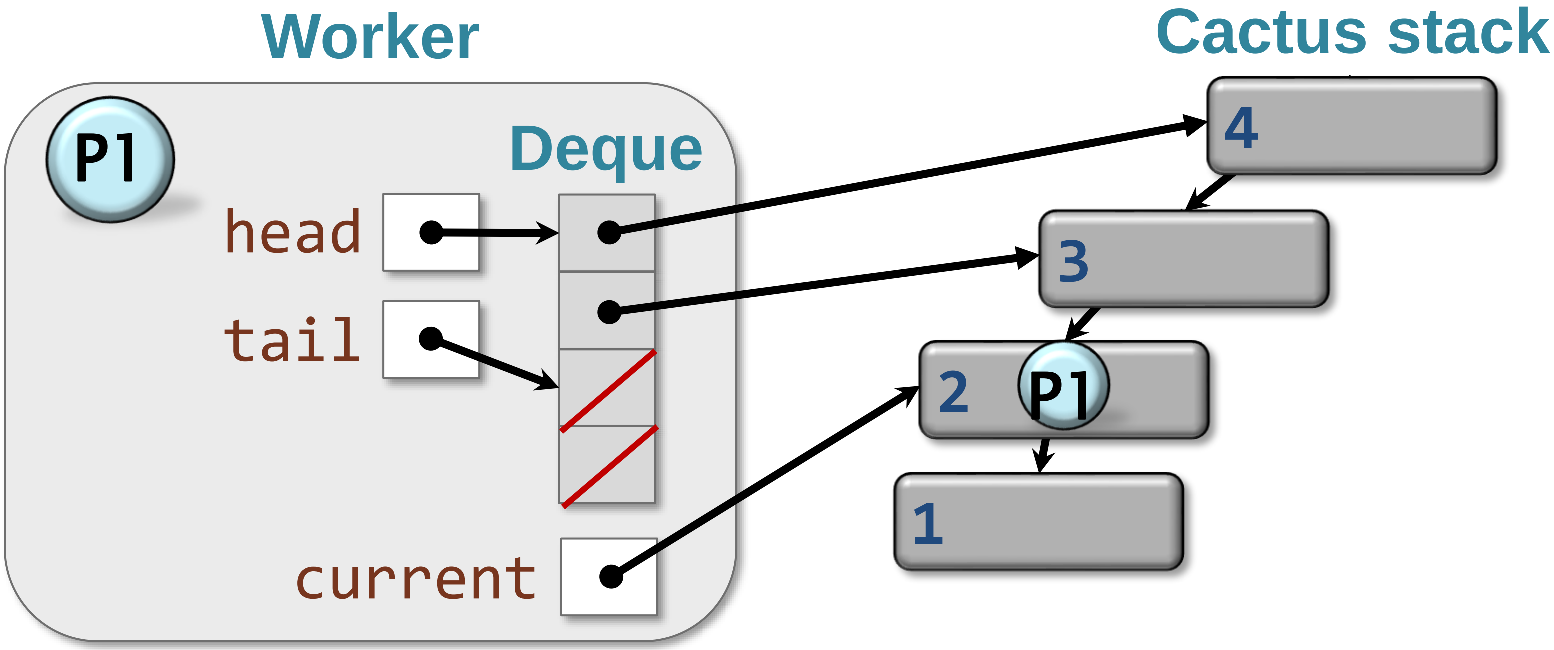
Return from Spawn

When returning from a spawn, the current frame is popped from the bottom of the deque.



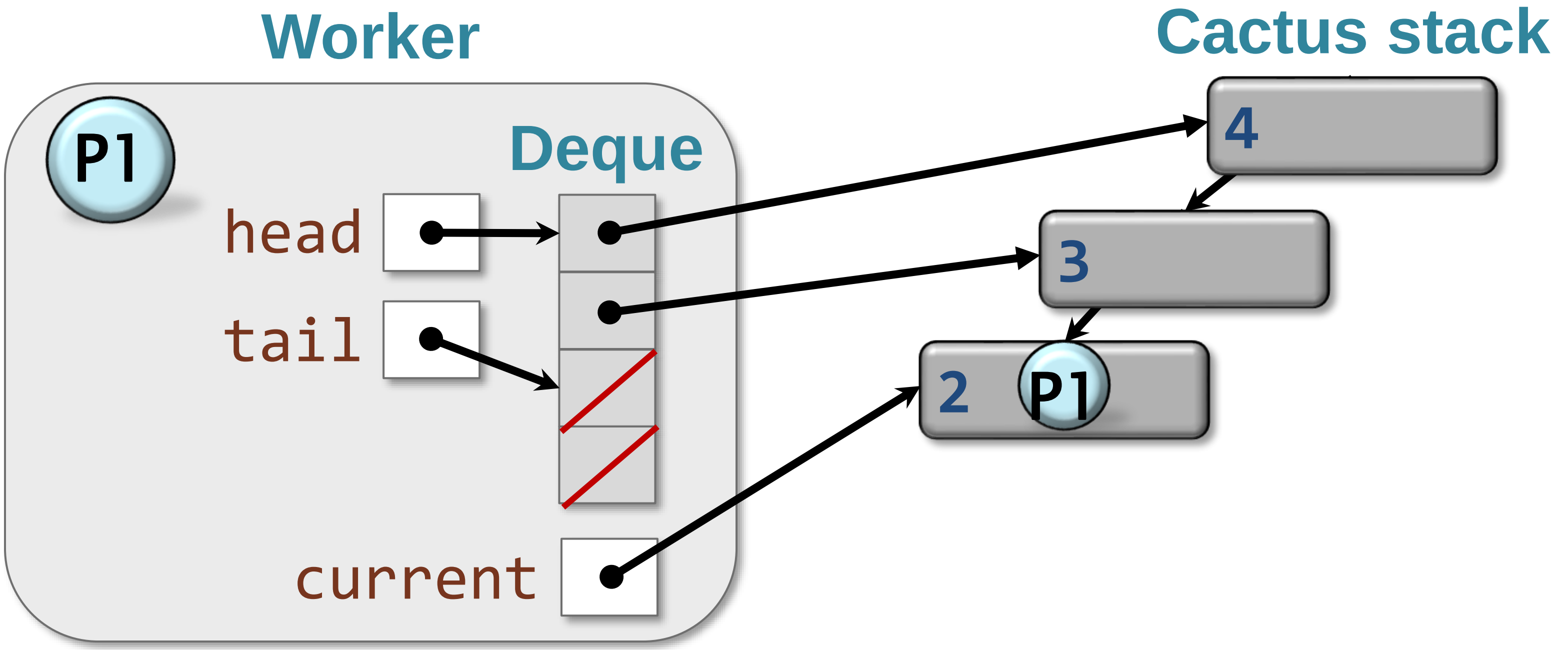
Return from Spawn

When returning from a spawn, the current frame is popped from the bottom of the deque.



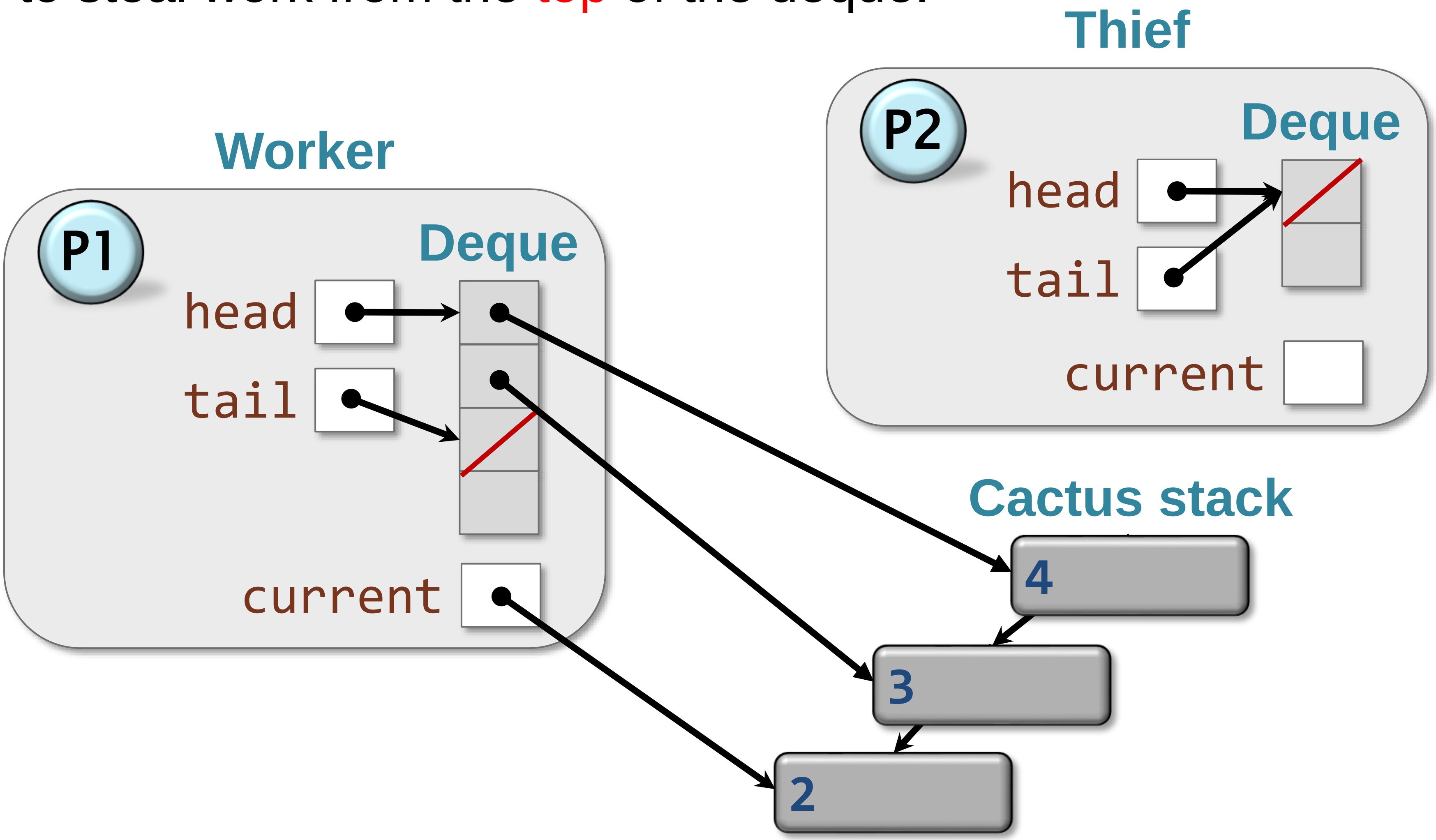
Return from Spawn

When returning from a spawn, the current frame is popped from the bottom of the deque.



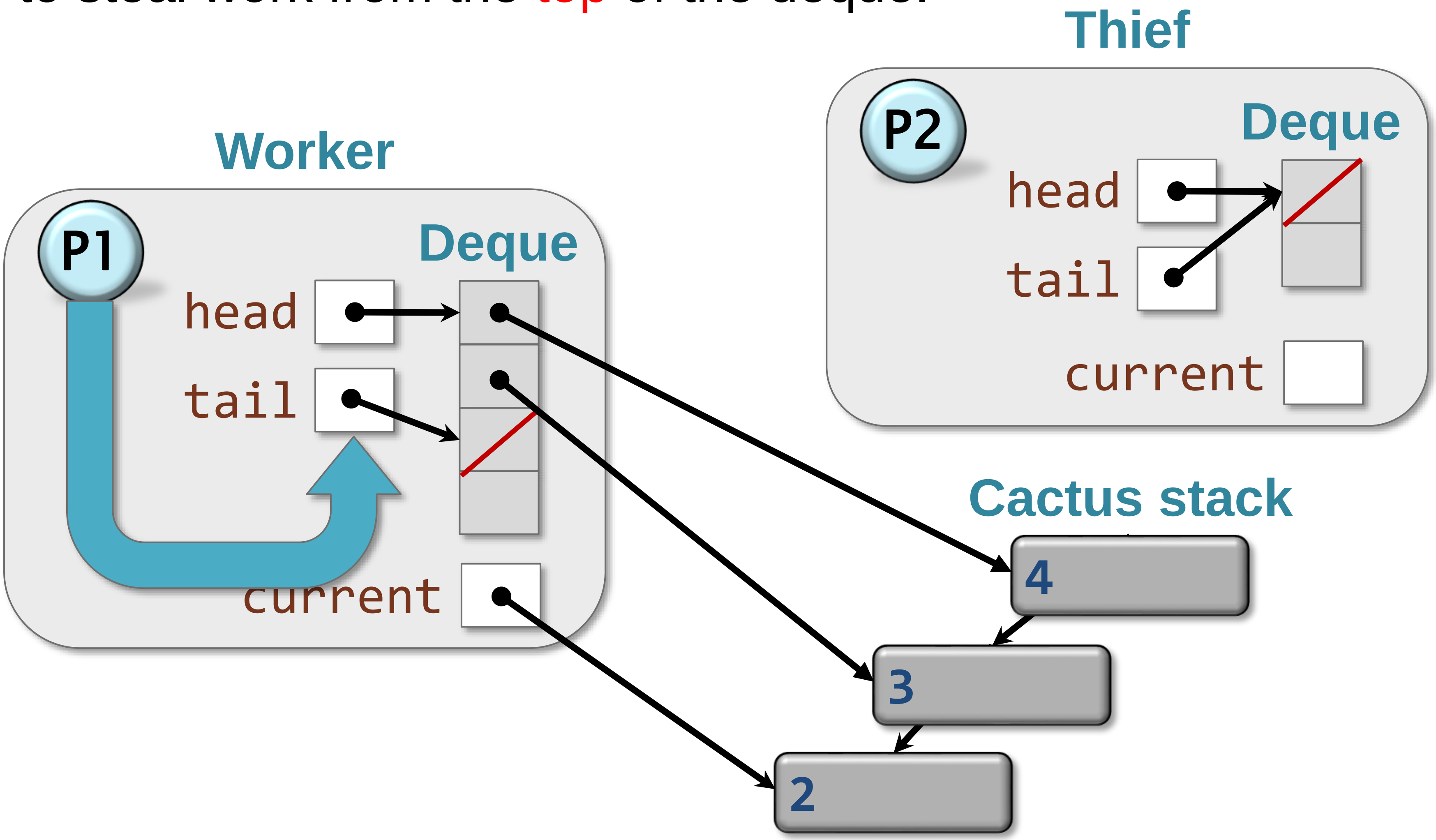
Stealing Frames

Workers operate on the bottom of the deque, while thieves try to steal work from the top of the deque.



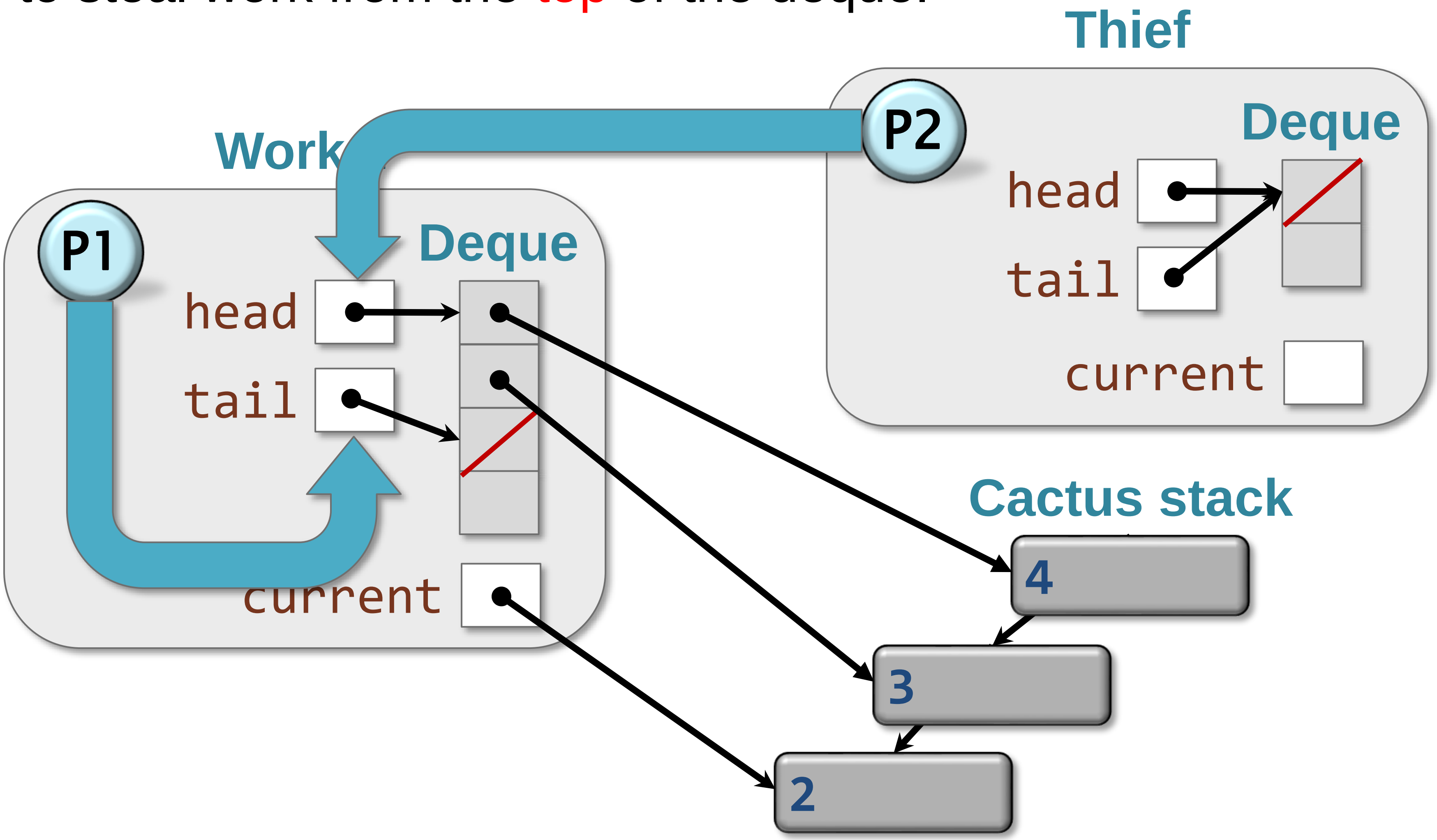
Stealing Frames

Workers operate on the bottom of the deque, while thieves try to steal work from the top of the deque.



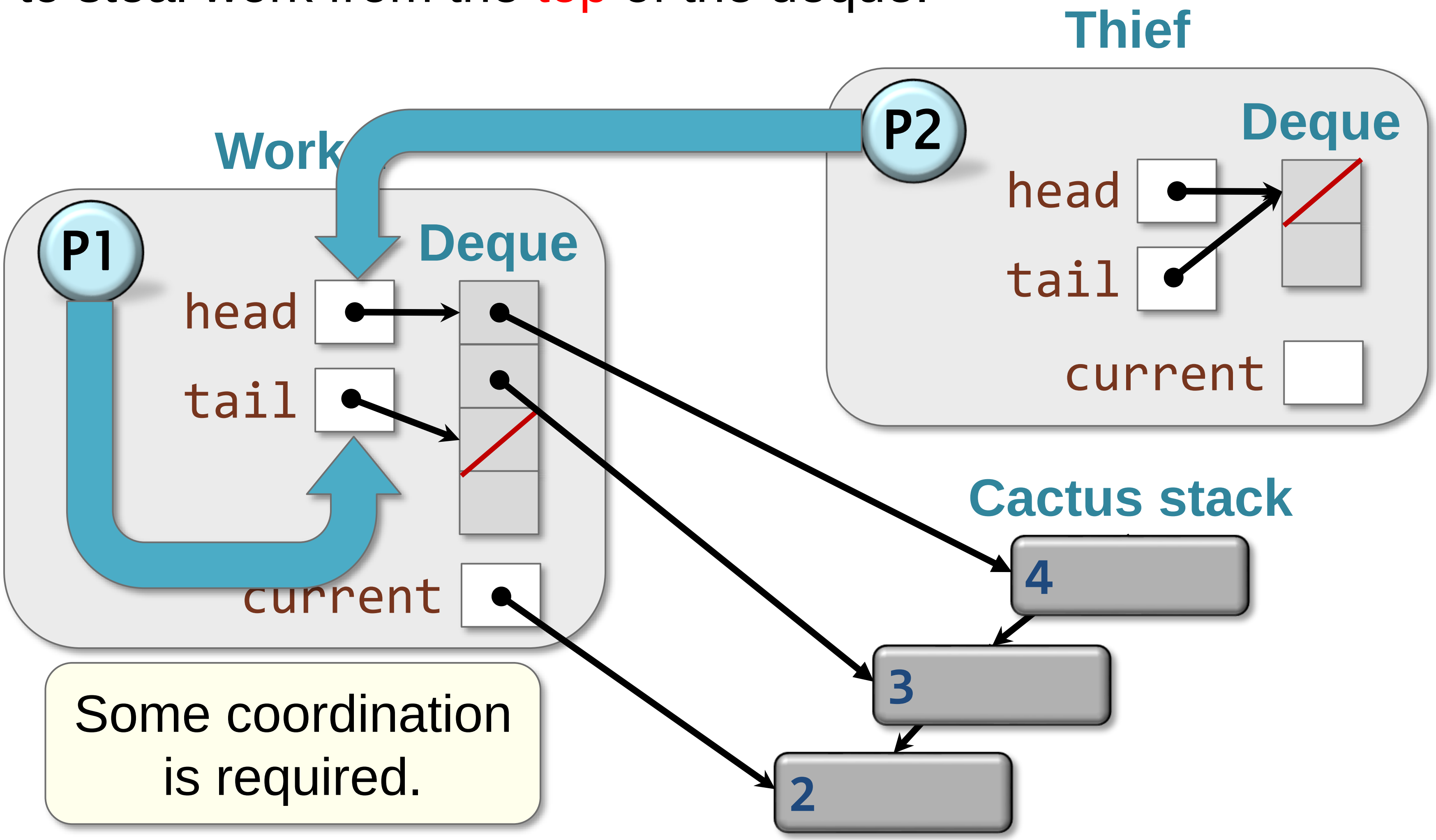
Stealing Frames

Workers operate on the bottom of the deque, while thieves try to steal work from the top of the deque.



Stealing Frames

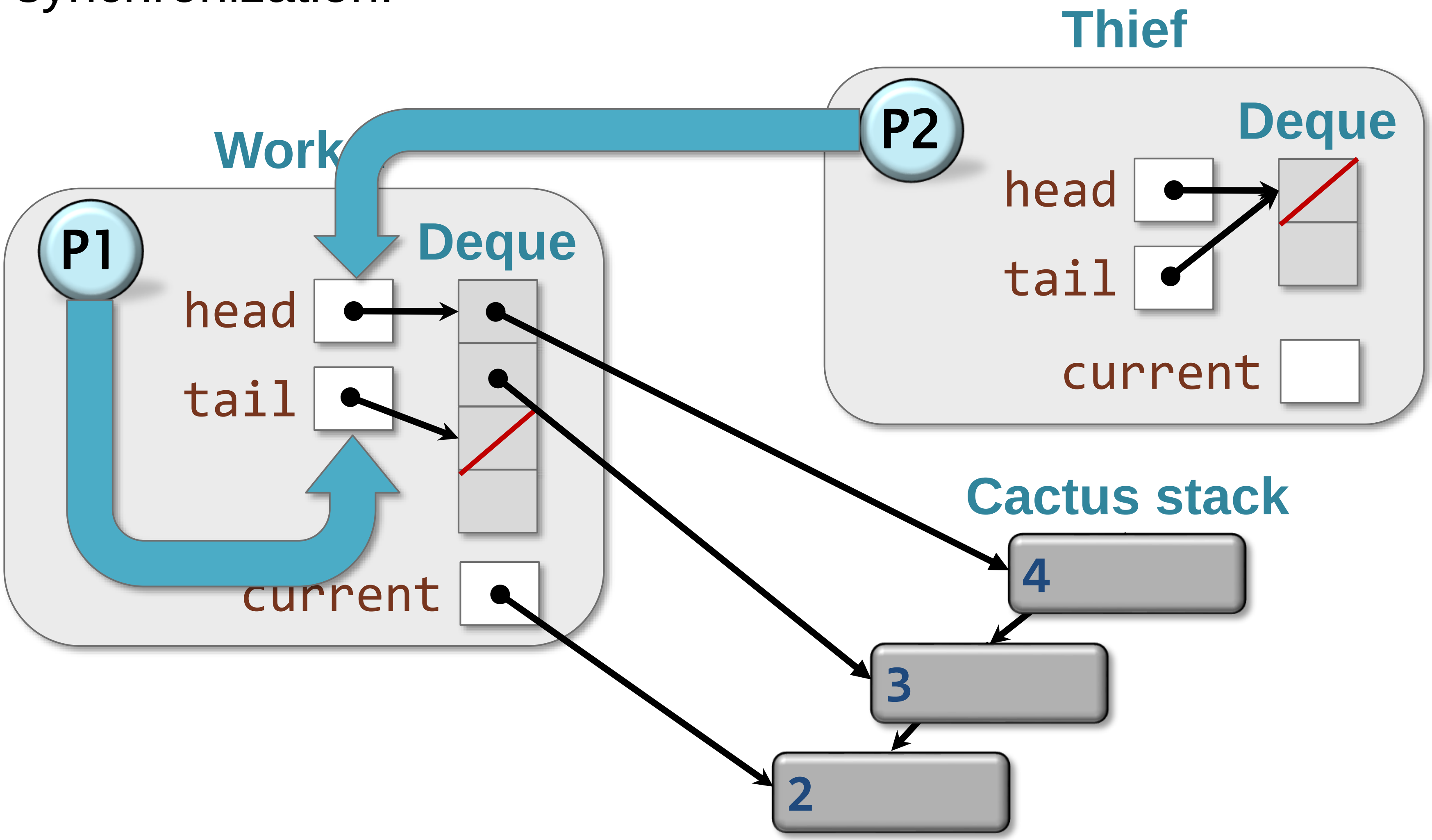
Workers operate on the bottom of the deque, while thieves try to steal work from the top of the deque.



Some coordination is required.

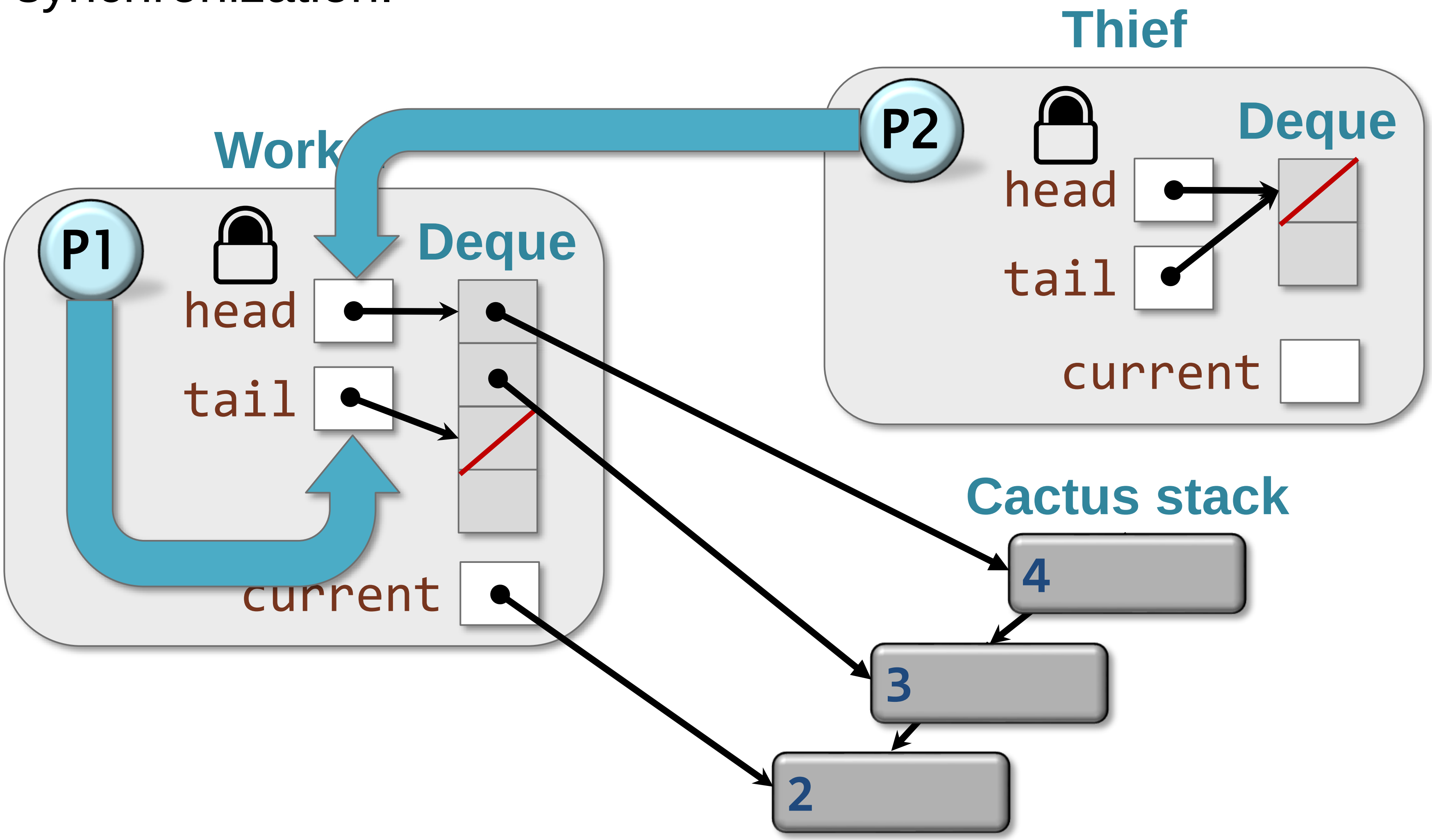
Synchronizing Thieves and Workers

Cilk uses a mutex associated with each deque to perform synchronization.



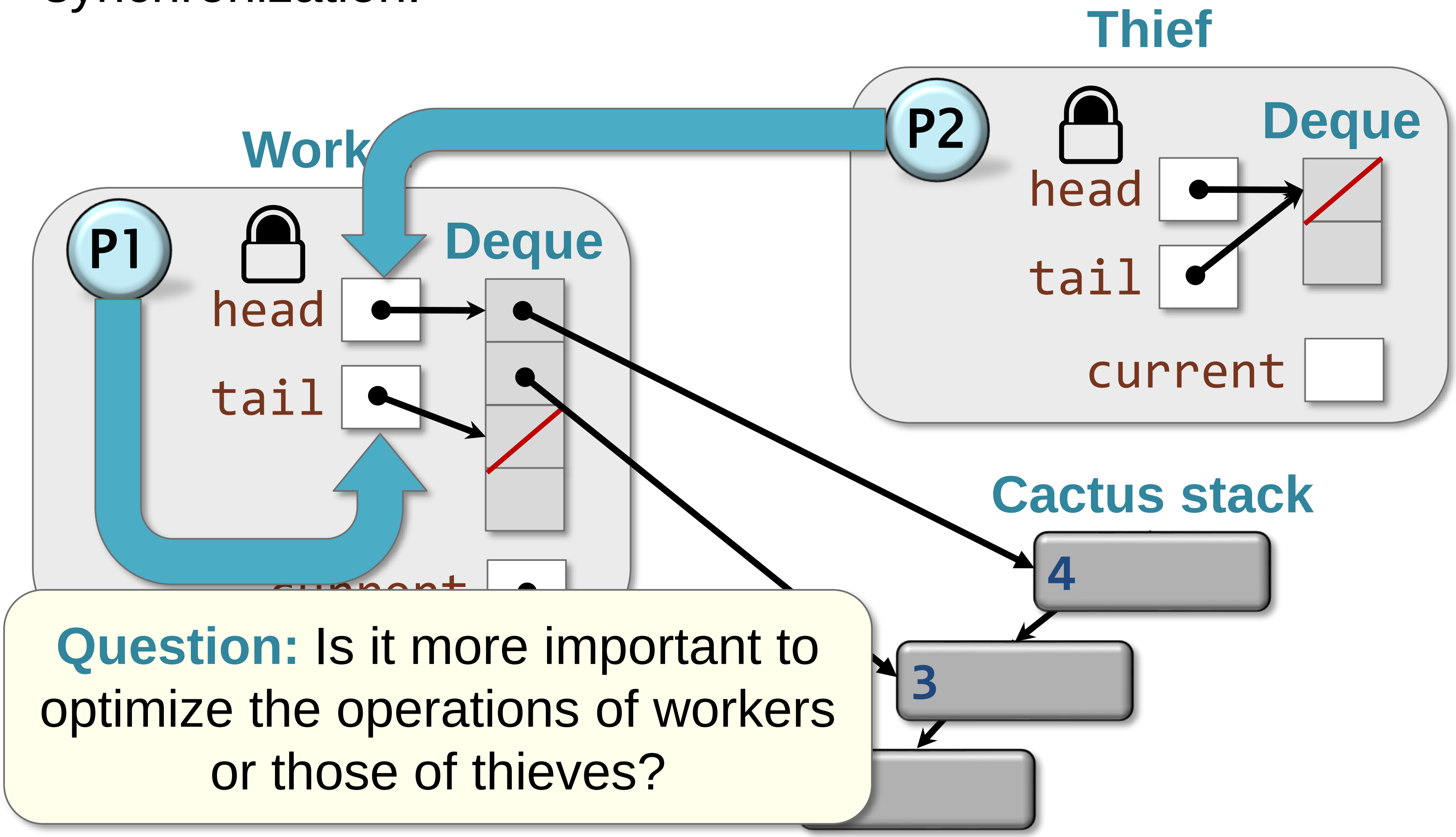
Synchronizing Thieves and Workers

Cilk uses a mutex associated with each deque to perform synchronization.



Synchronizing Thieves and Workers

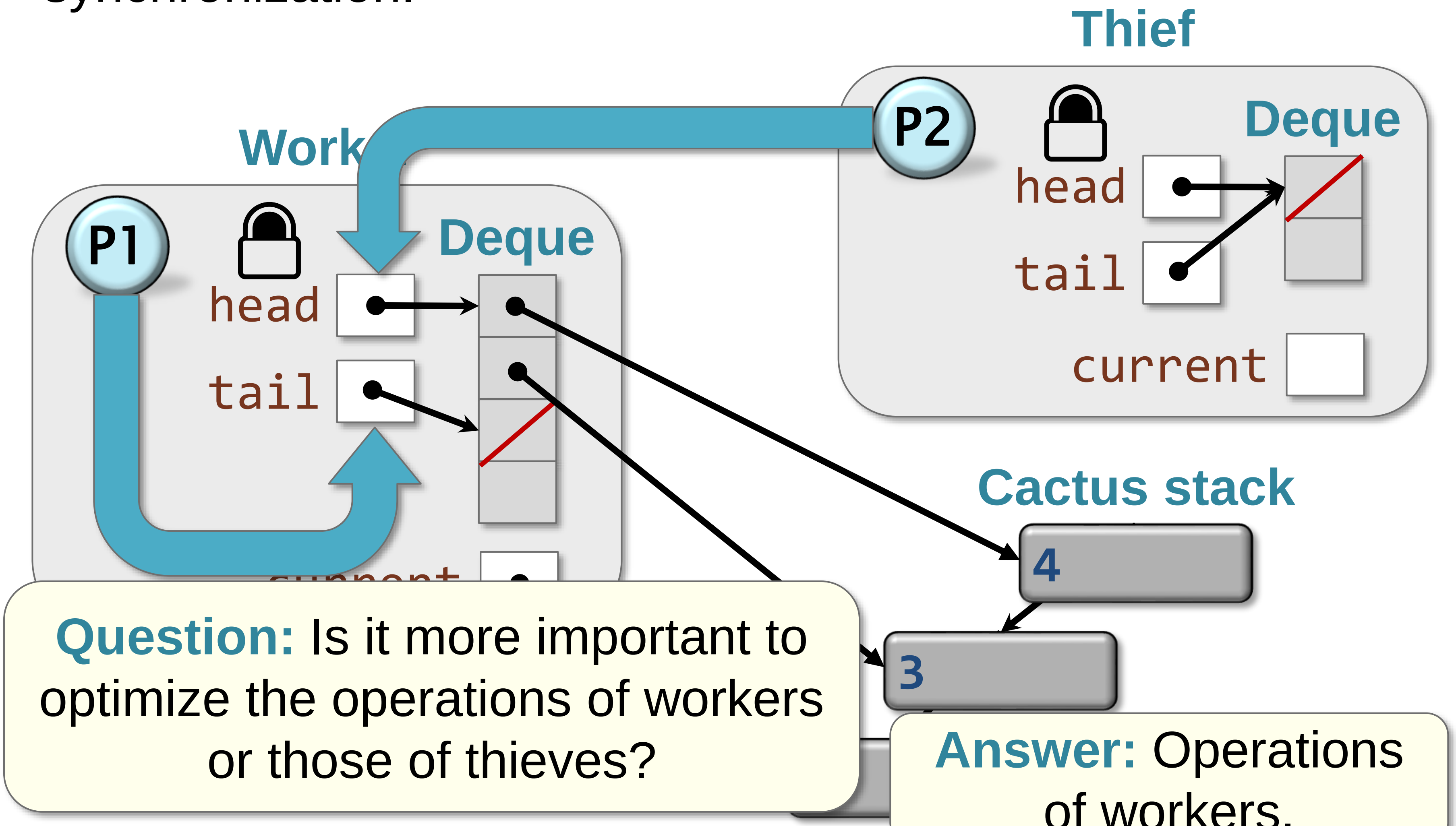
Cilk uses a mutex associated with each deque to perform synchronization.



Question: Is it more important to optimize the operations of workers or those of thieves?

Synchronizing Thieves and Workers

Cilk uses a mutex associated with each deque to perform synchronization.



Question: Is it more important to optimize the operations of workers or those of thieves?

Answer: Operations of workers.

Popping the Deque

When a worker is about to return from a spawned function, it tries to to pop the stack frame from the tail of the deque. There are two possible outcomes:

1. If the pop succeeds, then the execution continues as normal.
2. If the pop fails, then the worker is out of work to do, and it becomes a thief and tries to steal.

Popping the Deque

When a worker is about to return from a spawned function, it tries to to pop the stack frame from the tail of the deque. There are two possible outcomes:

1. If the pop succeeds, then the execution continues as normal.
2. If the pop fails, then the worker is out of work to do, and it becomes a thief and tries to steal.

Question: Which case is more important to optimize?

Popping the Deque

When a worker is about to return from a spawned function, it tries to to pop the stack frame from the tail of the deque. There are two possible outcomes:

1. If the pop succeeds, then the execution continues as normal.
2. If the pop fails, then the worker is out of work to do, and it becomes a thief and tries to steal.

Question: Which case is more important to optimize?

Answer: Case 1, successful pop.

The THE Protocol

Worker protocol

```
void push() { tail++; }

bool pop() {
  tail--;
  if (head > tail) {
    tail++;
    lock(L);
    tail--;
    if (head > tail) {
      tail++;
      unlock(L);
      return FAILURE;
    }
    unlock(L);
  }
  return SUCCESS;
}
```

The worker and the thief coordinate using *the THE protocol*

Thief protocol

```
bool steal() {
  lock(L);
  head++;
  if (head > tail) {
    head--;
    unlock(L);
    return FAILURE;
  }
  unlock(L);
  return SUCCESS;
}
```

The THE Protocol

Worker protocol

```
void push() { tail++; }

bool pop() {
  tail--;
  if (head > tail) {
    tail++;
    lock(L);
    tail--;
    if (head > tail) {
      tail++;
      unlock(L);
      return FAILURE;
    }
    unlock(L);
  }
  return SUCCESS;
}
```

Observation I: Synchronization is only necessary when the deque is almost empty.

Thief protocol

```
bool steal() {
  lock(L);
  head++;
  if (head > tail) {
    head--;
    unlock(L);
    return FAILURE;
  }
  unlock(L);
  return SUCCESS;
}
```

The THE Protocol

Worker protocol

```
void push() { tail++; }

bool pop() {
  tail--;
  if (head > tail) {
    tail++;
    lock(L);
    tail--;
    if (head > tail) {
      tail++;
      unlock(L);
      return FAILURE;
    }
    unlock(L);
  }
  return SUCCESS;
}
```

Observation II: The pop operation is more likely to succeed than fail.

Thief protocol

```
bool steal() {
  lock(L);
  head++;
  if (head > tail) {
    head--;
    unlock(L);
    return FAILURE;
  }
  unlock(L);
  return SUCCESS;
}
```

The THE Protocol

Worker protocol

```
void push() { tail++; }

bool pop() {
  tail--;
  if (head > tail) {
    tail++;
    lock(L);
    tail--;
    if (head > tail) {
      tail++;
      unlock(L);
      return FAILURE;
    }
    unlock(L);
  }
  return SUCCESS;
}
```

The Work-First Principle: Optimize the operations of workers.

Thief protocol

```
bool steal() {
  lock(L);
  head++;
  if (head > tail) {
    head--;
    unlock(L);
    return FAILURE;
  }
  unlock(L);
  return SUCCESS;
}
```

40.2

The THE Protocol

Worker protocol

```
void push() {

bool pop() {
  tail--;
  if (head > tail) {
    tail++;
    lock(L);
    tail--;
    if (head > tail) {
      tail++;
      unlock(L);
      return FAILURE;
    }
    unlock(L);
  }
  return SUCCESS;
}
```

Workers pop the deque optimistically…

The Work-First
Optimize the
ns of workers.

Thief protocol

```
bool steal() {
  lock(L);
  head++;
  if (head > tail) {
    head--;
    unlock(L);
    return FAILURE;
  }
  unlock(L);
  return SUCCESS;
}
```

The THE Protocol

Worker protocol

```
void push() {

bool pop() {
  tail--;
  if (head > tail) {
    tail++;
    lock(L);
    tail--;
    if (head > tail) {
      tail++;
      unlock(L);
      return FAILURE;
    }
    unlock(L);
  }
  return SUCCESS;
}
```

Workers pop the deque optimistically…

The Work-First

Optimize the

ns of workers.

Thief protocol

```
bool steal() {
  lock(L);
  head++;
  if (head > tail) {
    head--;
    unlock(L);
    return FAILURE;
  }
    lock(L);
```

…and only grab the deque’s lock if the deque appears to be empty.

40.4

The THE Protocol

Worker protocol

```
void push() {

bool pop() {
  tail--;
  if (head > tail) {
    tail++;
    lock(L);
    tail--;
    if (head > tail) {
      tail++;
      unlock(L);
      return FAILURE;
    }
    unlock(L);
  }
  return SUCCESS;
}
```

Workers pop the deque optimistically…

The Work-First

Optimize the

ns of workers.

Thief protocol

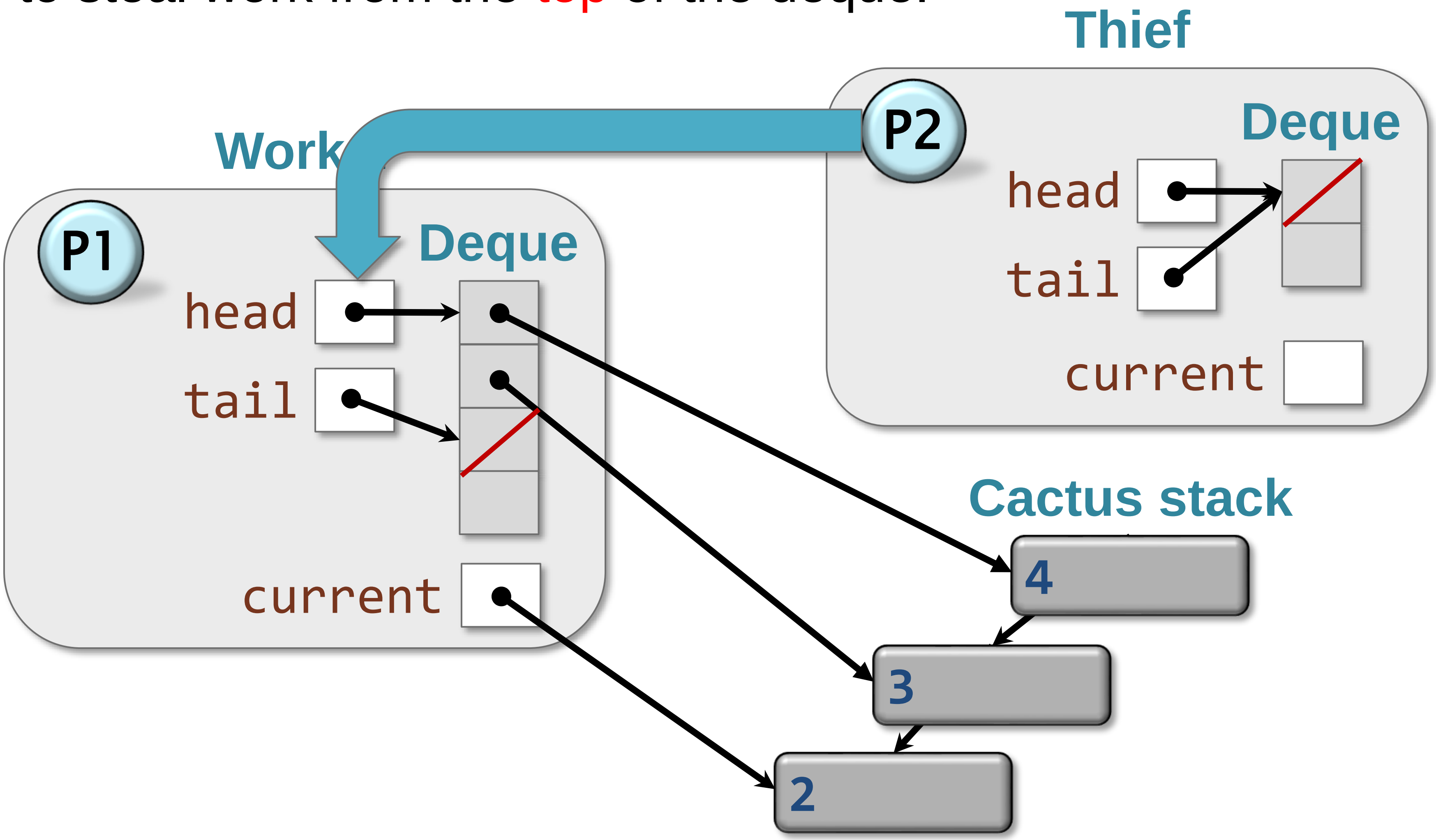
```
bool steal() {
  lock(L);
  head++;
  if (head > tail) {
    head--;
    unlock(L);
    return FAILURE;
  }
  lock(L);
```

Thieves always grab the lock.

…and only grab the deque's lock if the deque appears to be empty.

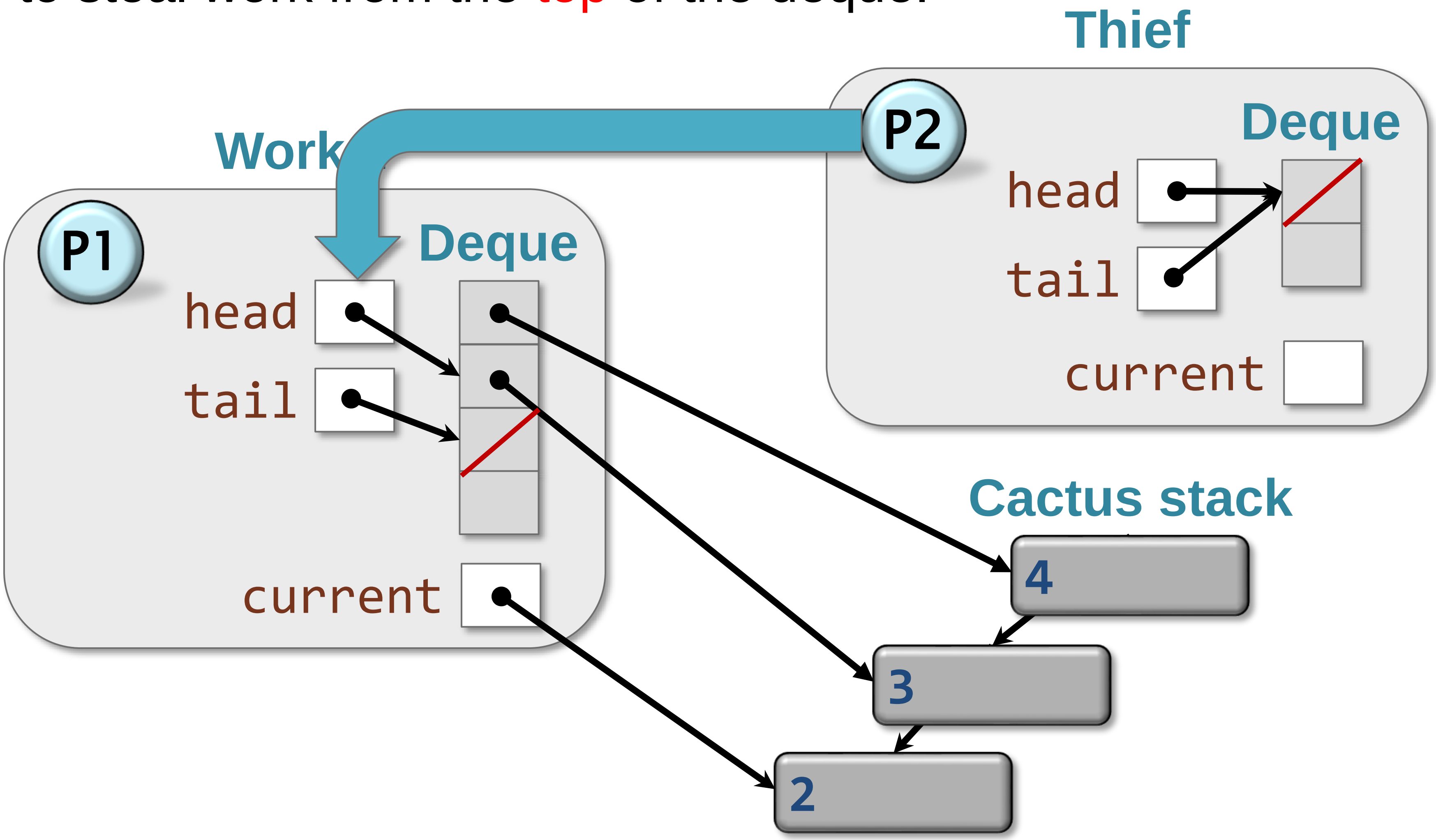
Successful Steal

Workers operate on the bottom of the deque, while thieves try to steal work from the top of the deque.



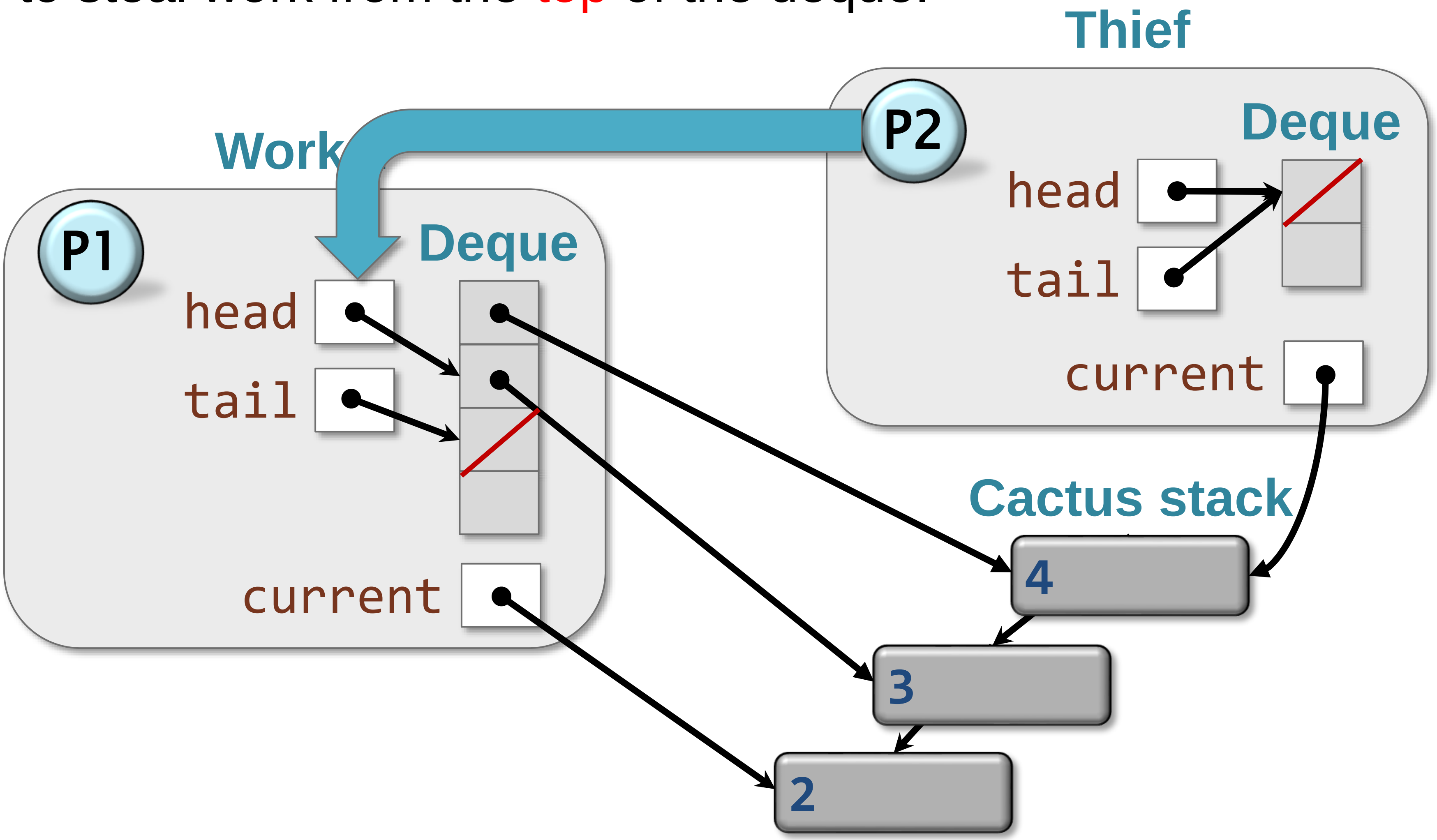
Successful Steal

Workers operate on the bottom of the deque, while thieves try to steal work from the top of the deque.



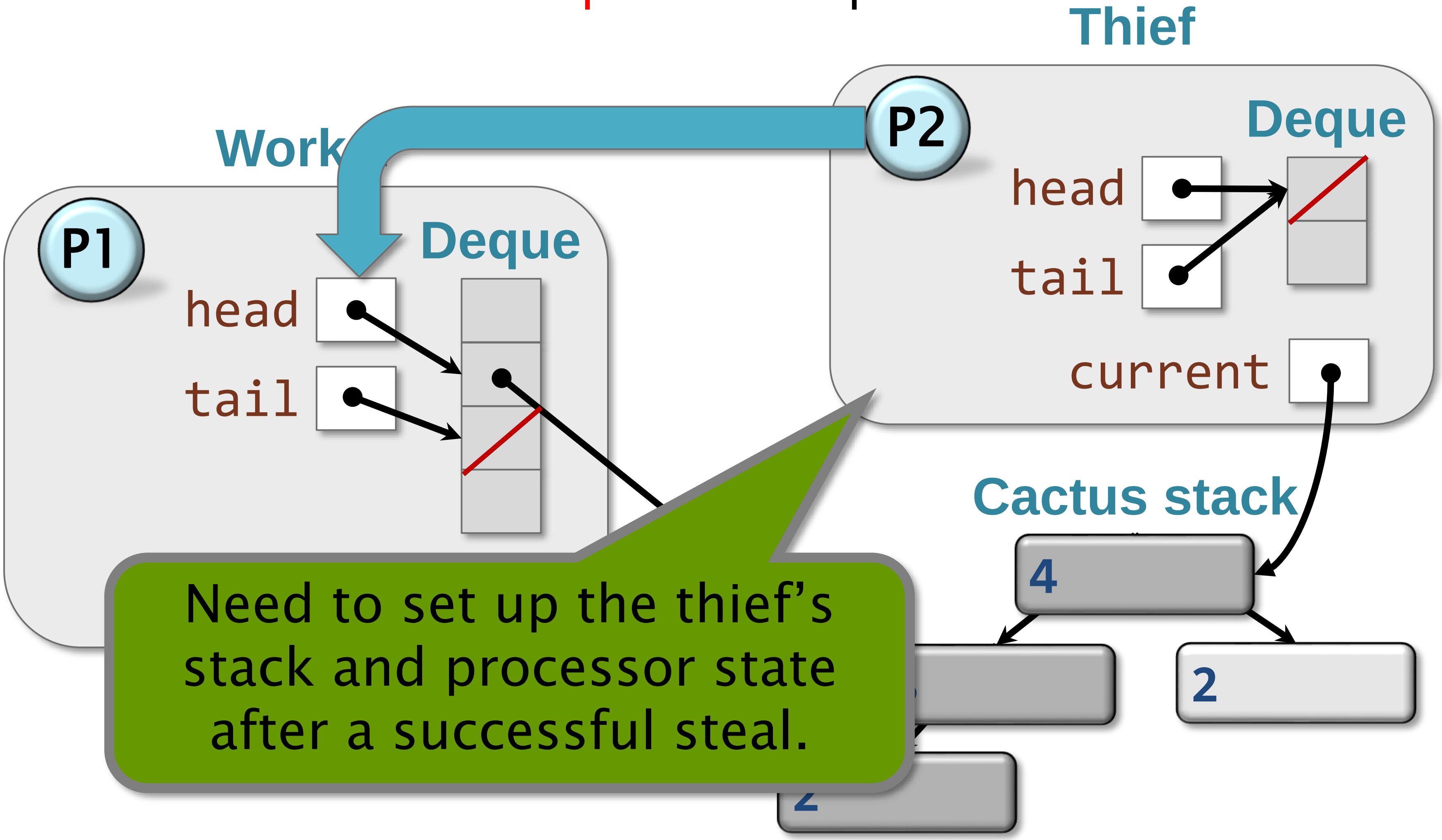
Successful Steal

Workers operate on the bottom of the deque, while thieves try to steal work from the top of the deque.



Successful Steal

Workers operate on the bottom of the deque, while thieves try to steal work from the top of the deque.



Need to set up the thief's stack and processor state after a successful steal.

Saving and Restoring Processor State

To save and restore processor state, the Cilk compiler allocates a local buffer in each frame that spawns.

Cilk code

```
x = cilk_spawn fib(n-1);
```

Compiled pseudocode

```
BUFFER ctx;
SAVE_STATE(&ctx);
if (!setjmp(&ctx))
  x = fib(n-1);
// (continuation)
```

Saving and Restoring Processor State

To save and restore processor state, the Cilk compiler allocates a local buffer in each frame that spawns.

Cilk code

```
x = cilk_spawn fib(n-1);
```

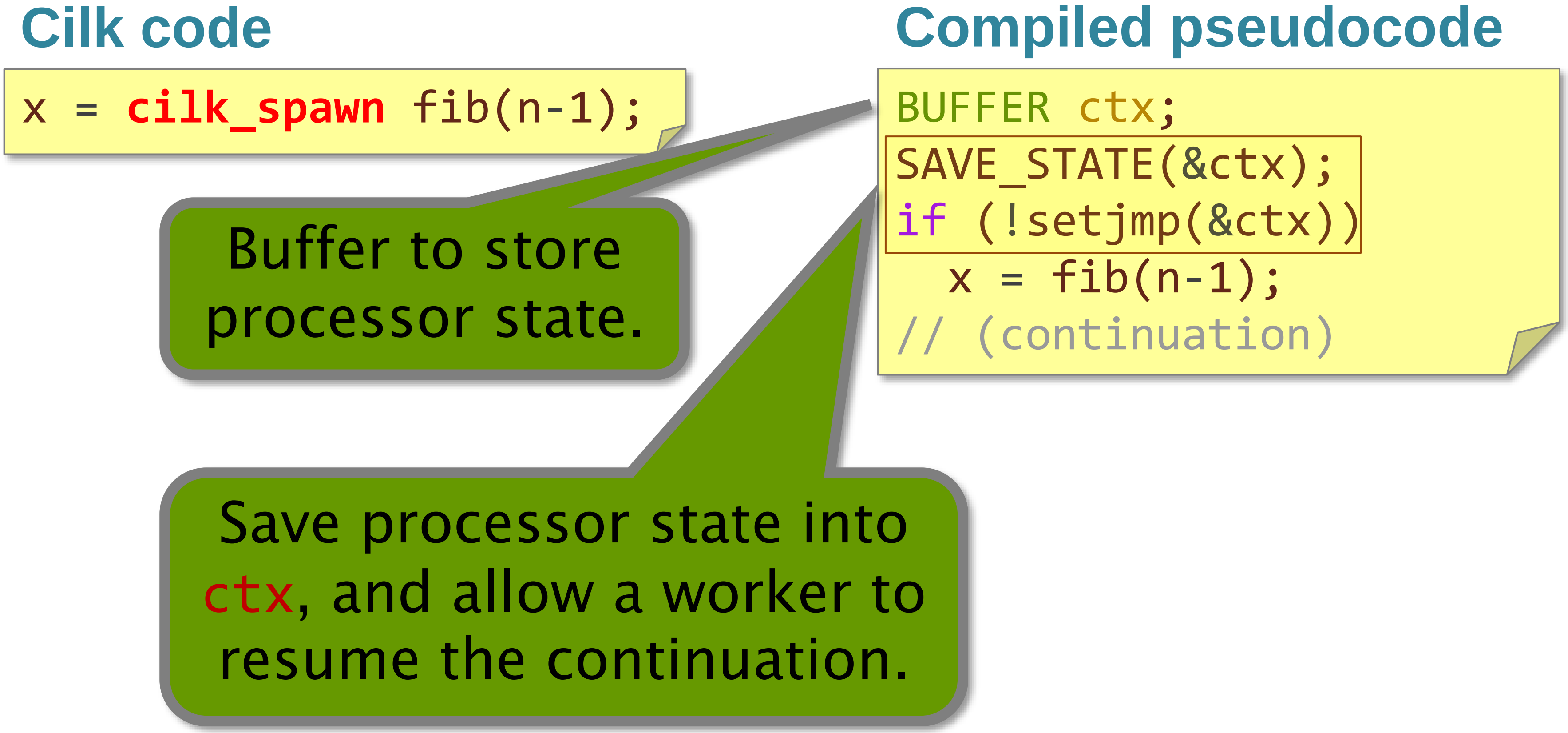
Compiled pseudocode

```
BUFFER ctx;
SAVE_STATE(&ctx);
if (!setjmp(&ctx))
  x = fib(n-1);
// (continuation)
```

Buffer to store processor state.

Saving and Restoring Processor State

To save and restore processor state, the Cilk compiler allocates a local buffer in each frame that spawns.



Cilk code

```
x = cilk_spawn fib(n-1);
```

Compiled pseudocode

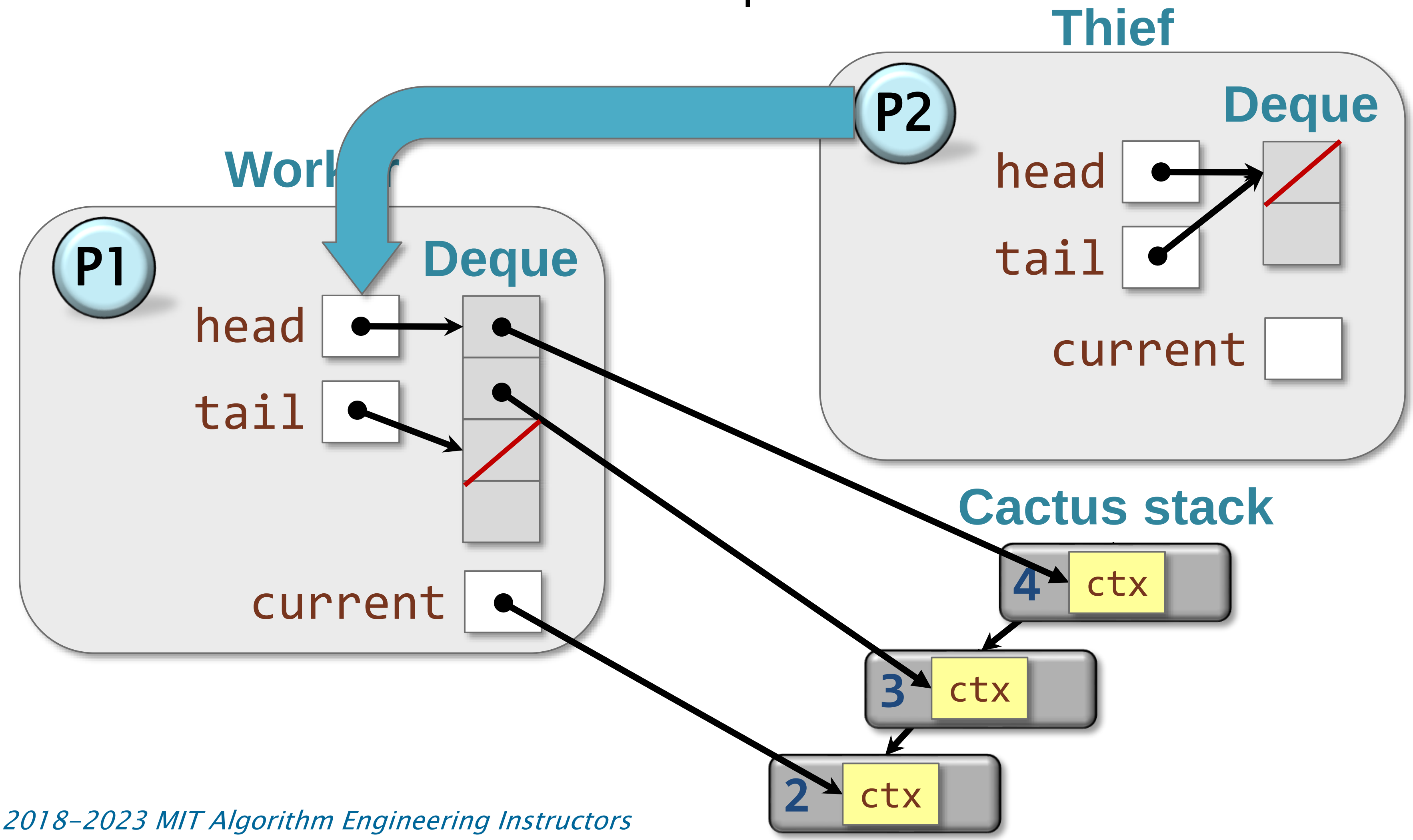
```
BUFFER ctx;
SAVE_STATE(&ctx);
if (!setjmp(&ctx))
  x = fib(n-1);
// (continuation)
```

Buffer to store processor state.

Save processor state into ctx, and allow a worker to resume the continuation.

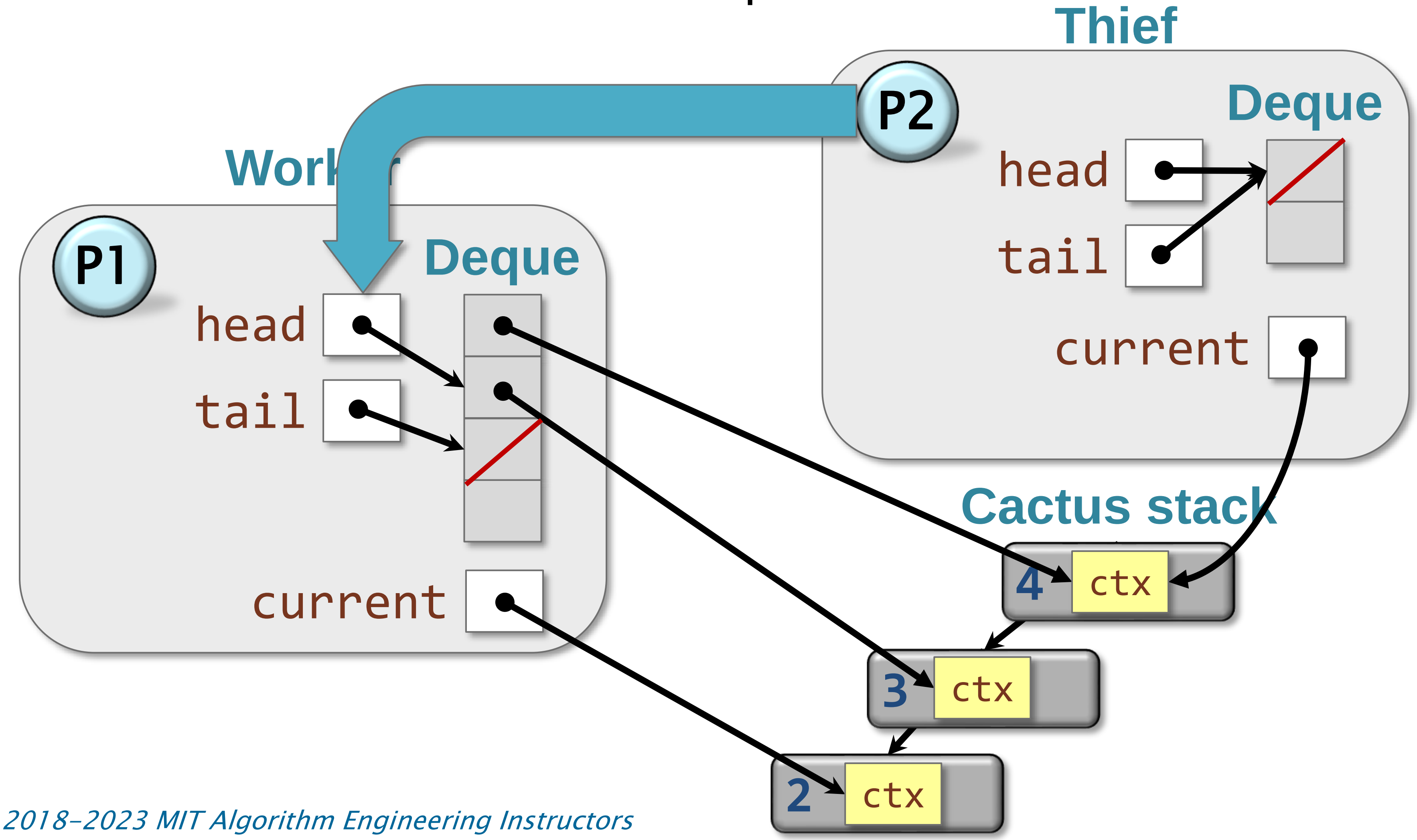
Deque References to Frames

Worker deques store references to the buffers in each frame, from which thieves can retrieve processor state.



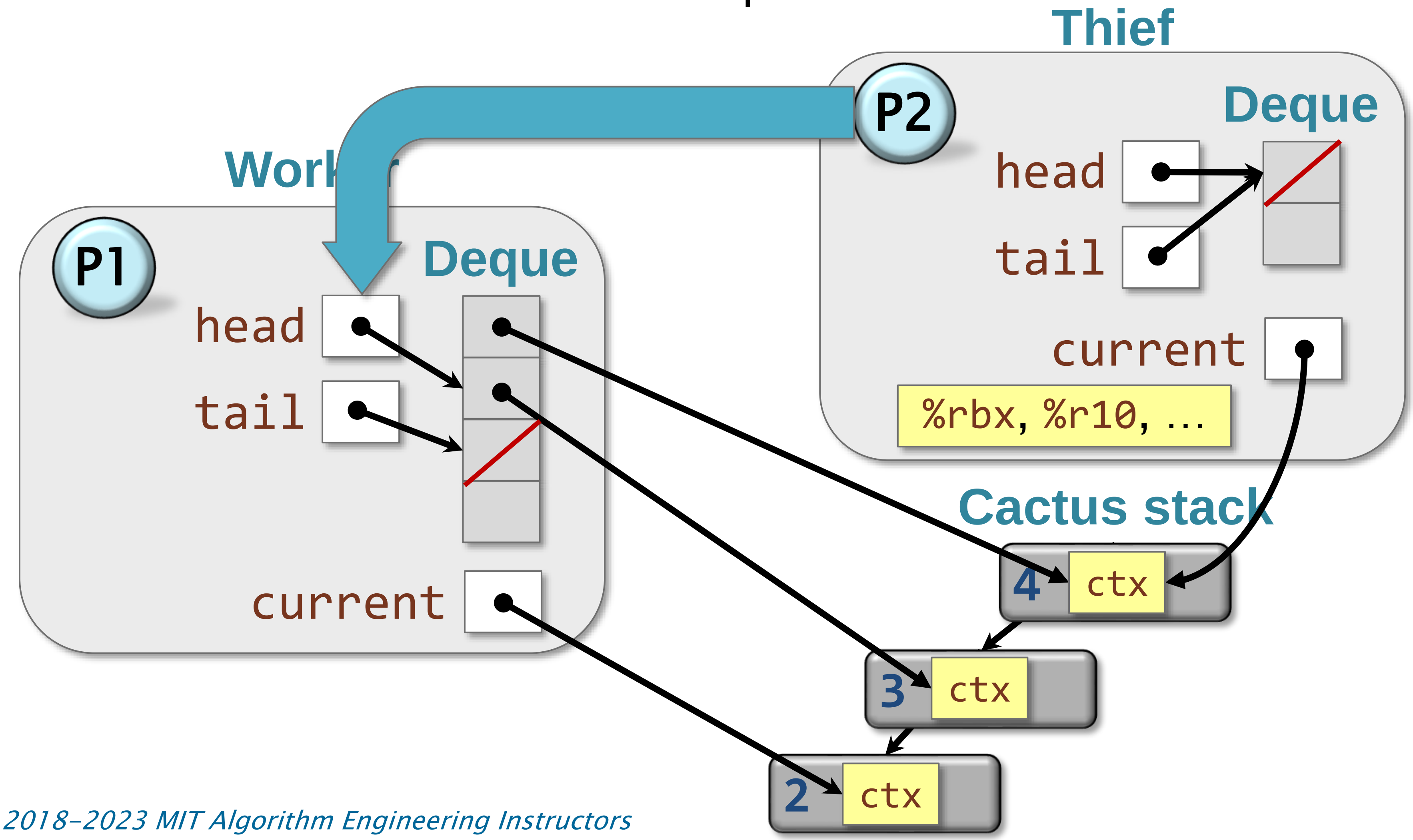
Deque References to Frames

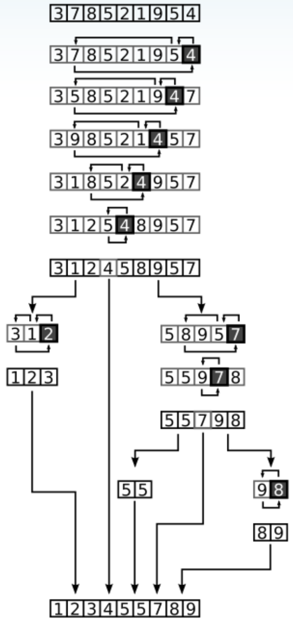
Worker deques store references to the buffers in each frame, from which thieves can retrieve processor state.



Deque References to Frames

Worker deques store references to the buffers in each frame, from which thieves can retrieve processor state.



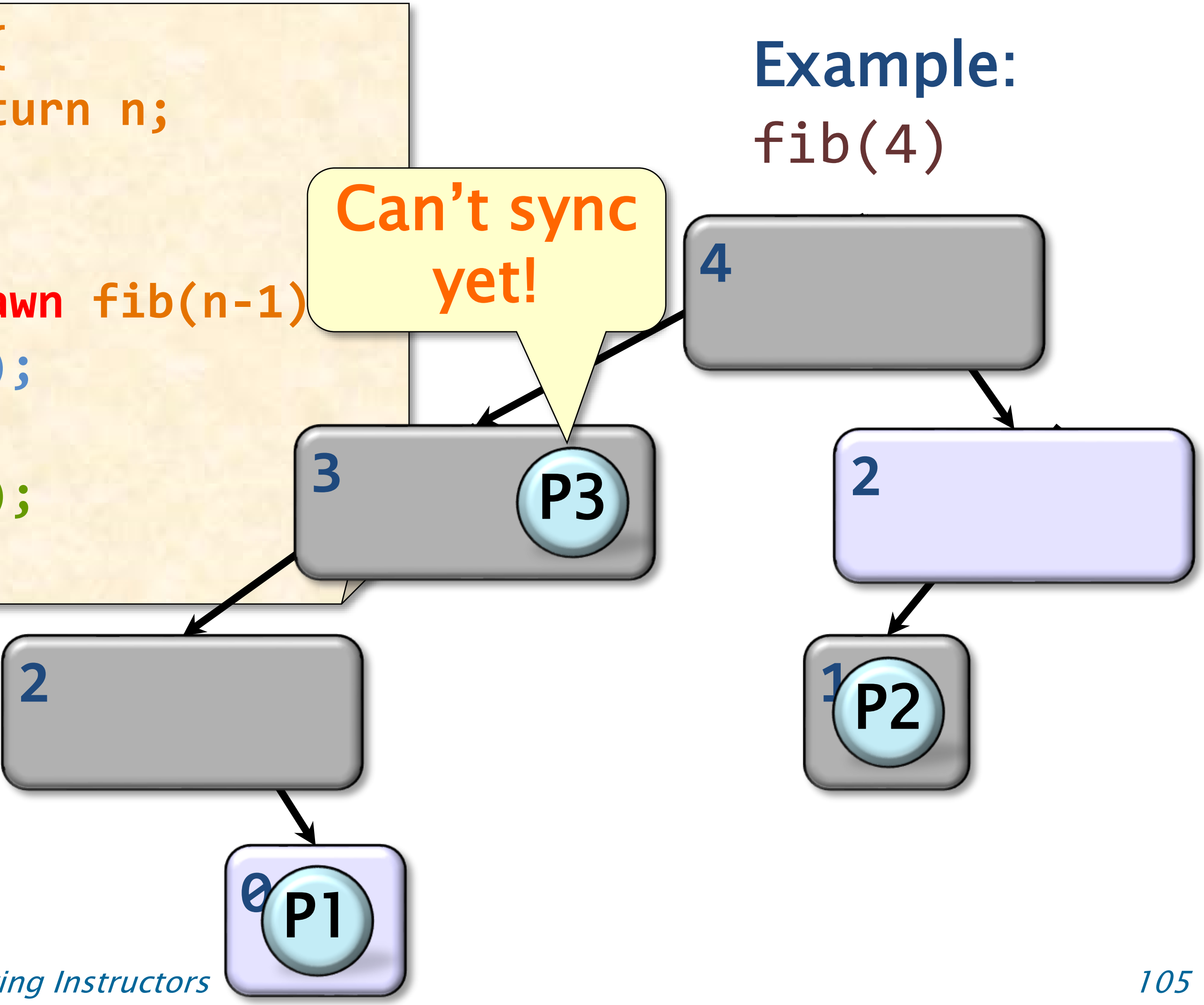


Syncs: The Full-Frame Tree

Semantics of Sync

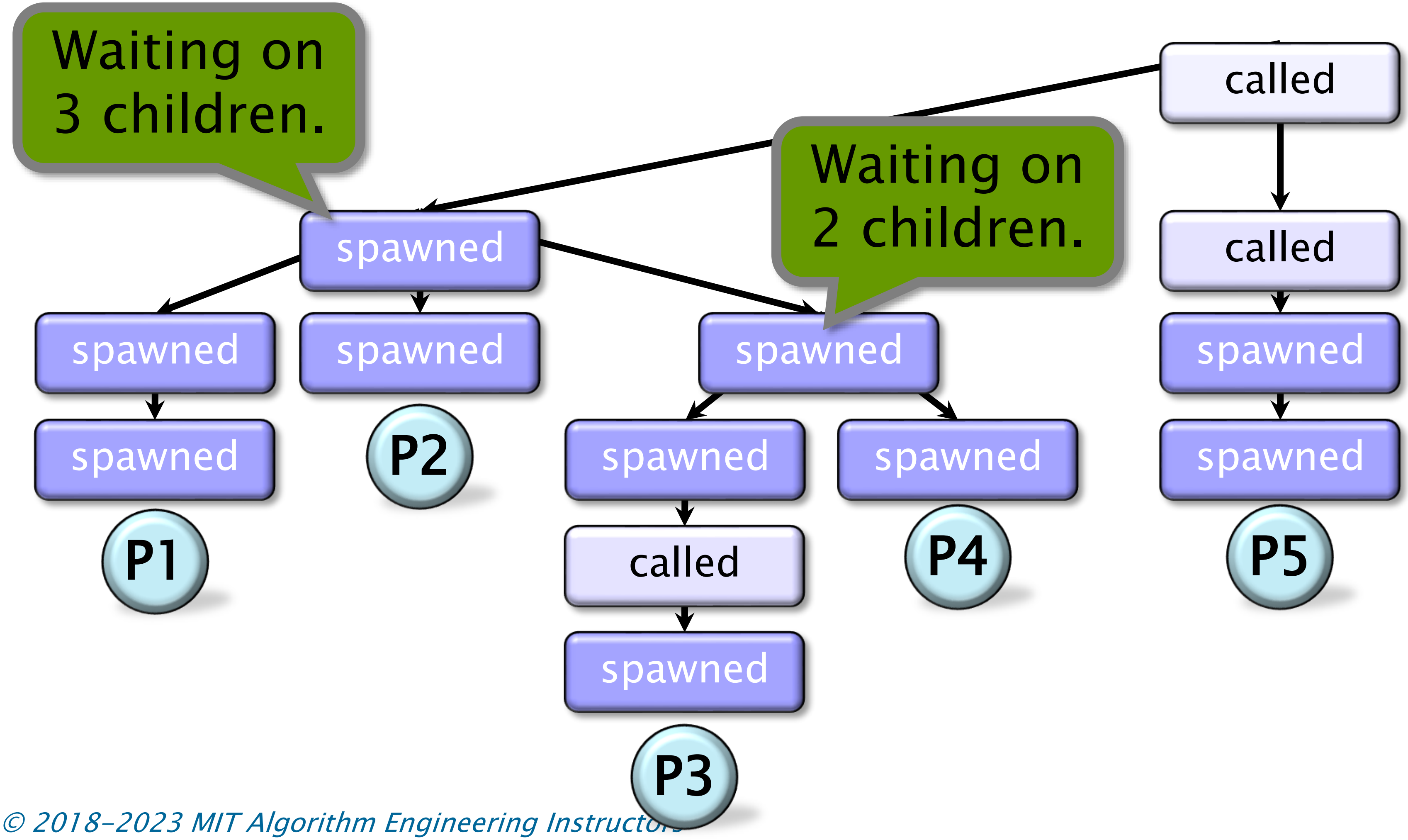
A **cilk_scope** waits on child frames, not on workers.

```
int fib(int n) {
  if (n < 2) return n;
  int x, y;
  cilk_scope {
    x = cilk_spawn fib(n-1)
    y = fib(n-2);
  }
  return (x + y);
}
```



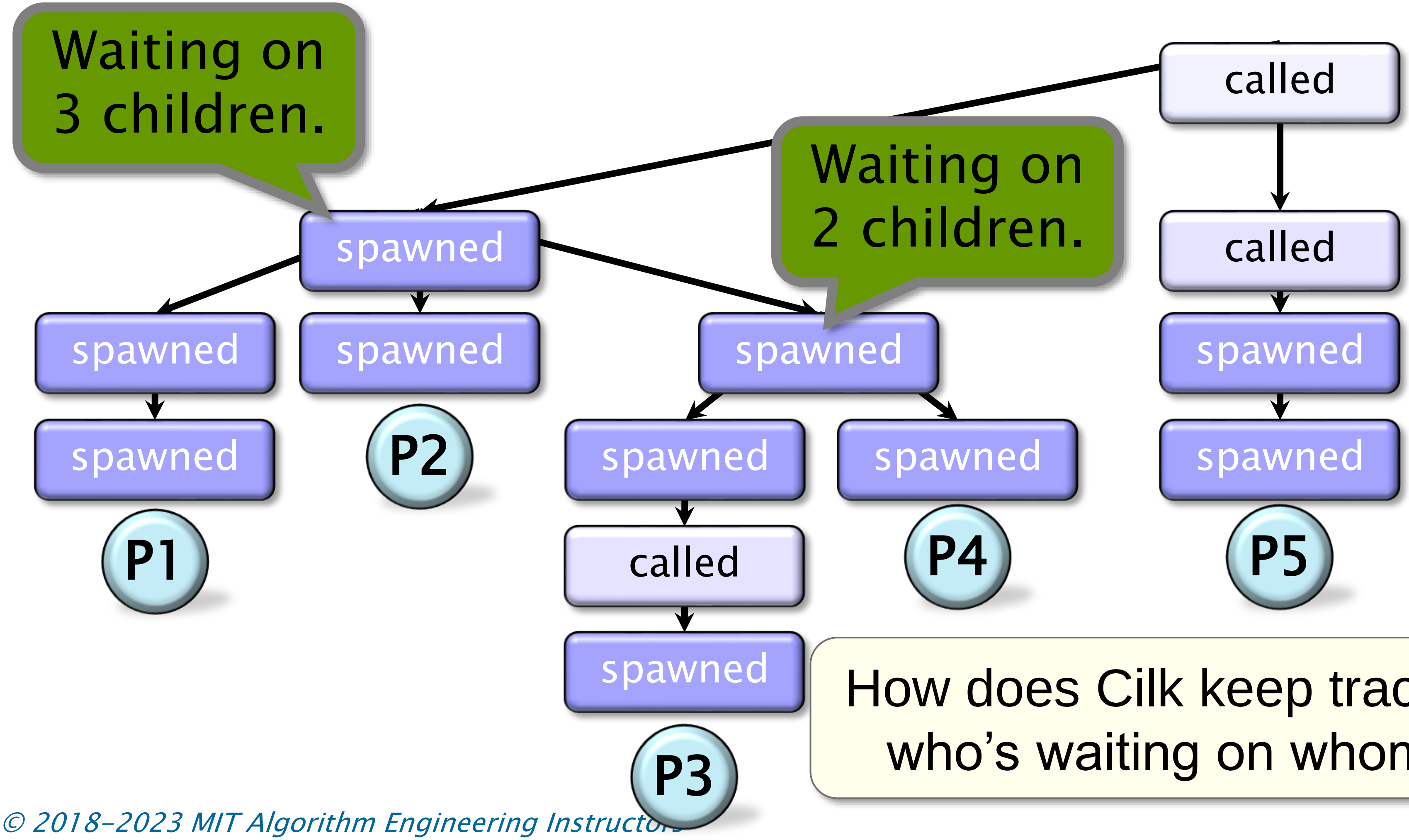
Nested Synchronization

Cilk supports nested synchronization, where a frame waits only on its child subcomputations.



Nested Synchronization

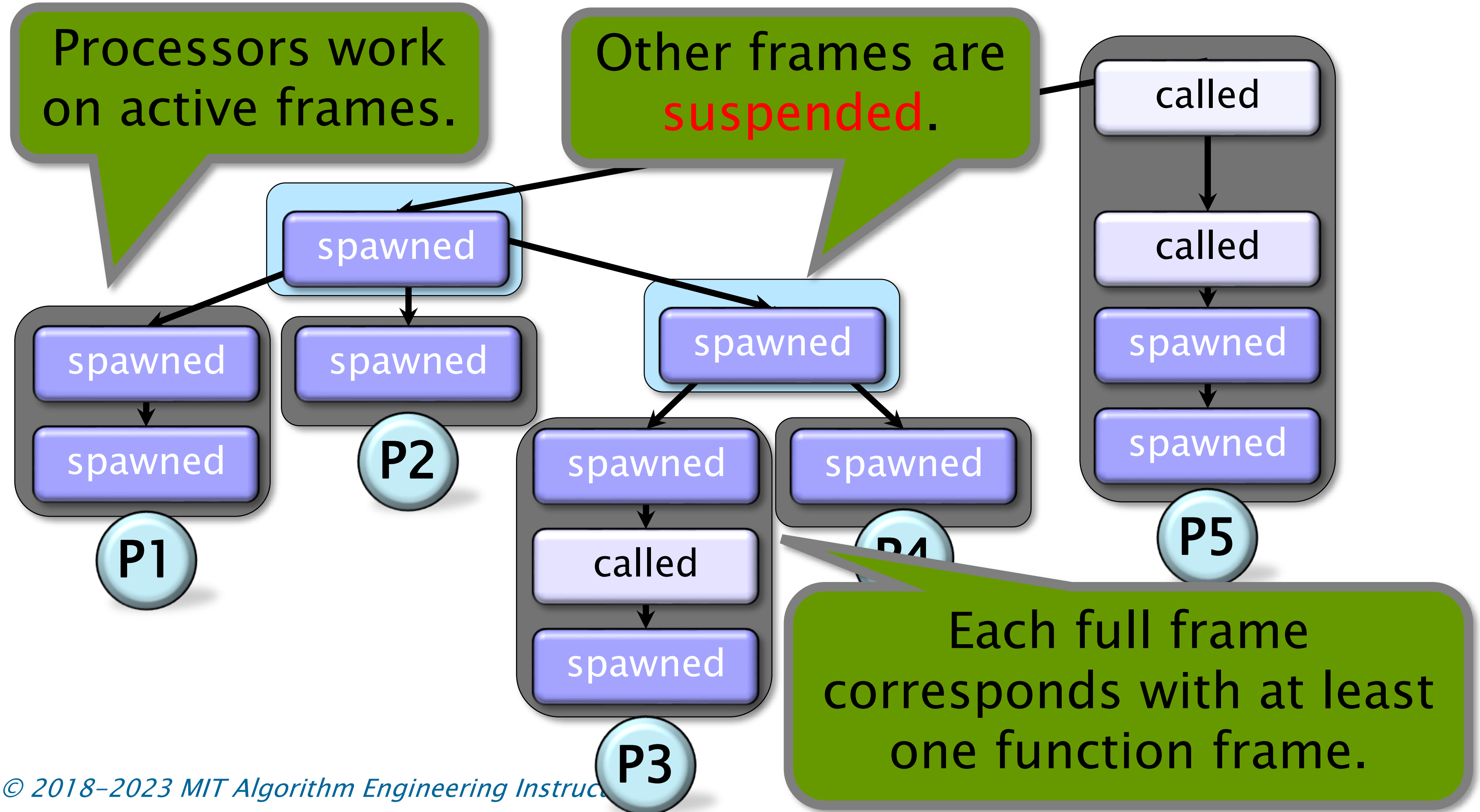
Cilk supports nested synchronization, where a frame waits only on its child subcomputations.



How does Cilk keep track of who's waiting on whom?

Full-Frame Tree

The Cilk runtime maintains a tree of full frames to keep track of synchronization information.



Processors work on active frames.

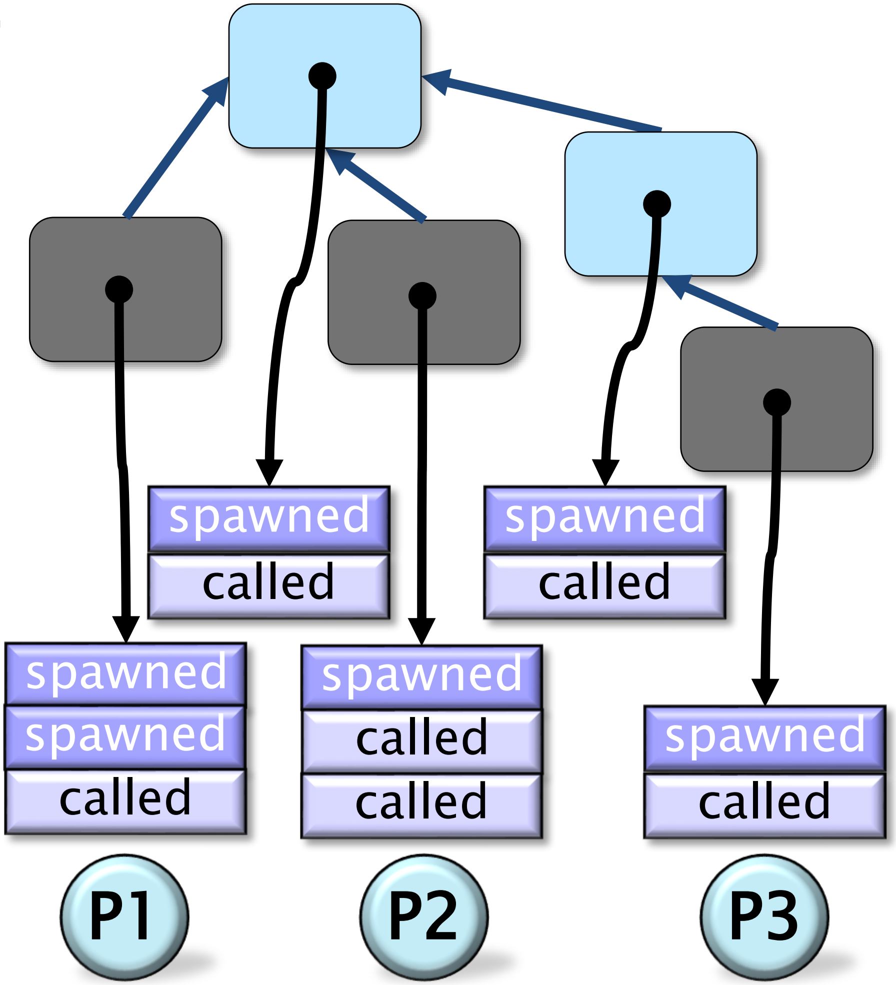
Other frames are suspended.

Each full frame corresponds with at least one function frame.

Full-Frame Data

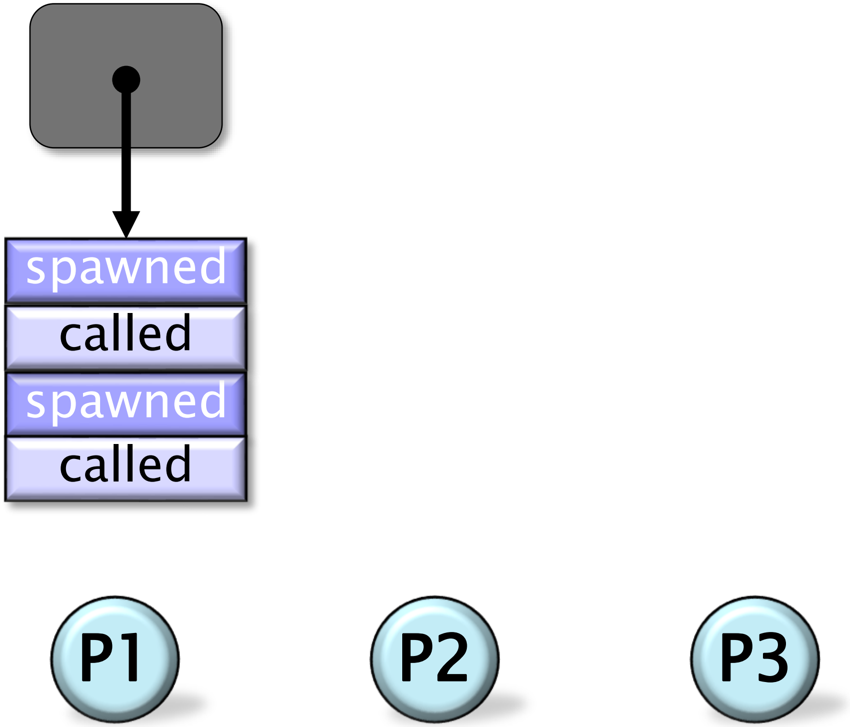
To maintain the state of the running program, each full frame maintains:

- A join counter of the number of (unsynched) child frames.
- References to parent and child full frames.
- References into the corresponding Cilk stack frames on the cactus stack.



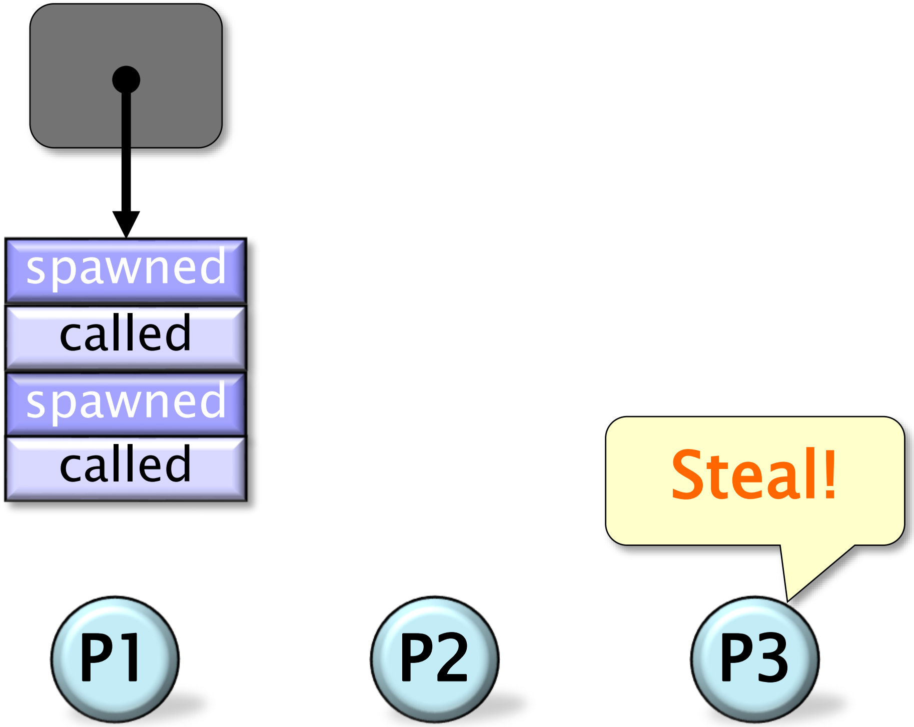
Maintaining the Full-Frame Tree

Let’s see how the tree structure is maintained.



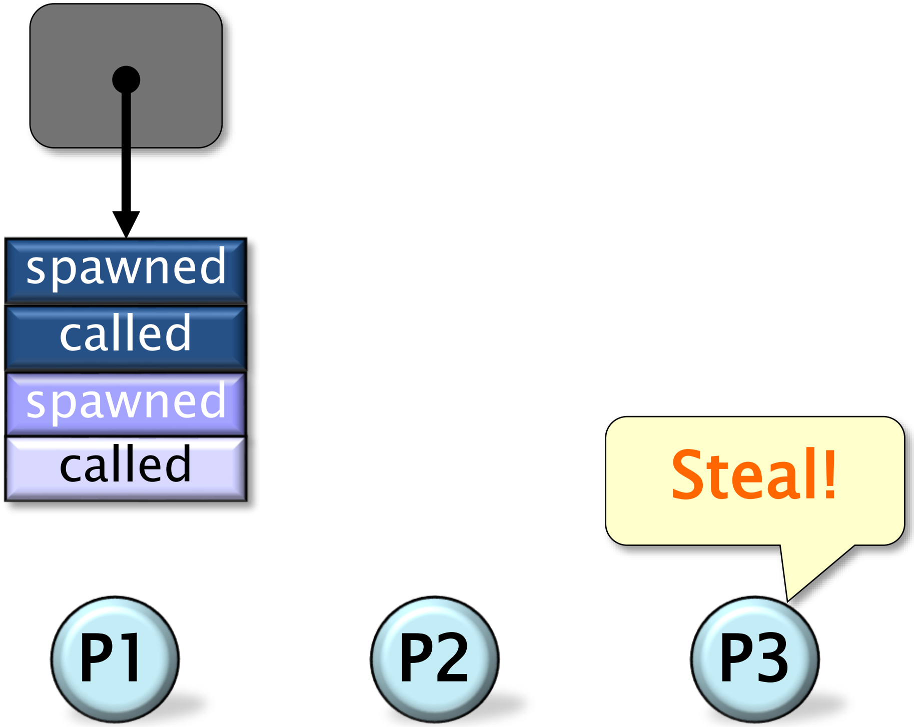
Maintaining the Full-Frame Tree

Let's see how the tree structure is maintained.



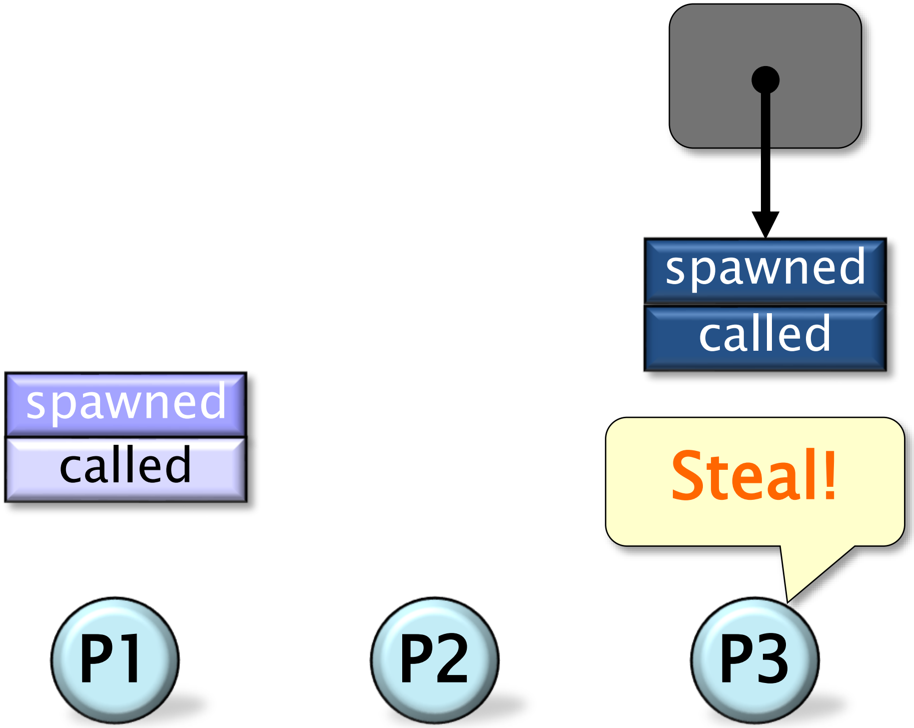
Maintaining the Full-Frame Tree

Let’s see how the tree structure is maintained.



Maintaining the Full-Frame Tree

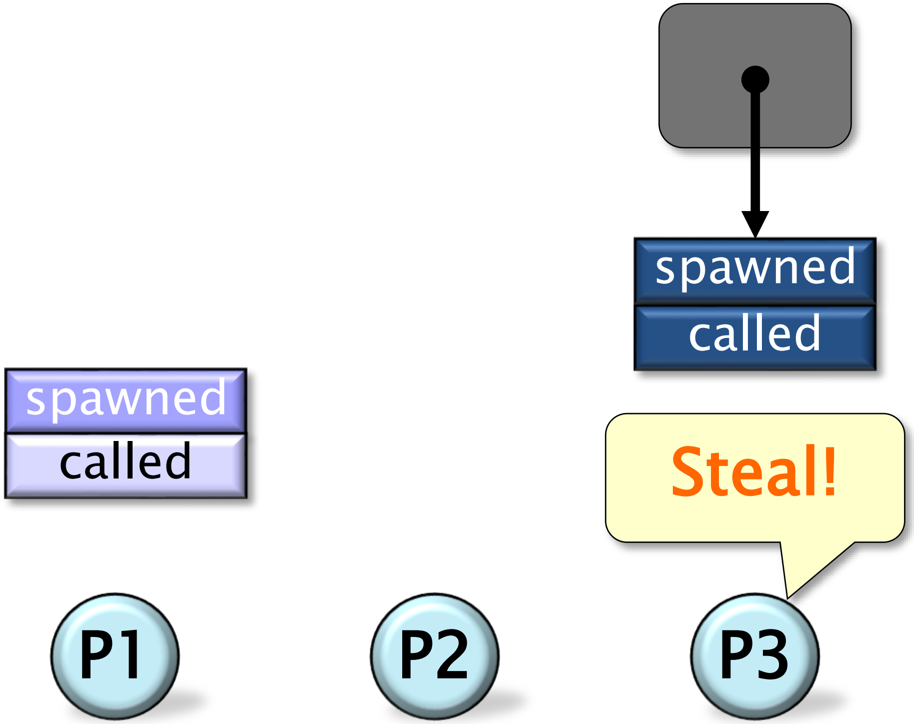
Let’s see how the tree structure is maintained.



Maintaining the Full-Frame Tree

Let’s see how the tree structure is maintained.

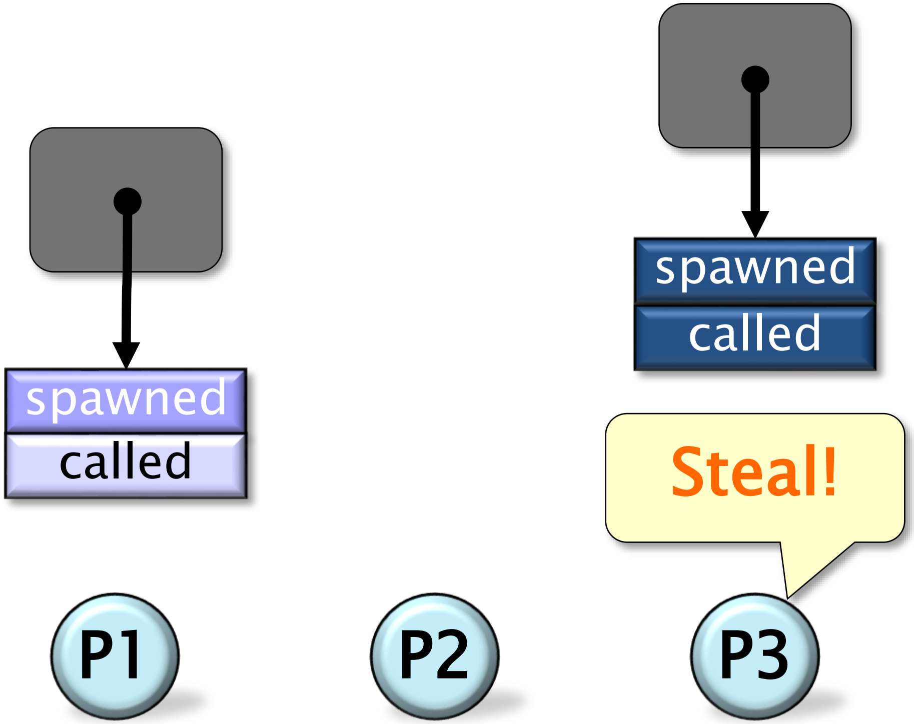
The thief steals the full frame and creates a new full frame for the victim.



Maintaining the Full-Frame Tree

Let’s see how the tree structure is maintained.

The thief steals the full frame and creates a new full frame for the victim.

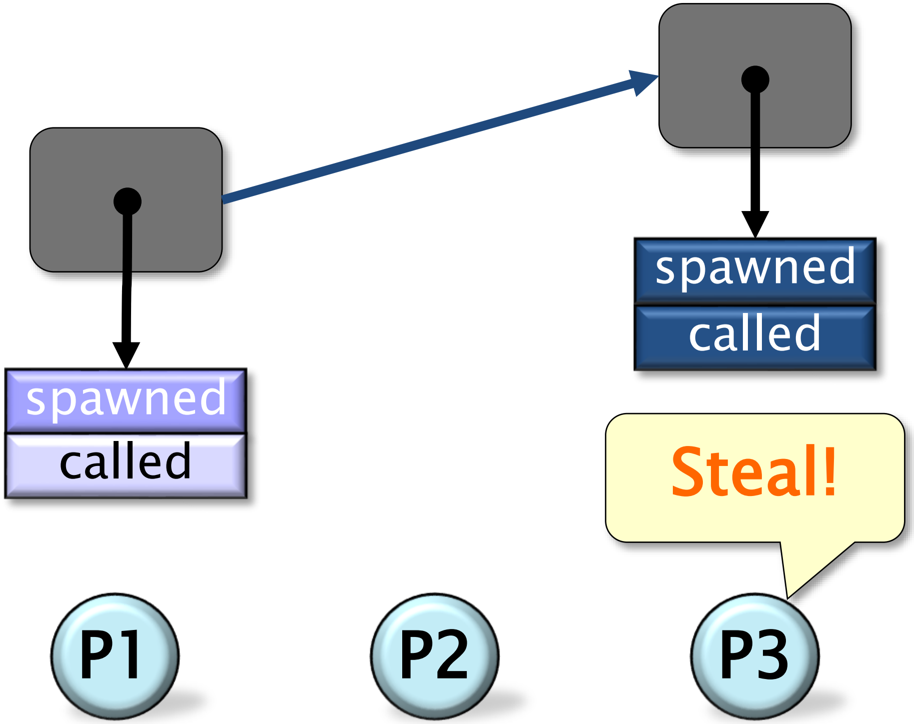


Maintaining the Full–Frame Tree

Let’s see how the tree structure is maintained.

The thief steals the full frame and creates a new full frame for the victim.

The victim’s new full frame is a child of the stolen full frame.

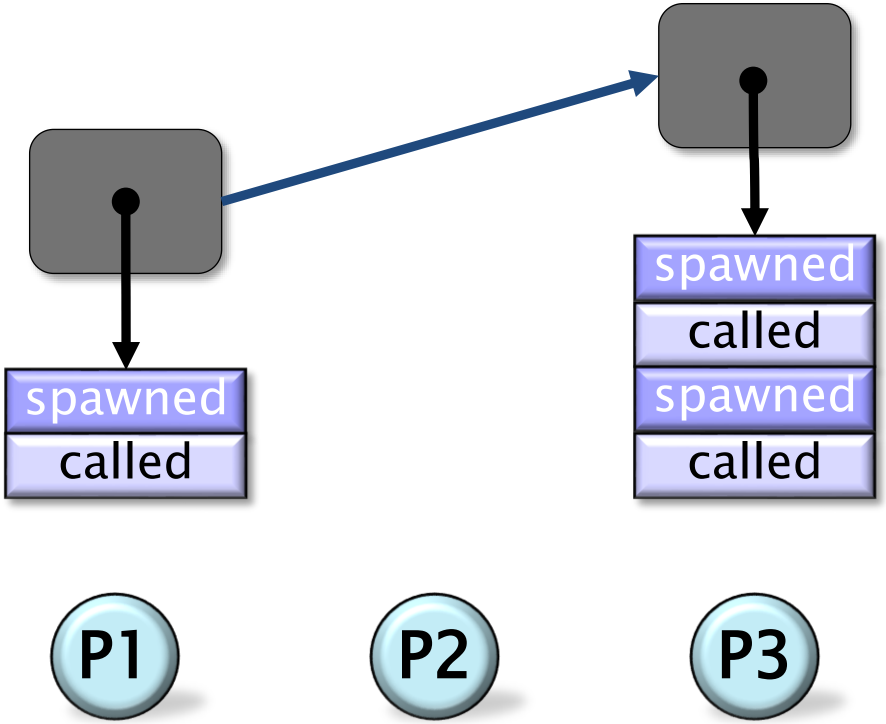


Maintaining the Full-Frame Tree

Let’s see how the tree structure is maintained.

The thief steals the full frame and creates a new full frame for the victim.

The victim’s new full frame is a child of the stolen full frame.

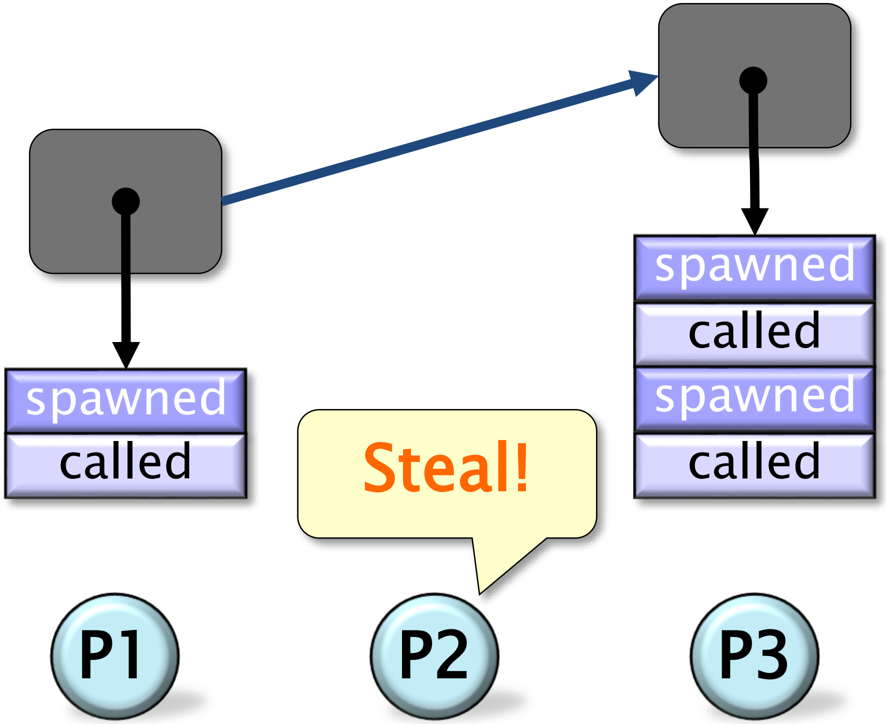


Maintaining the Full-Frame Tree

Let’s see how the tree structure is maintained.

The thief steals the full frame and creates a new full frame for the victim.

The victim’s new full frame is a child of the stolen full frame.

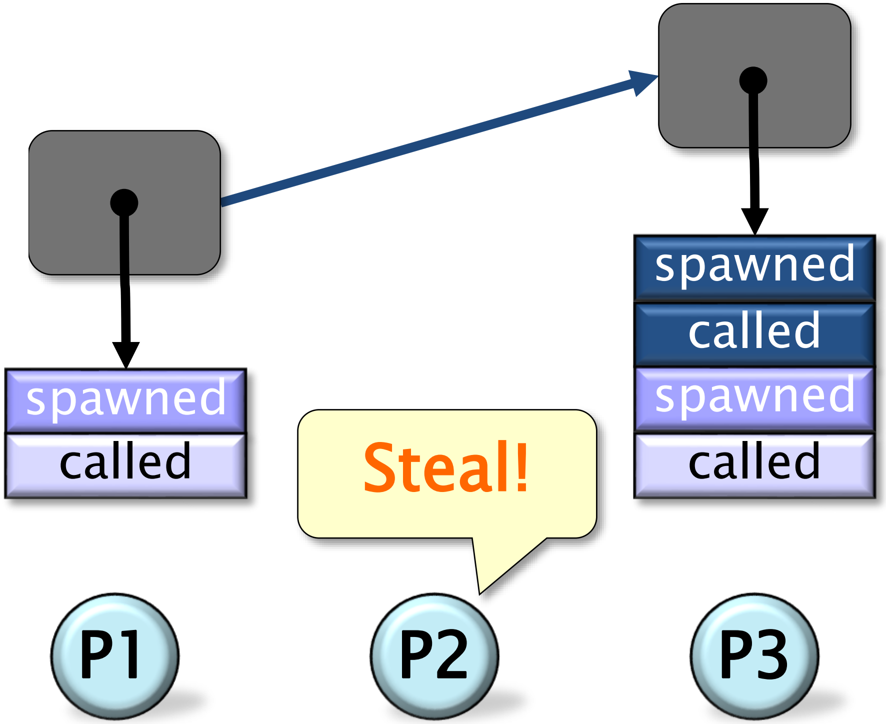


Maintaining the Full-Frame Tree

Let’s see how the tree structure is maintained.

The thief steals the full frame and creates a new full frame for the victim.

The victim’s new full frame is a child of the stolen full frame.

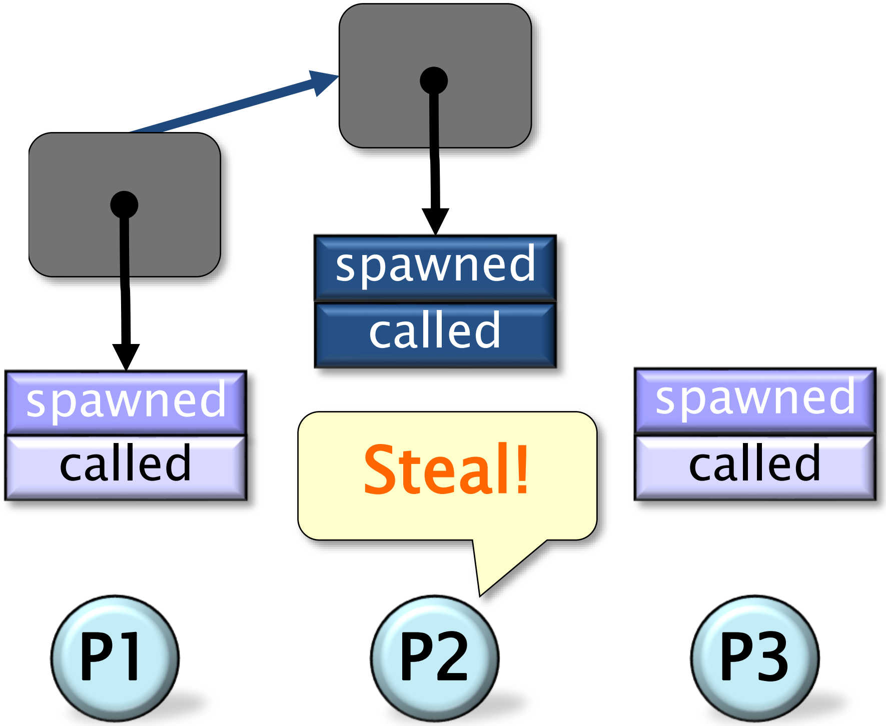


Maintaining the Full-Frame Tree

Let’s see how the tree structure is maintained.

The thief steals the full frame and creates a new full frame for the victim.

The victim’s new full frame is a child of the stolen full frame.

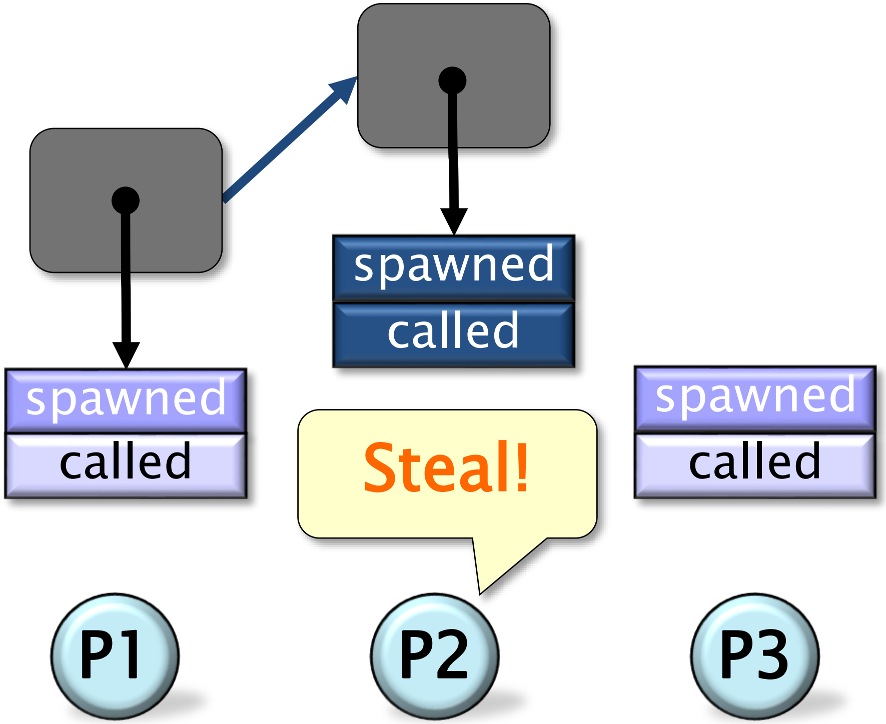


Maintaining the Full-Frame Tree

Let’s see how the tree structure is maintained.

The thief steals the full frame and creates a new full frame for the victim.

The victim’s new full frame is a child of the stolen full frame.

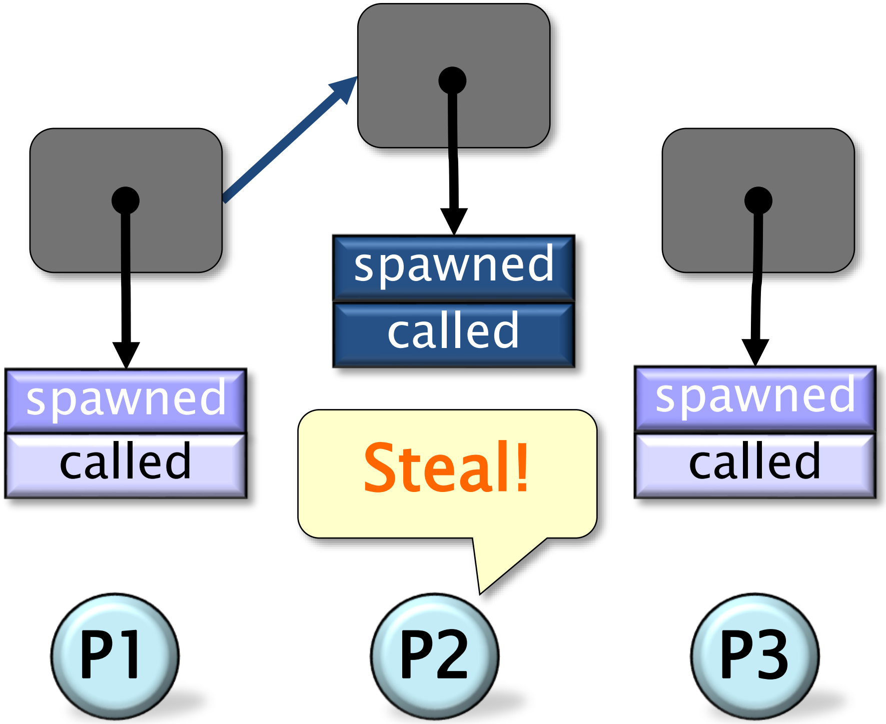


Maintaining the Full-Frame Tree

Let’s see how the tree structure is maintained.

The thief steals the full frame and creates a new full frame for the victim.

The victim’s new full frame is a child of the stolen full frame.

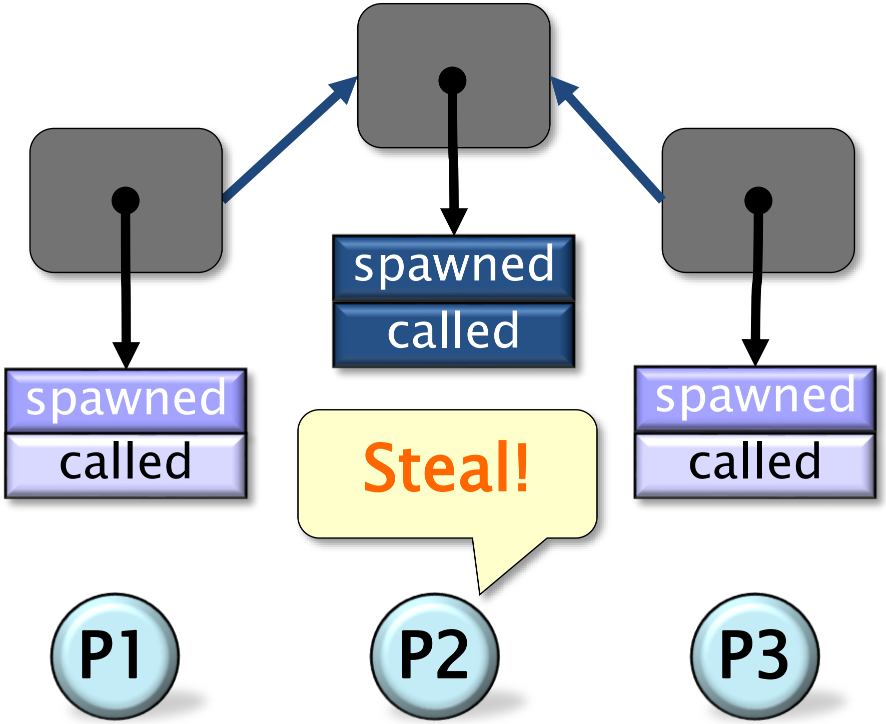


Maintaining the Full-Frame Tree

Let’s see how the tree structure is maintained.

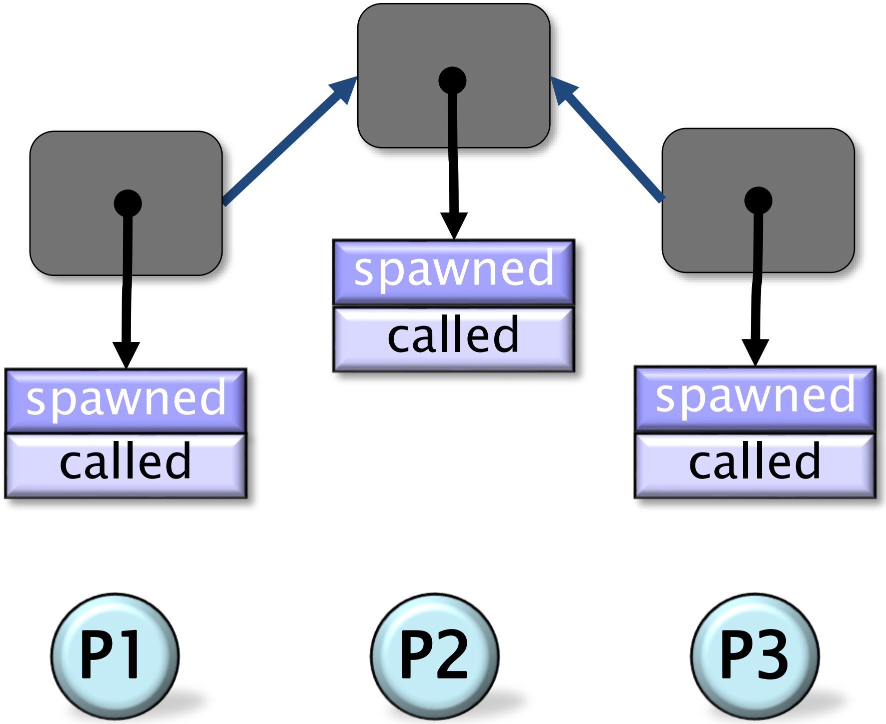
The thief steals the full frame and creates a new full frame for the victim.

The victim’s new full frame is a child of the stolen full frame.



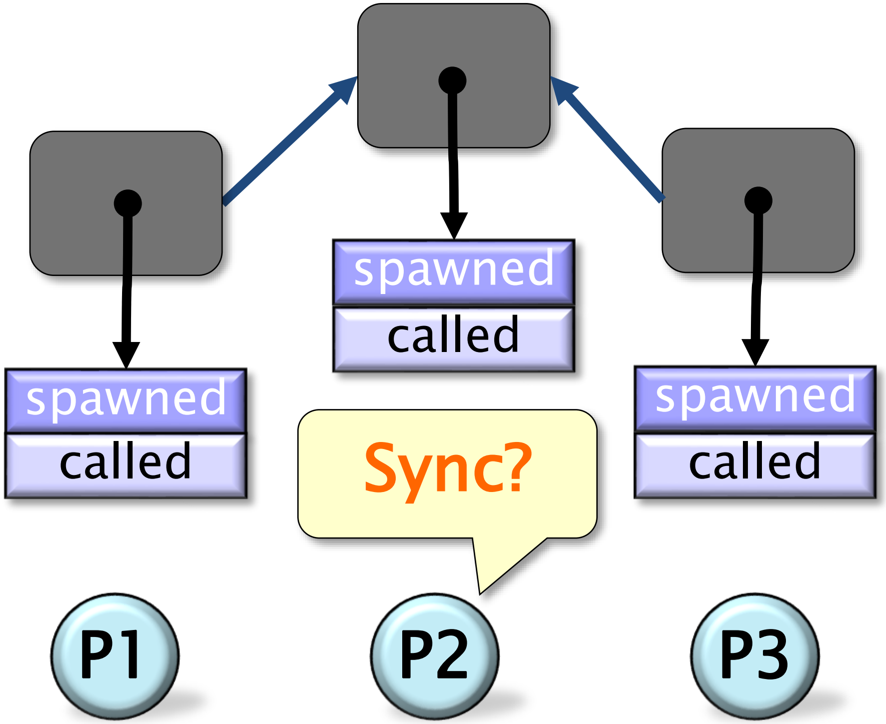
Maintaining the Full-Frame Tree

Let’s see how the tree structure is maintained.



Maintaining the Full-Frame Tree

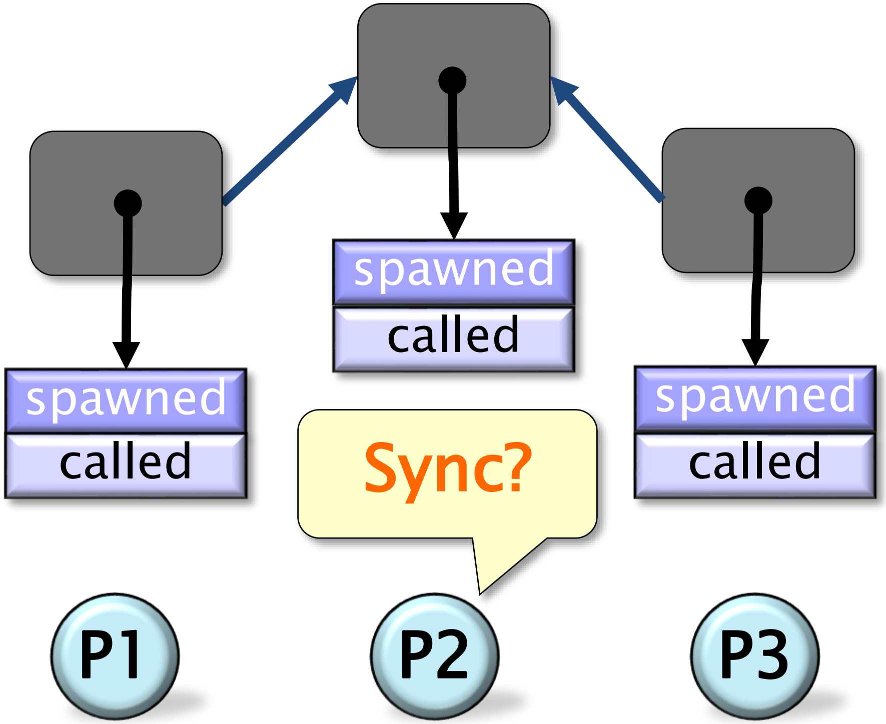
Let’s see how the tree structure is maintained.



Maintaining the Full-Frame Tree

Let’s see how the tree structure is maintained.

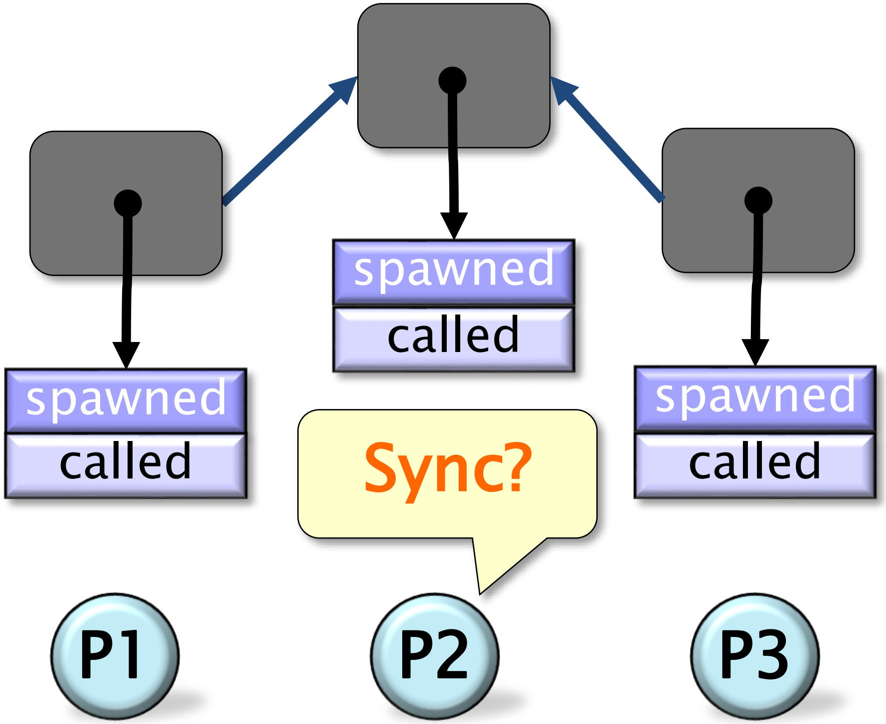
A full frame suspends at a sync if it has outstanding child frames.



Maintaining the Full-Frame Tree

Let’s see how the tree structure is maintained.

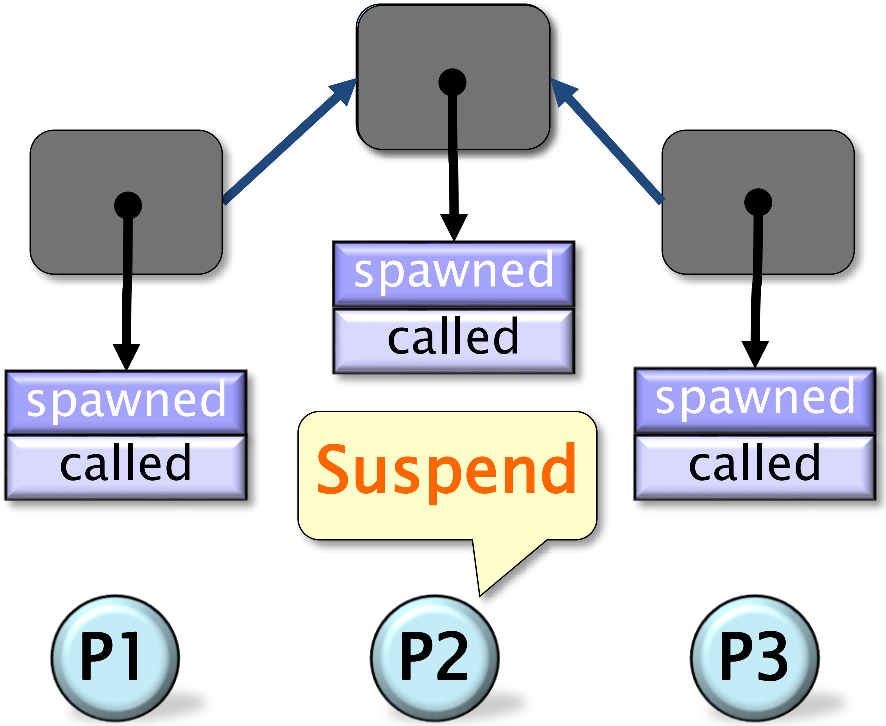
A full frame suspends at a sync if it has outstanding child frames.



Maintaining the Full-Frame Tree

Let’s see how the tree structure is maintained.

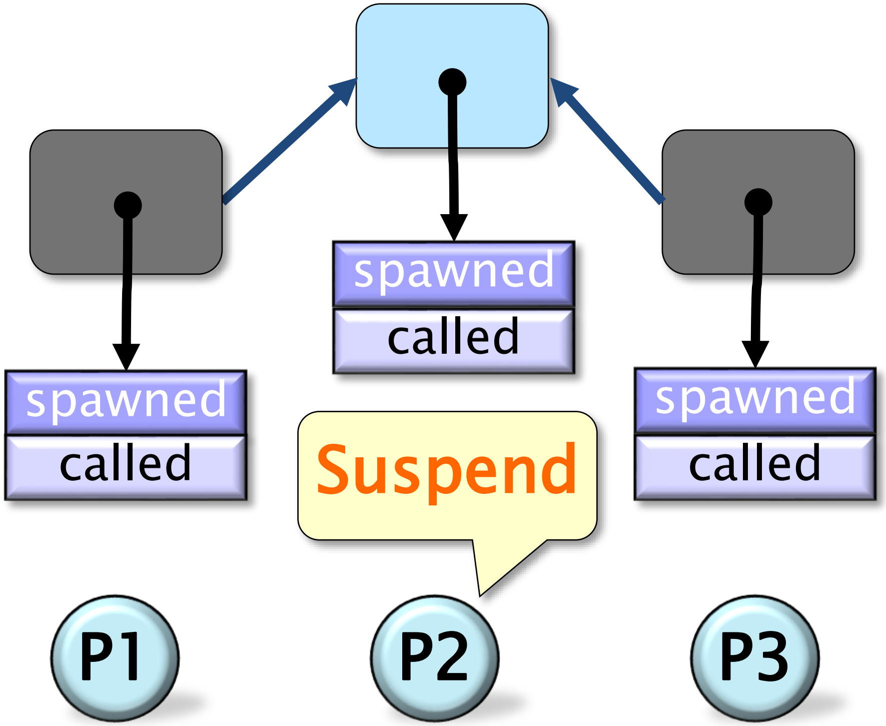
A full frame suspends at a sync if it has outstanding child frames.



Maintaining the Full-Frame Tree

Let’s see how the tree structure is maintained.

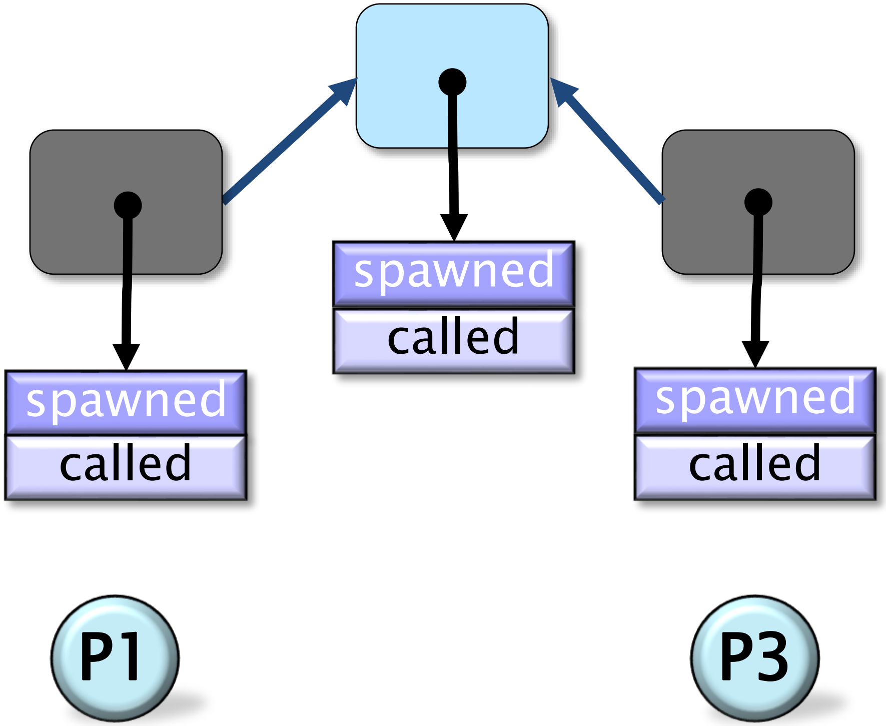
A full frame suspends at a sync if it has outstanding child frames.



Maintaining the Full-Frame Tree

Let’s see how the tree structure is maintained.

A full frame suspends at a sync if it has outstanding child frames.



Common Case for Sync

Question: If the program has ample parallelism, what do we expect typically happens when the program execution reaches the end of a `cilk_scope`?

Common Case for Sync

Question: If the program has ample parallelism, what do we expect typically happens when the program execution reaches the end of a `cilk_scope`?

Answer: The executing function contains no outstanding spawned children.

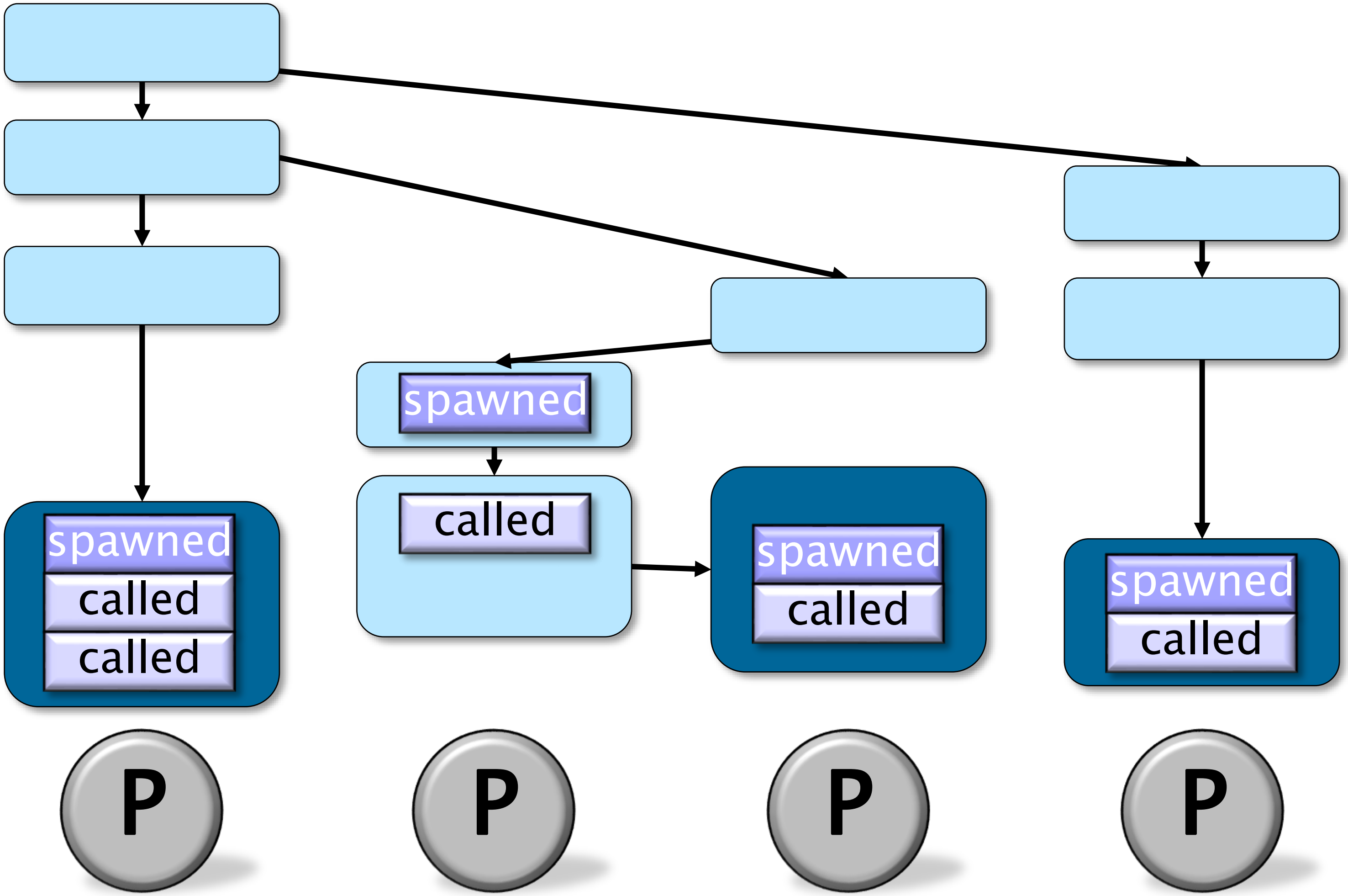
Common Case for Sync

Question: If the program has ample parallelism, what do we expect typically happens when the program execution reaches the end of a **`cilk_scope`**?

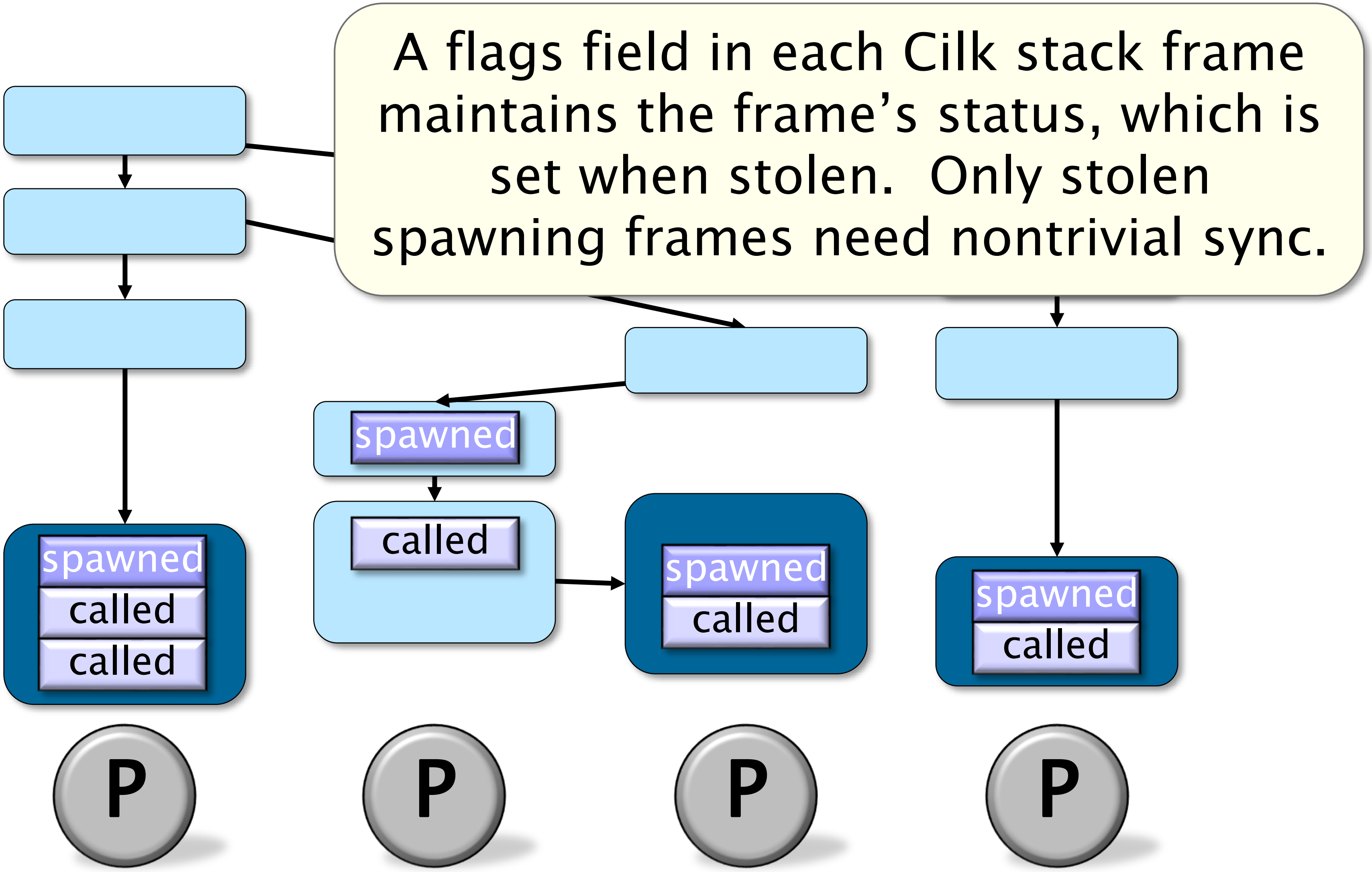
Answer: The executing function contains no outstanding spawned children.

How does the runtime optimize for this case?

Managing the Full-Frame Tree: Sync



Managing the Full-Frame Tree: Sync



A flags field in each Cilk stack frame maintains the frame’s status, which is set when stolen. Only stolen spawning frames need nontrivial sync.

Compiled Code for Sync

Like `cilk_spawn`, a `cilk_scope` is compiled using `setjmp`, in order to save the processor’s state when the frame is suspended.

Cilk code

```
cilk_scope { … };
```

C pseudocode

```
BUFFER ctx;
…
if (WAS_STOLEN)
  if (!setjmp(&ctx))
    __cilkrts_sync(&ctx);
```

Compiled Code for Sync

Like `cilk_spawn`, a `cilk_scope` is compiled using `setjmp`, in order to save the processor's state when the frame is suspended.

Cilk code

```
cilk_scope { … };
```

Same buffer as used for spawns.

C pseudocode

```
BUFFER ctx;
…
if (WAS_STOLEN)
  if (!setjmp(&ctx))
    __cilkrts_sync(&ctx);
```

Compiled Code for Sync

Like `cilk_spawn`, a `cilk_scope` is compiled using `setjmp`, in order to save the processor’s state when the frame is suspended.

Cilk code

```
cilk_scope { … };
```

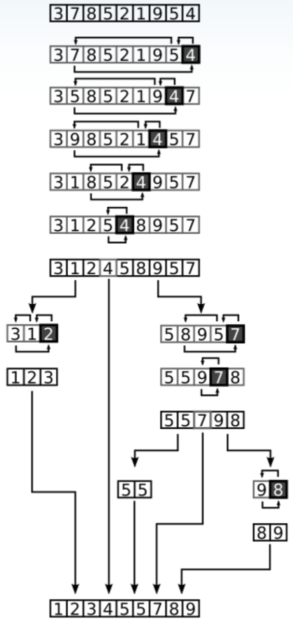
Same buffer as used for spawns.

C pseudocode

```
BUFFER ctx;
…
if (WAS_STOLEN)
  if (!setjmp(&ctx))
    __cilkrts_sync(&ctx);
```

Call into the runtime to suspend the frame.

DESIGN CHOICES



The Work-First Principle

To optimize the execution of programs with sufficient parallelism, the implementation of the Cilk runtime system works to maintain high work-efficiency by abiding by the work-first principle:

Optimize for the ordinary serial execution, at the expense of some additional overhead in steals.

Division of Labor

The work-first principle guides the division of the Cilk runtime system between the compiler and the runtime library.

- The compiler implements optimized fast paths for execution of functions when no steals have occurred (i.e., no actual parallelism has been realized).
- The runtime library handles slow paths of execution, e.g., when a steal occurs.

Division of Labor

The work-first principle guides the division of the Cilk runtime system between the compiler and the runtime library.

- The compiler implements optimized fast paths for execution of functions when no steals have occurred (i.e., no actual parallelism has been realized).
- The runtime library handles slow paths of execution, e.g., when a steal occurs.

Examples:

- The THE protocol
- The implementation of **`cilk_spawn`** and **`cilk_sync`**
- The organization of full frames vs Cilk stack frames

Choice of Whom / What to Steal

Classic randomized work-stealing:
Steal from a randomly chosen victim and steal from the top of its deque.

Choice of Whom / What to Steal

Classic randomized work-stealing:
Steal from a randomly chosen victim and steal from the top of its deque.

- The random choice and stealing from top allow us to amortize the cost of steals against the span term.

Choice of Whom / What to Steal

Classic randomized work-stealing:
Steal from a randomly chosen victim and steal from the top of its deque.

- The random choice and stealing from top allow us to amortize the cost of steals against the span term.
- Randomness also avoids contention.

Choice of Whom / What to Steal

Classic randomized work-stealing:
Steal from a randomly chosen victim and steal from the top of its deque.

- The random choice and stealing from top allow us to amortize the cost of steals against the span term.
- Randomness also avoids contention.
- An old performance bug in the runtime: every worker had a random number generator initialized with the same seed, which leads to high contention because everyone chose the same sequence of victims.

Spawn Semantics

Continuation-stealing (work-first): execute the spawned child and prepare the continuation to be stolen.

```
int foo(int n) {
  int x, y;
  cilk_scope {
    x = cilk_spawn bar(n);
    y = baz(n);
  }
  return x + y;
}
```

Spawn Semantics

Continuation-stealing (work-first): execute the spawned child and prepare the continuation to be stolen.

Child-stealing (help-first): push the spawned child onto the deque so it can be stolen and continue executing the spawning function. Pop off spawned children to execute when encountering a sync.

```
int foo(int n) {
  int x, y;
  cilk_scope {
    x = cilk_spawn bar(n);
    y = baz(n);
  }
  return x + y;
}
```

Issues with Child-Stealing: Space

```
cilk_scope {
    for(int i=0; i<1000; i++) {
        cilk_spawn foo(i);
    }
}
```

Child-stealing: create 1000 work items and push them onto the deque before start doing any work!

Continuation-stealing: work on the spawned iteration and let the rest of the loops to be stolen potentially.

Continuation-Stealing vs Child-Stealing

Continuation-stealing:

- Bounded space utilization.
- Better work-efficiency.
- One-worker execution follows that of serial projection.
- For private caches, one can bound the cache misses during parallel executions.

Child-stealing:

- Potentially unbounded space utilization.
- Worse work-efficiency.
- One-worker execution does NOT follow that of serial projection.
- No proven bound on cache misses during parallel executions.

Continuation-Stealing vs Child-Stealing

Continuation-stealing:

- Bounded space utilization.
- Better work-efficiency.
- For private caches, one can bound the cache misses during parallel executions.

Child-stealing:

- Potentially unbounded space utilization.
- Worse work-efficiency.
- No proven bound on cache misses during parallel executions.

Only monsters steal children!