# Algorithm Engineering
# (aka. How to Write Fast Code)

CS260 – Lecture 8
Yan Gu

## What is Parallelism and Scheduling

Many slides in this lecture are borrowed from the seventh lecture in 6.172 Performance Engineering of Software Systems at MIT. The credit is to Prof. Charles E. Leiserson, and the instructor appreciates the permission to use them in this course.

# CS260: Algorithm Engineering Lecture 8

- Fork-Join Parallelism
- Greedy Scheduler
- Work-Stealing Scheduler

# Recall: Basics of Cilk

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

The named ***child*** function may execute in parallel with the ***parent*** caller.

Control cannot pass this point until all spawned children have returned.

- Cilk keywords grant permission for parallel execution. They do not command parallel execution.

# Execution Model

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

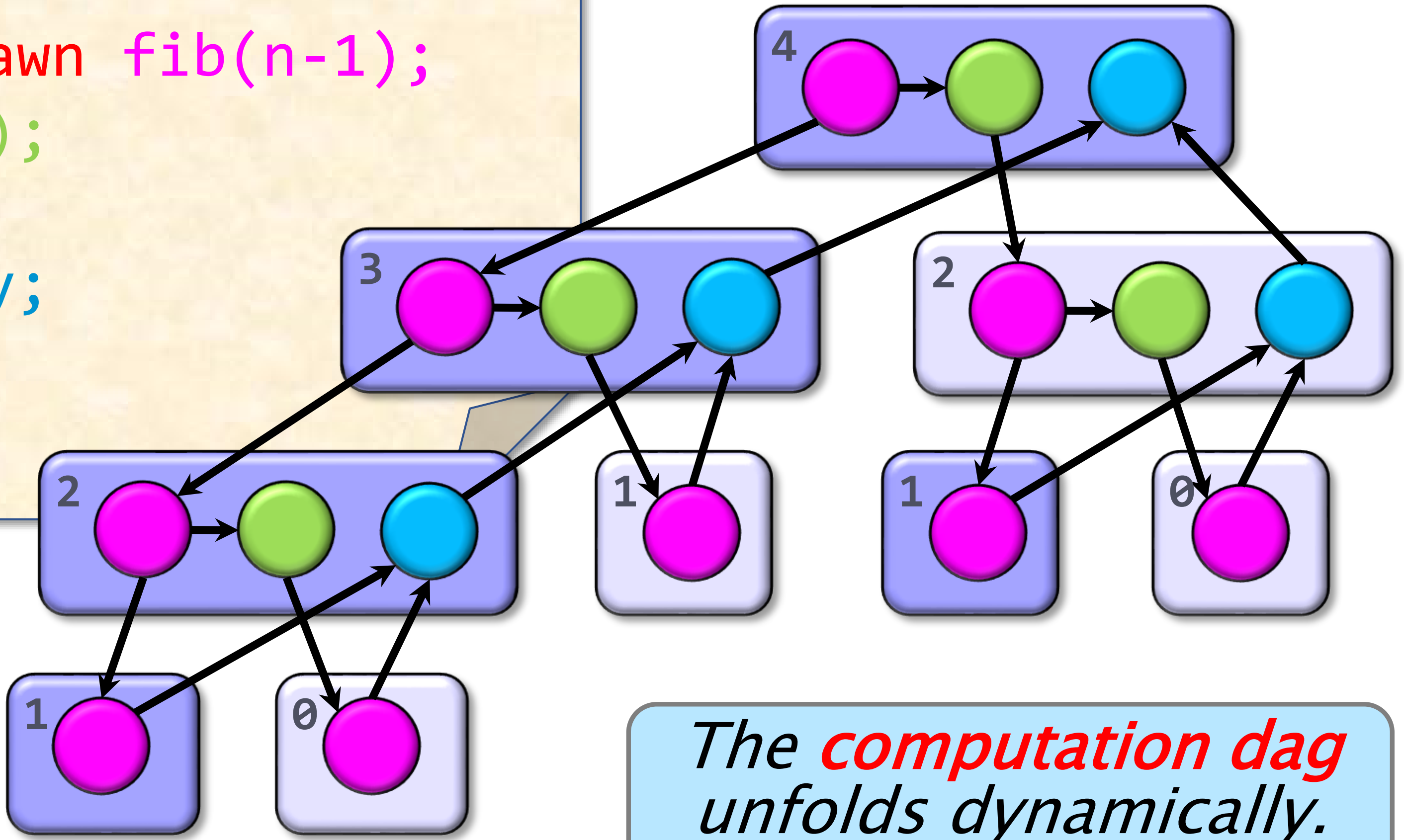
**Example:**

`fib(4)`

# Execution Model

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

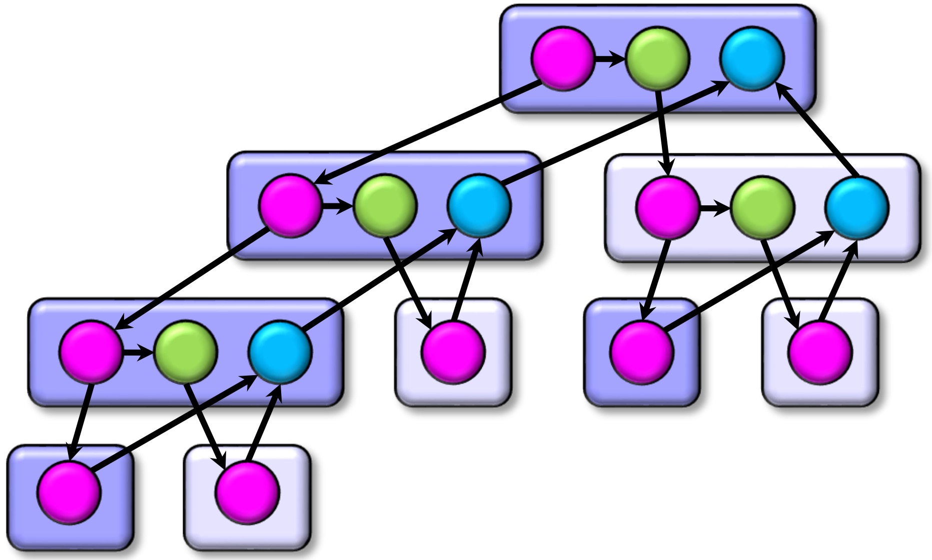
Example:
`fib(4)`




"Processor oblivious"

*The **computation dag** unfolds dynamically.*

# How Much Parallelism?

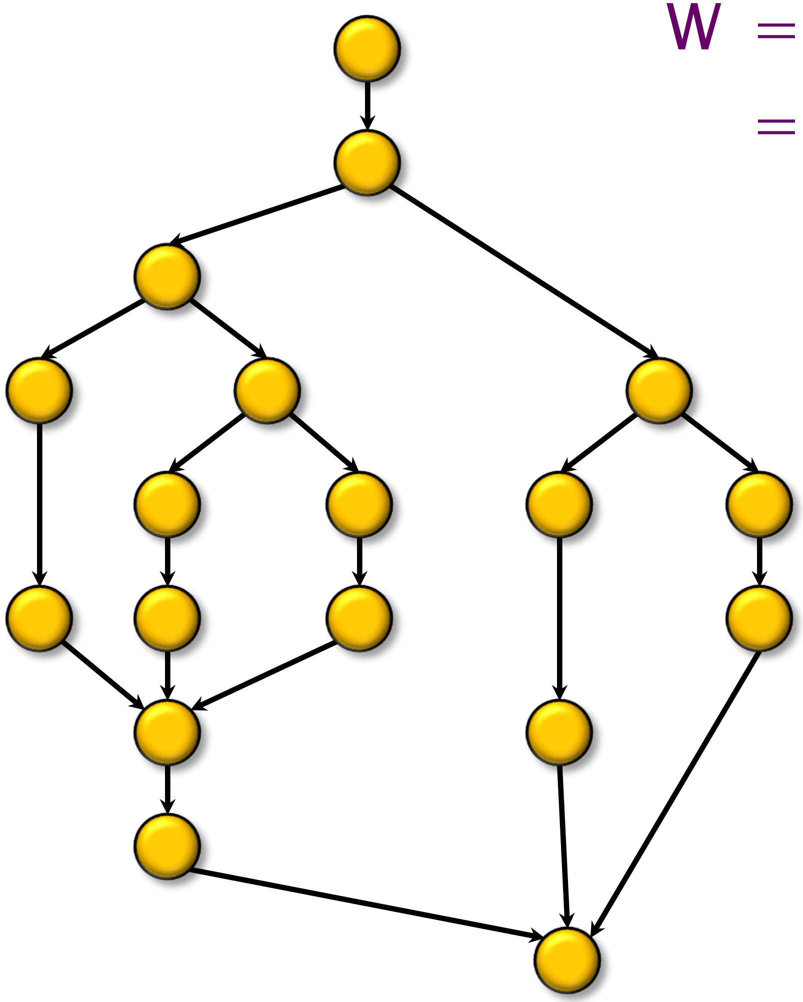


Loop parallelism (`cilk_for`) is converted to spawns and syncs using recursive divide-and-conquer.

Assuming that each node executes in unit time, what is the **parallelism** of this computation?

# Performance Measures

$T_P$ = execution time on $P$ processors

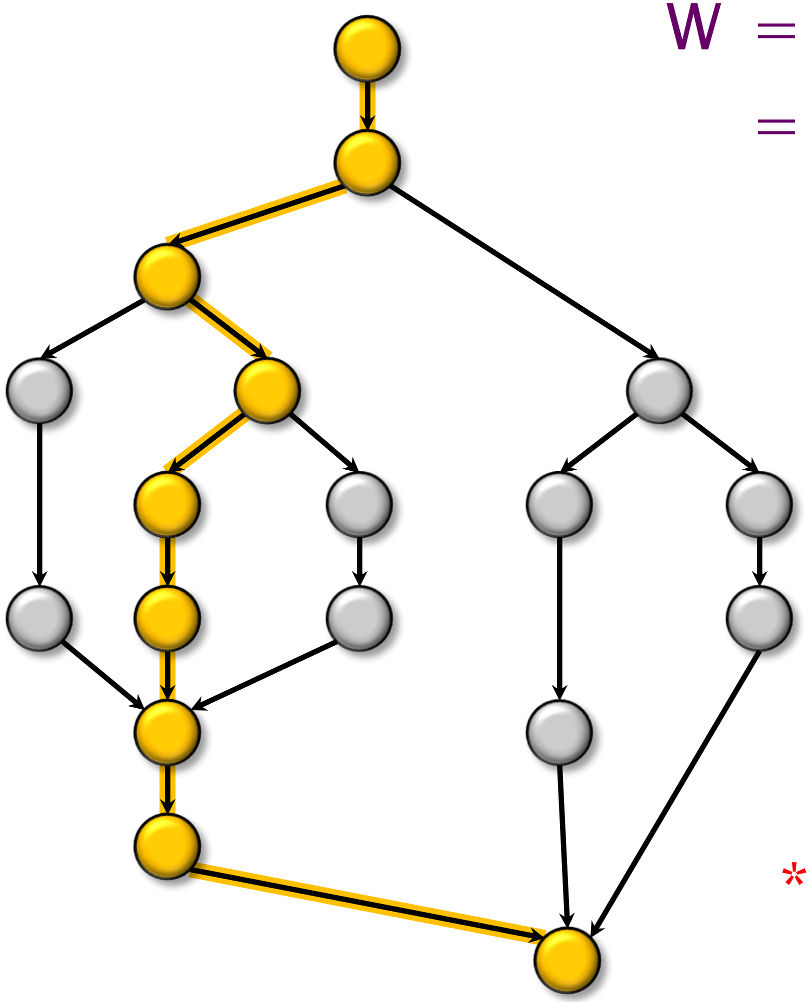


$W$ = work

= 18

# Performance Measures

$T_P$ = execution time on P processors



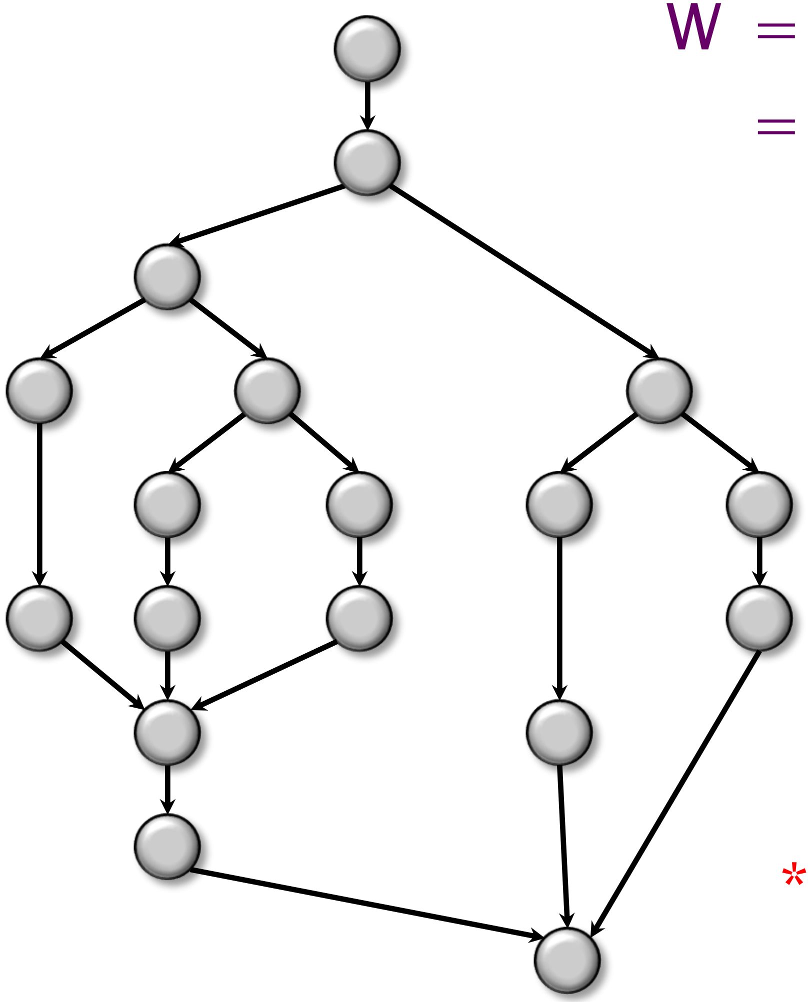
$T_1$ = work = 18

$T_\infty$ = span* = 9

*Also called critical-path length or computational depth.

# Performance Measures

$T_P$ = execution time on P processors



$T_1$ = work = 18

$T_\infty$ = span* = 9

**Work Law**

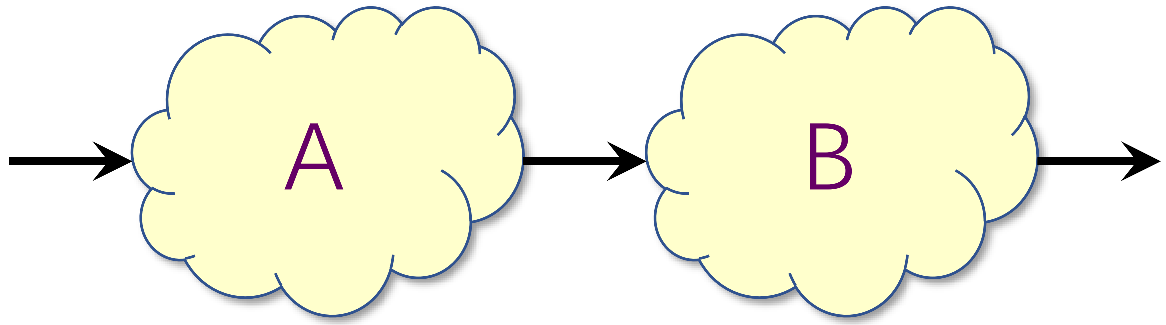
- $T_P \geq T_1/P$

**Span Law**

- $T_P \geq T_\infty$

*Also called critical-path length or computational depth.

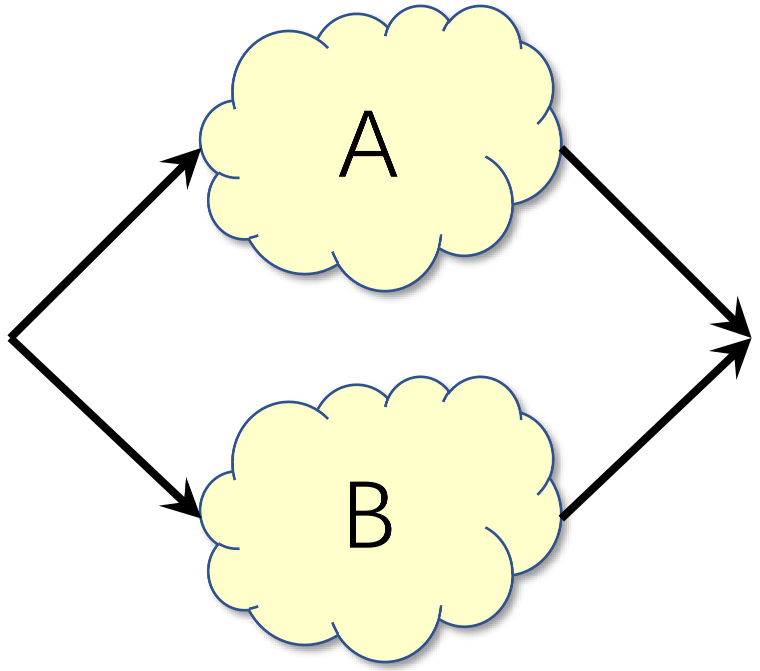
# Series Composition




*Work:* $W(A \cup B) = W(A) + W(B)$

*Span:* $D(A \cup B) = D(A) + D(B)$

# Parallel Composition




*Work:* $W(A \cup B) = W(A) + W(B)$

*Span:* $D(A \cup B) = \max\{D(A), D(B)\}$

# Speedup

Definition. $W/T =$ speedup on $P$ processors.

- If $W/T < P$, we have sublinear speedup.
- If $W/T = P$, we have (perfect) linear speedup.
- If $W/T > P$, we have superlinear speedup, which is not possible in this simple performance model, because of the WORK LAW $T \geqslant W/P$.

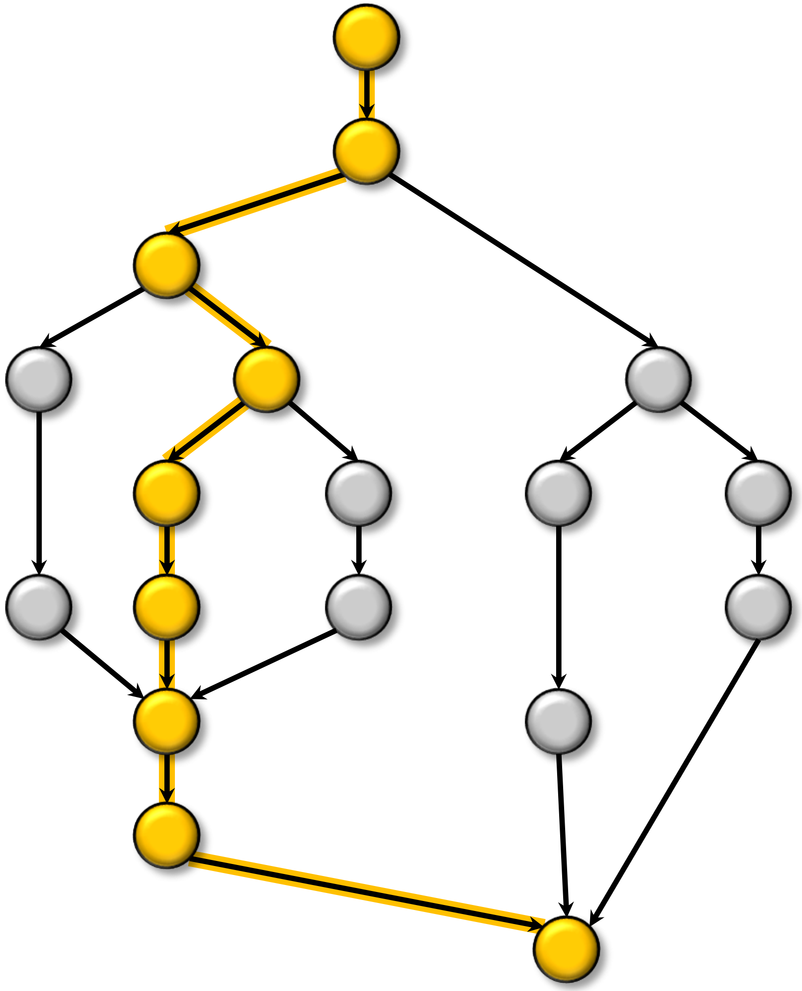
# Parallelism

**Because the SPAN LAW dictates that $T \geq D$, the maximum possible speedup given W and D is**

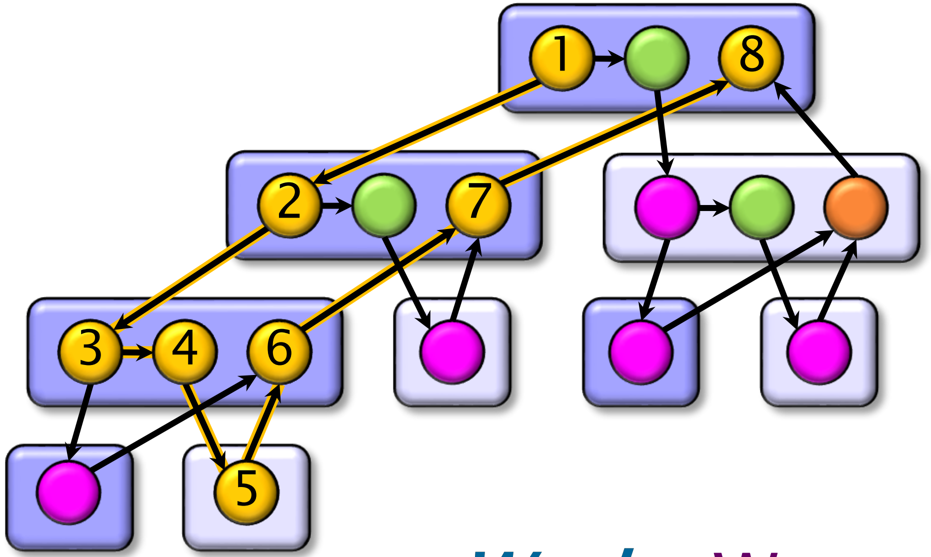
**W/D = parallelism**

**= the average amount of work per step along the span**

**= 18/9 = 2**



# Example: fib(4)




Assume for simplicity that each strand in fib(4) takes unit time to execute.

*Work:* W = 17

*Span:* D = 8

*Parallelism:* W/D = 2.125

Using many more than 2 processors can yield only marginal performance gains.

# CS260: Algorithm Engineering Lecture 8

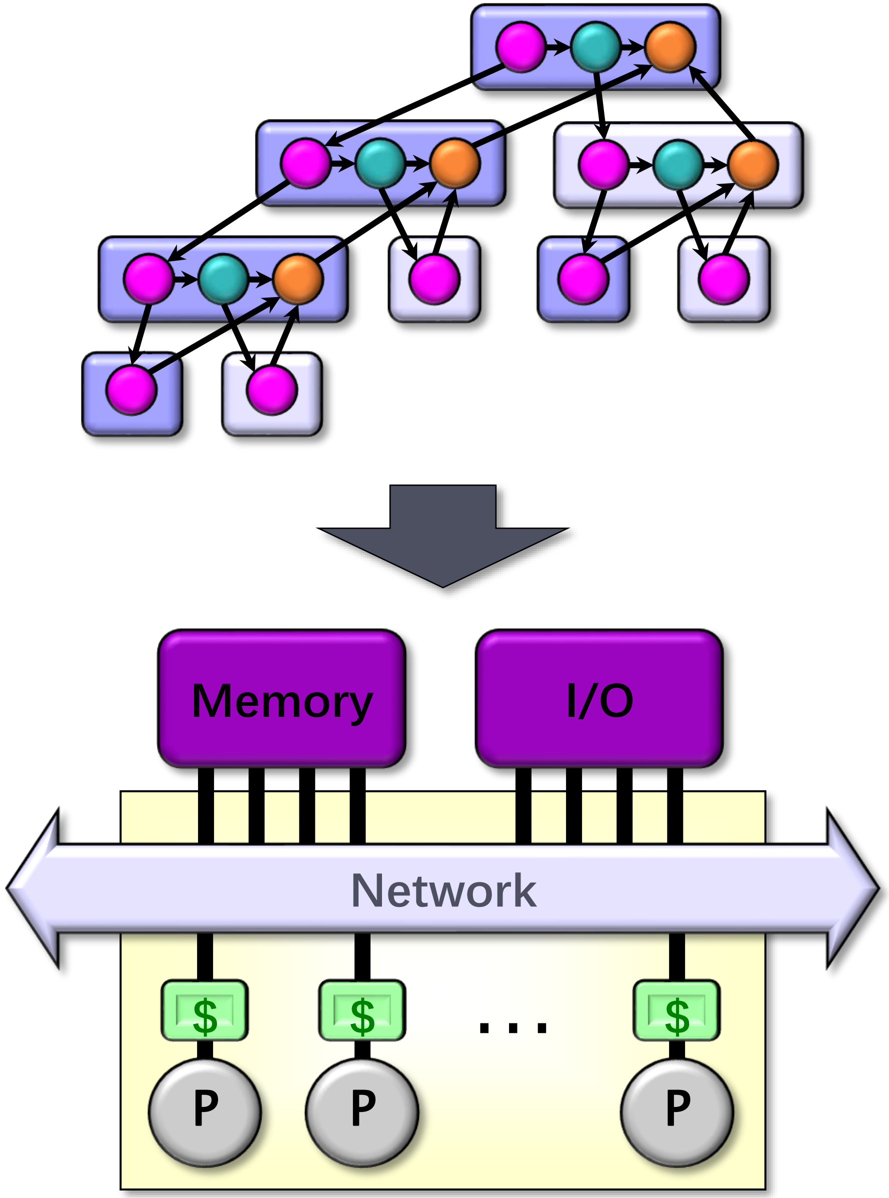
Fork-Join Parallelism

Greedy Scheduler

Work-Stealing Scheduler

# Scheduling

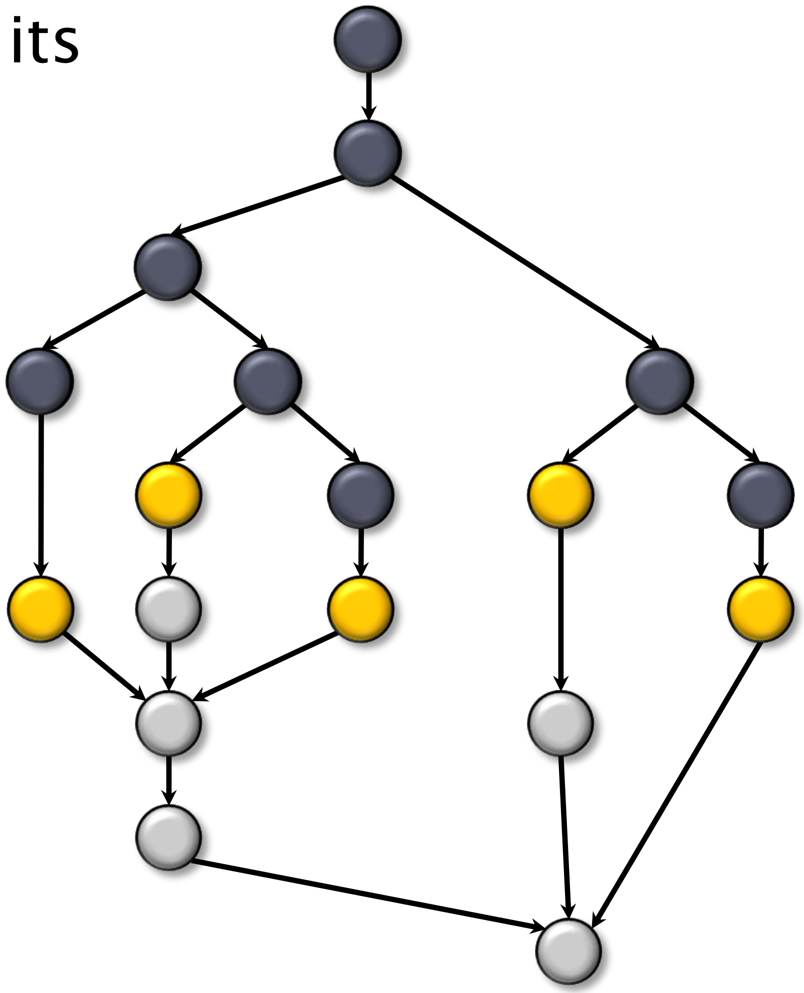
- Fork–Join parallelism allows the programmer to express potential parallelism in an application
- The scheduler maps strands onto processors dynamically at runtime
- Since the theory of distributed schedulers is complicated, we'll first explore the ideas with a centralized scheduler



# Greedy Scheduling

**IDEA: Do as much as possible on every step.**

Definition. A node is ready if all its predecessors have executed.



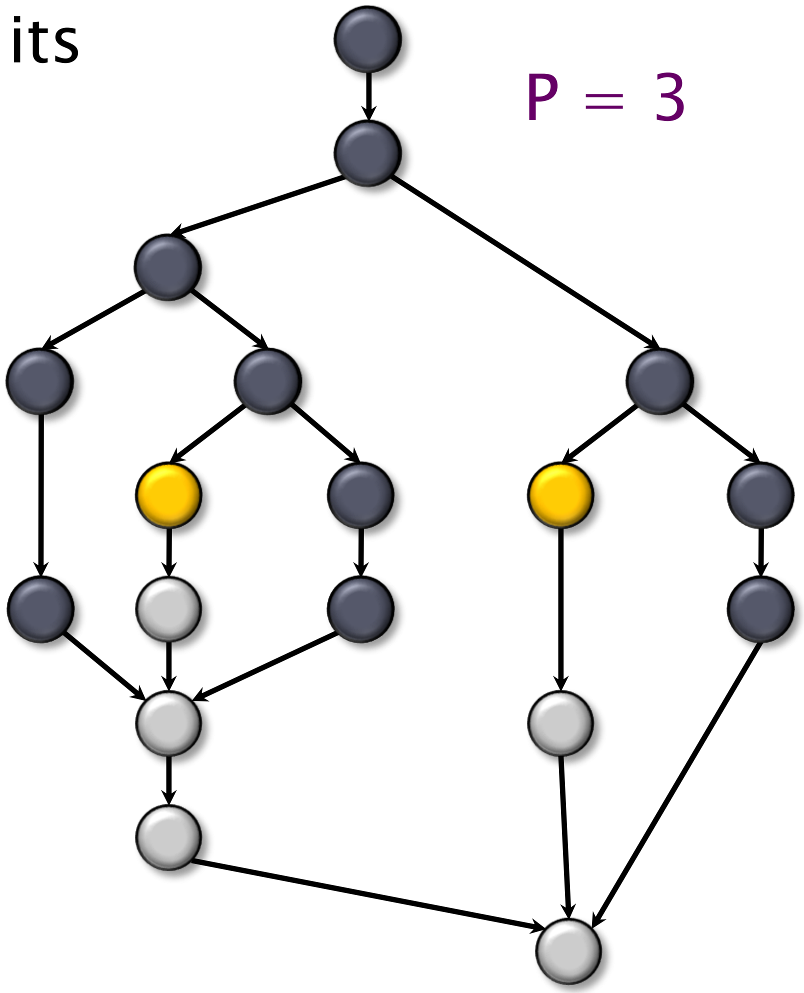
# Greedy Scheduling

**IDEA:** Do as much as possible on every step.

**Definition.** A node is ready if all its predecessors have executed.

## Complete step

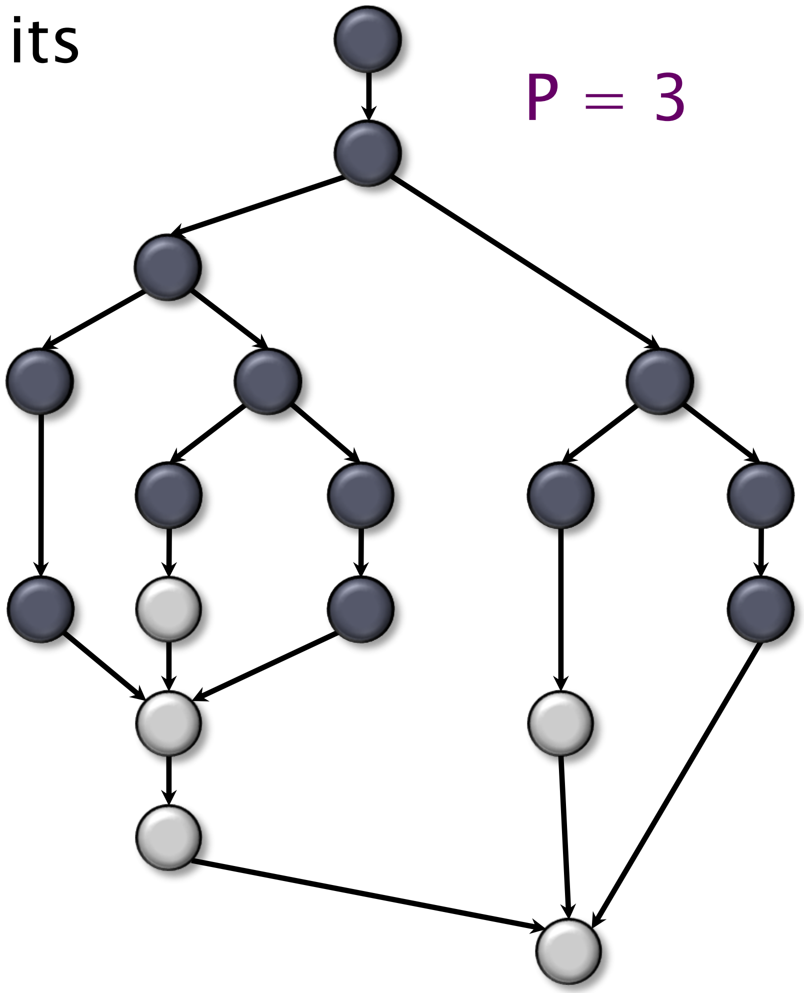
- $\geqslant P$ strands ready.
- Run any $P$.



# Greedy Scheduling

IDEA: Do as much as possible on every step.

**Definition.** A node is *ready* if all its predecessors have executed.




## Complete step

- $\geqslant P$ strands ready.
- Run any $P$.

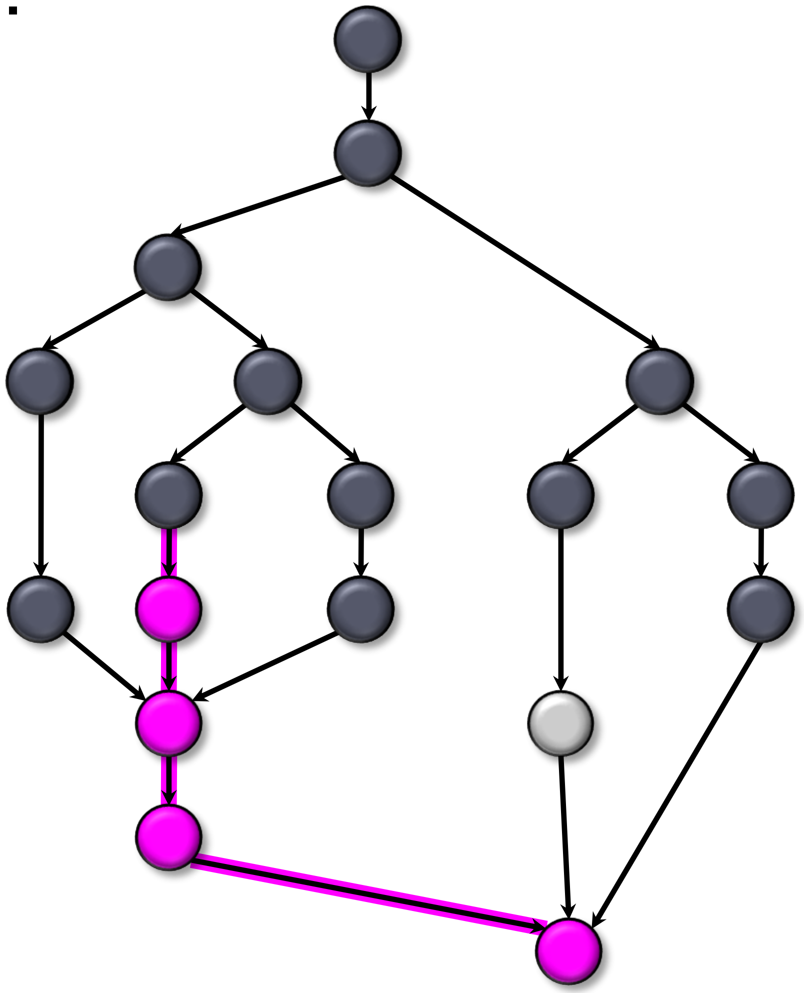
## Incomplete step

- $< P$ strands ready.
- Run all of them.

# Analysis of Greedy

Theorem [G68, B75, EZL89]. Any greedy scheduler achieves

$$T \leq W/P + D.$$



*Proof.*

- \# complete steps $\leqslant W/P$, since each complete step performs $P$ work.
- \# incomplete steps $\leqslant D$, since each incomplete step reduces the span of the unexecuted dag by $1$. ■

# CS260: Algorithm Engineering Lecture 8

Fork-Join Parallelism

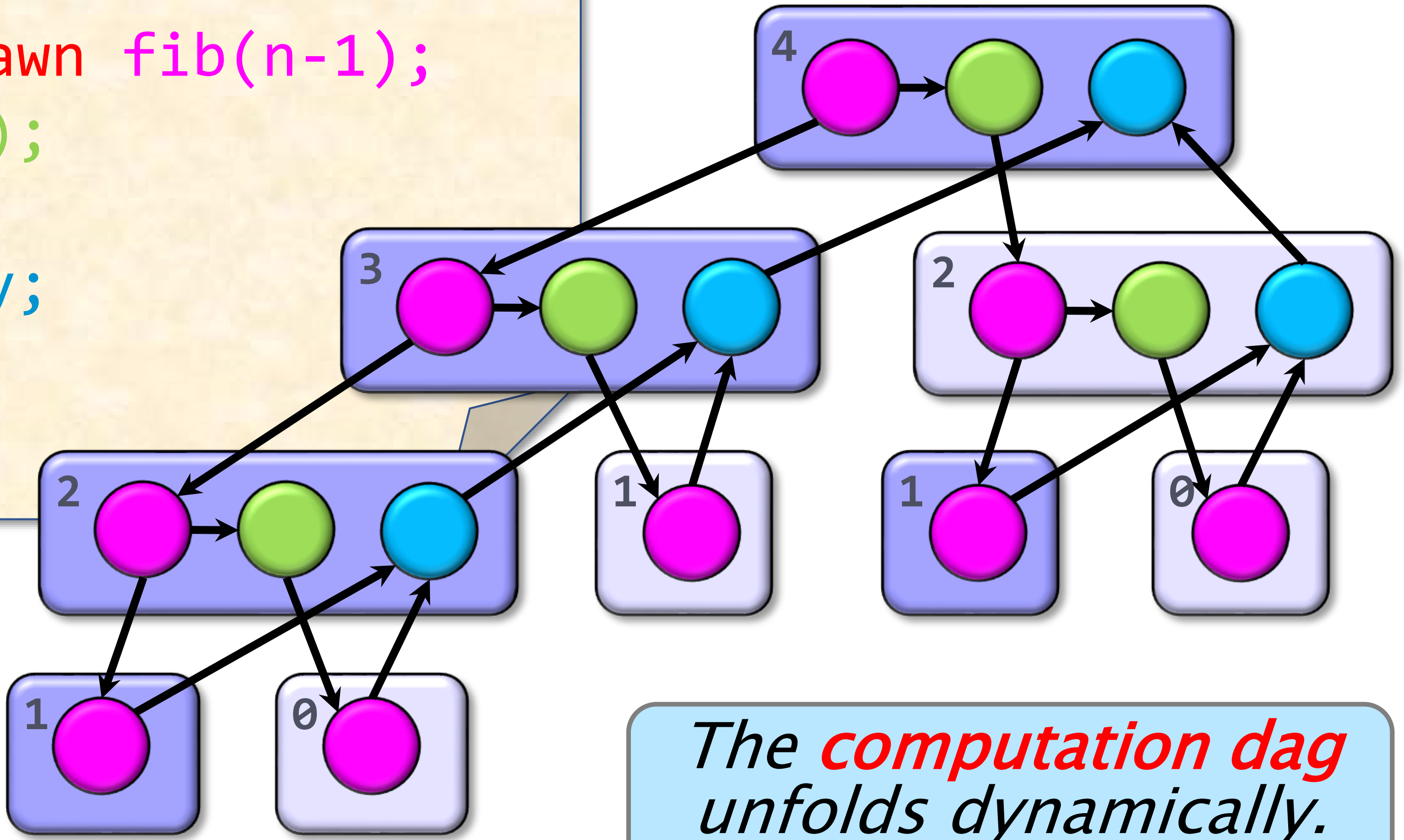
Greedy Scheduler

Work-Stealing Scheduler

# Execution Model

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

Example:
`fib(4)`



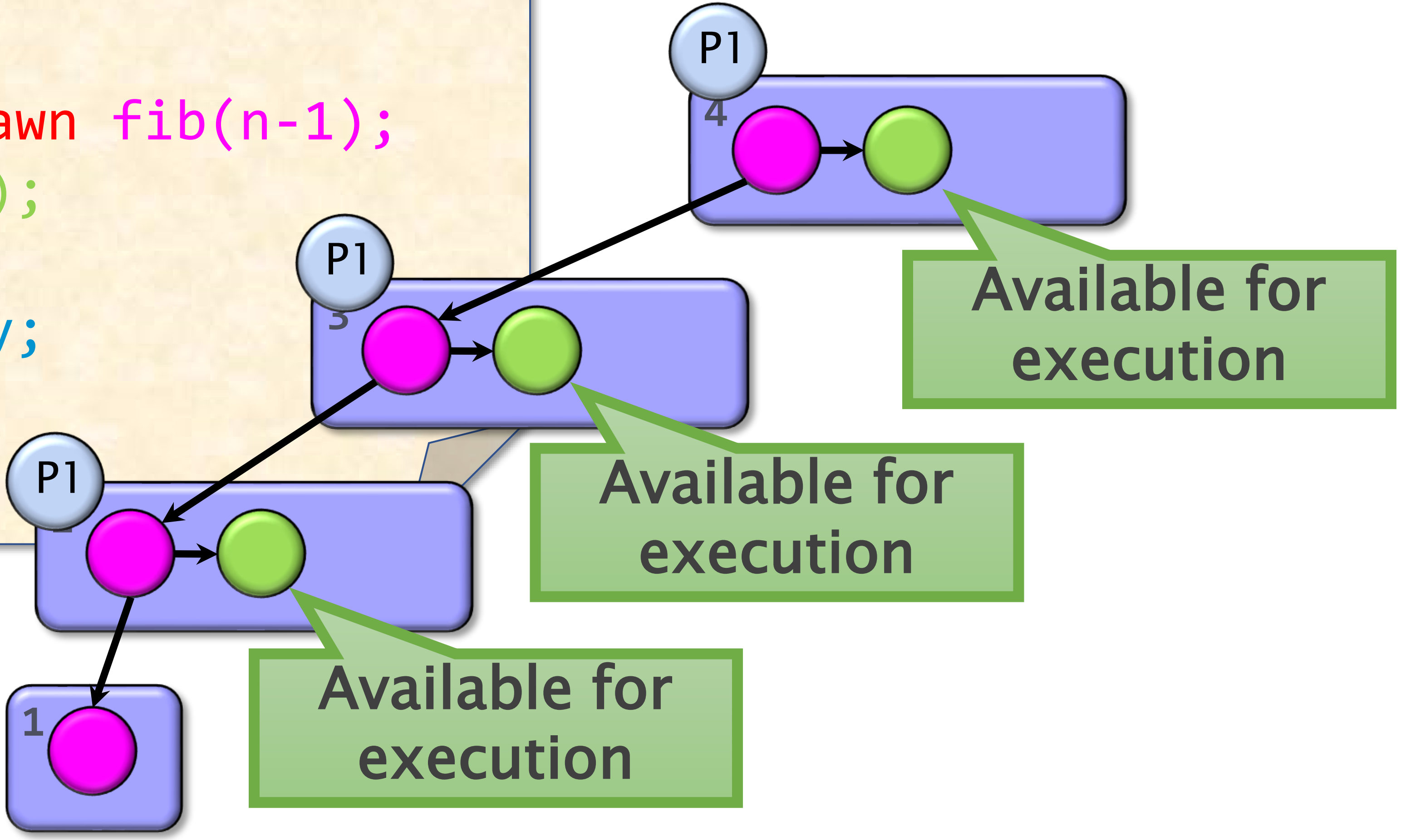

"Processor oblivious"

*The **computation dag** unfolds dynamically.*

# Execution Model

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

Example:
`fib(4)`

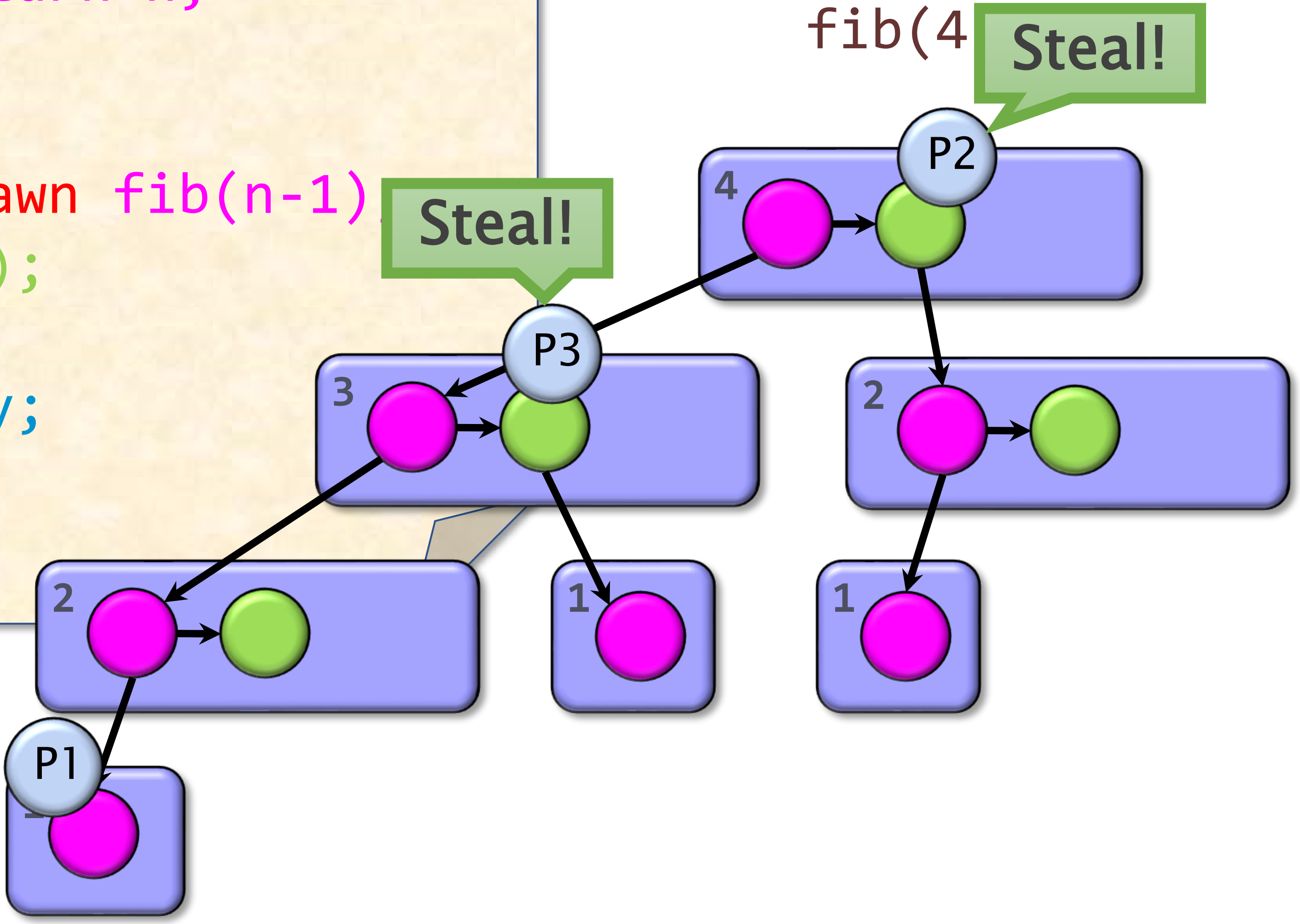


# Execution Model

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

Example:

fib(4

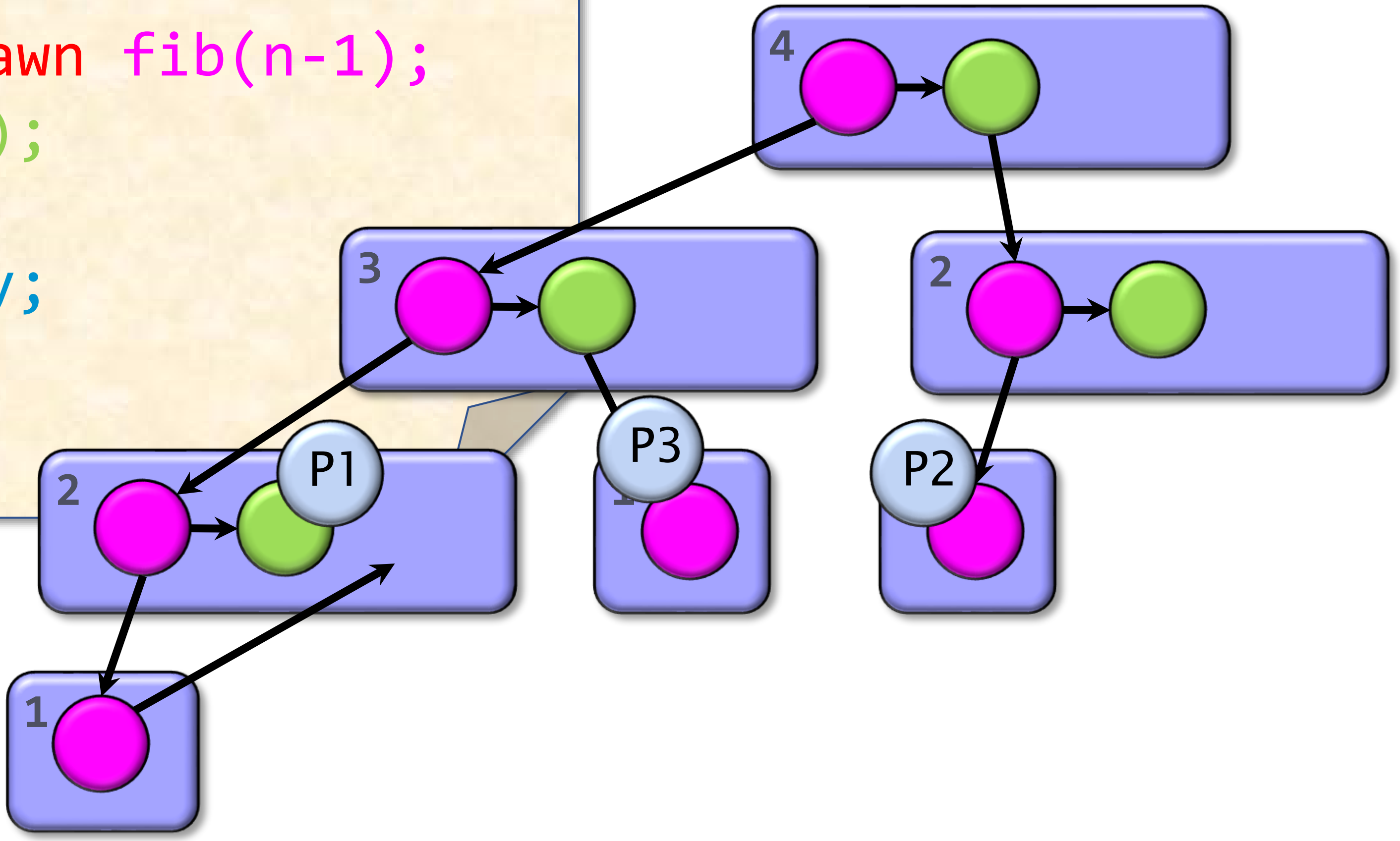


# Execution Model

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

**Example:**

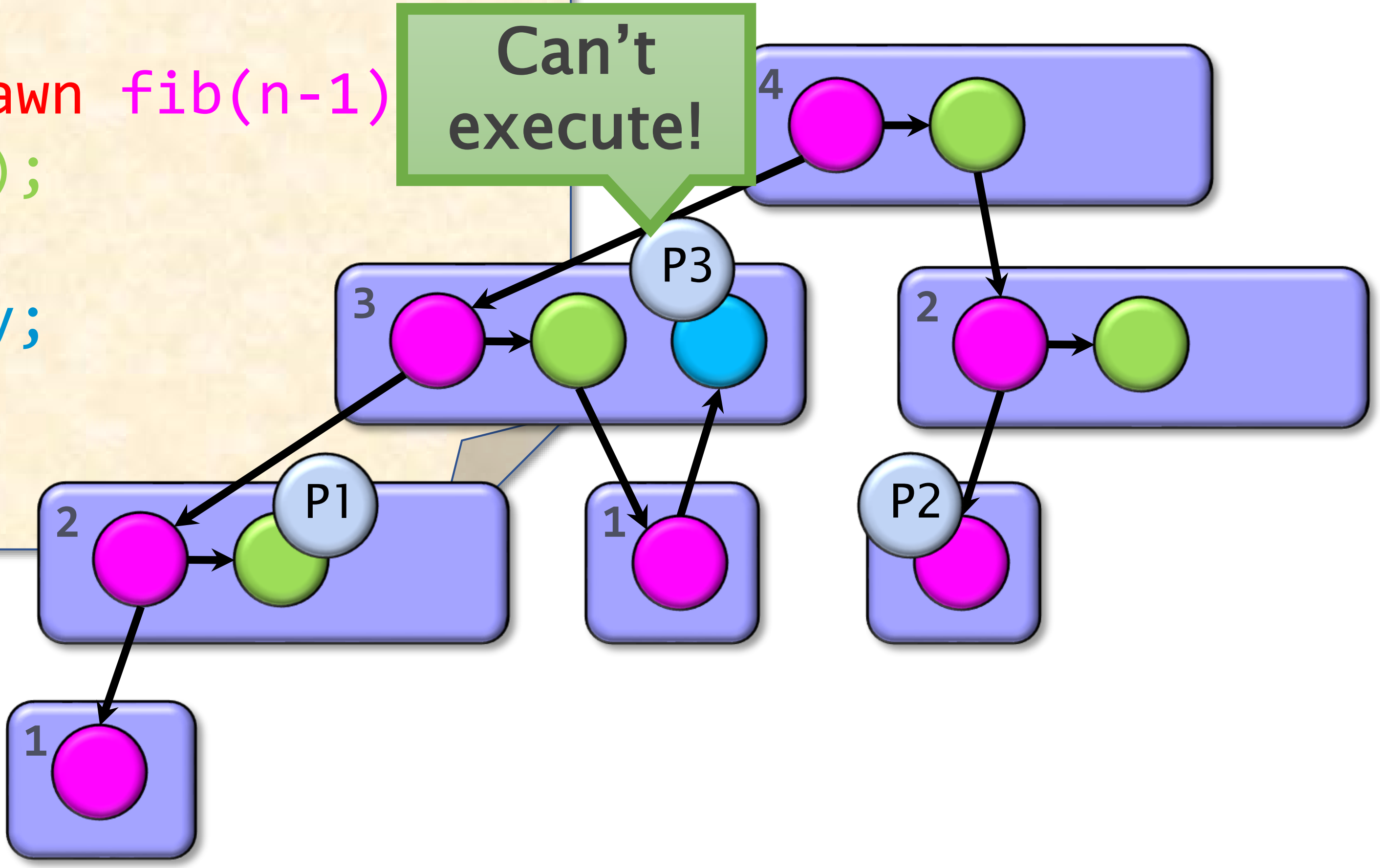
`fib(4)`



# Execution Model

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

Example:
`fib(4)`

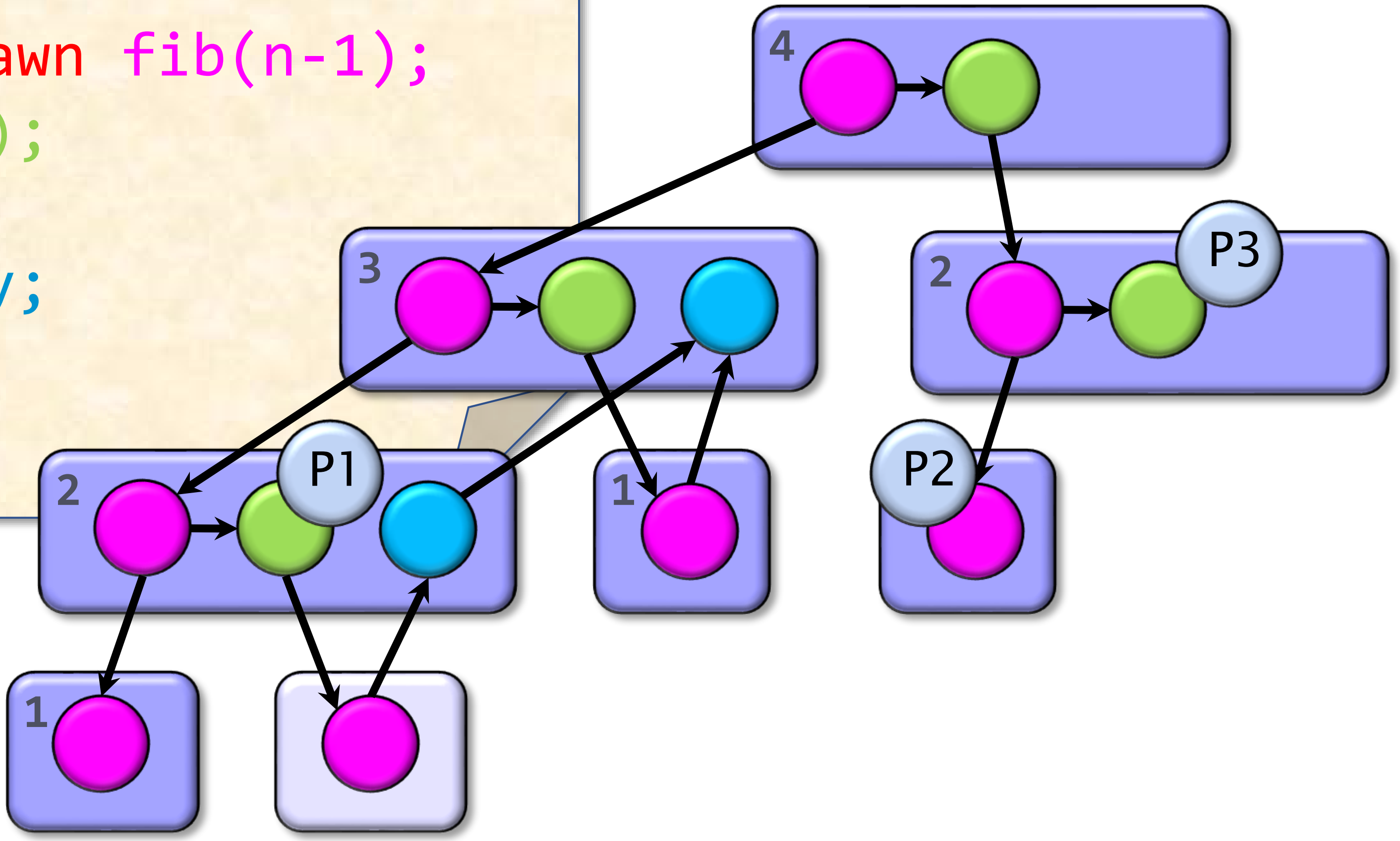


# Execution Model

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

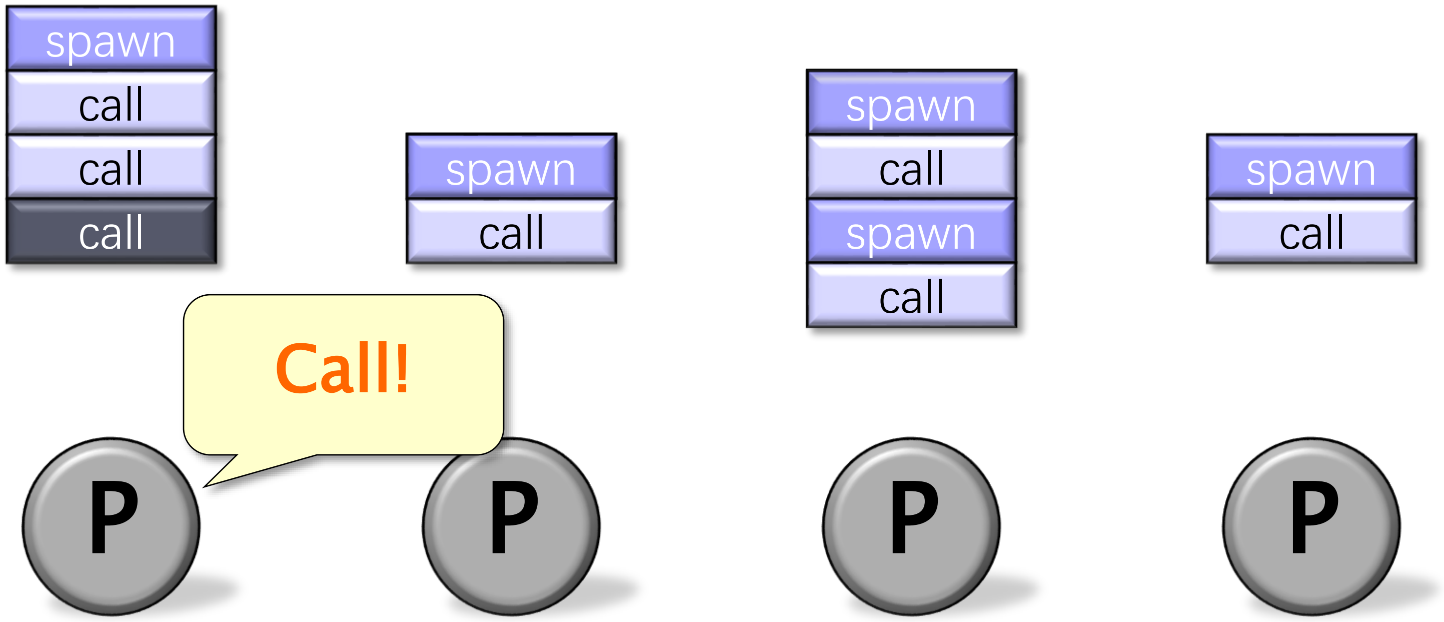
Example:

fib(4)



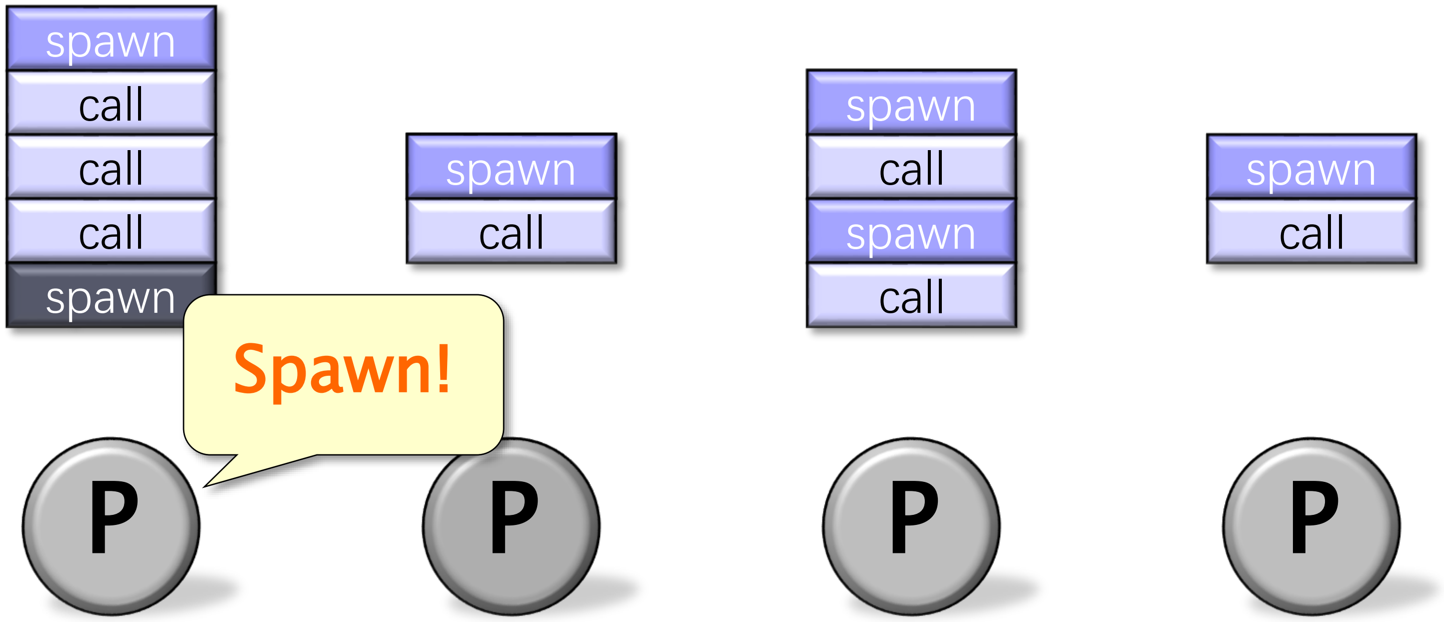
# Cilk Runtime System

Each worker (processor) maintains a work deque of ready strands, and it manipulates the bottom of the deque like a stack [MKH90, BL94, FLR98].



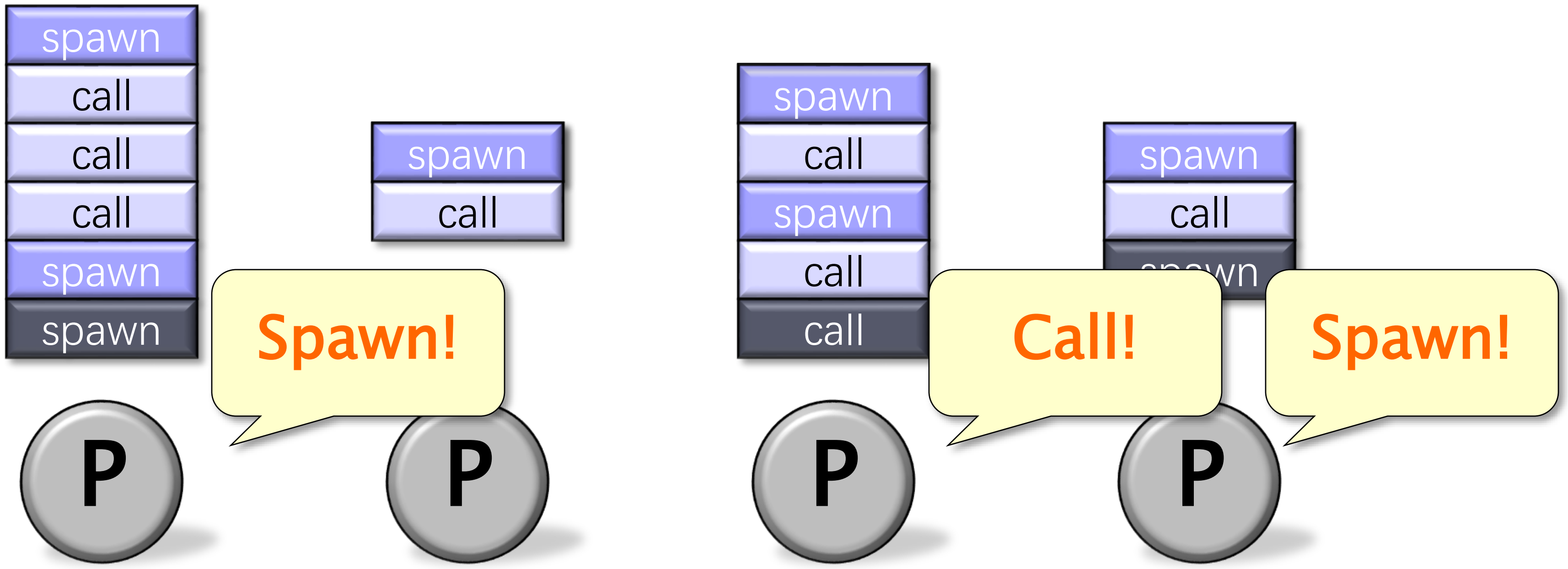
# Cilk Runtime System

Each worker (processor) maintains a work deque of ready strands, and it manipulates the bottom of the deque like a stack [MKH90, BL94, FLR98].



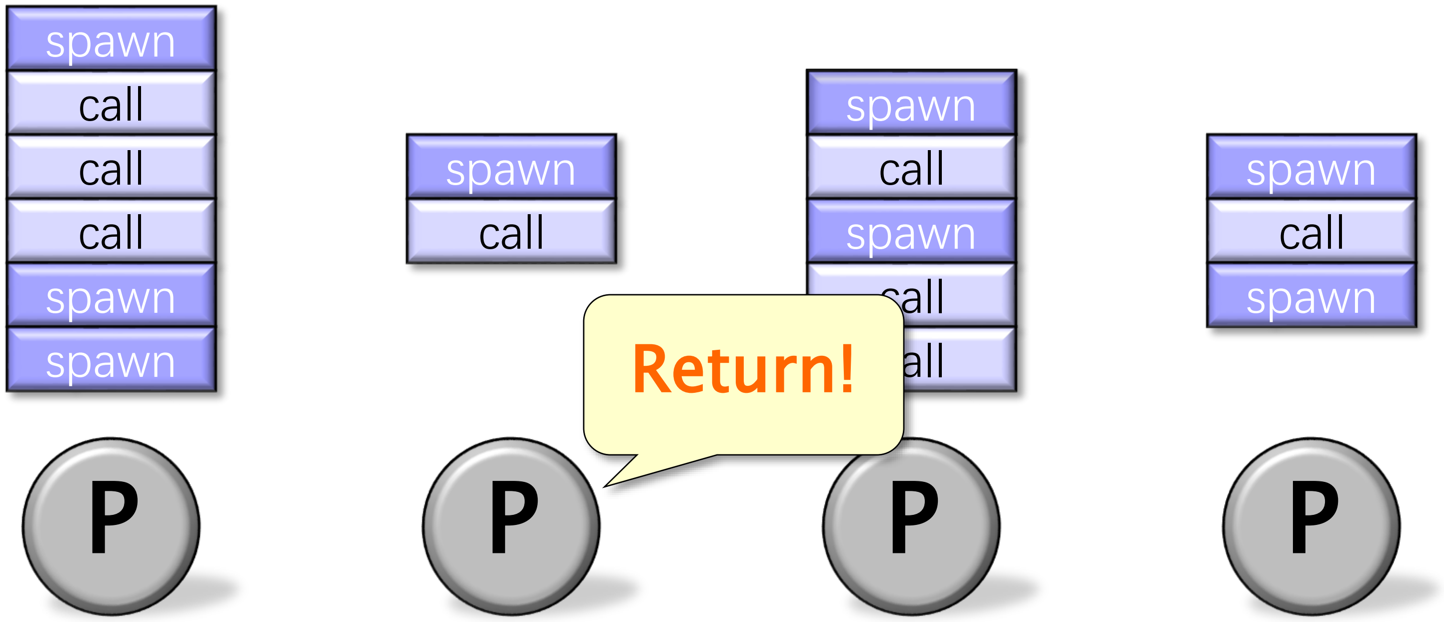
# Cilk Runtime System

Each worker (processor) maintains a work deque of ready strands, and it manipulates the bottom of the deque like a stack [MKH90, BL94, FLR98].



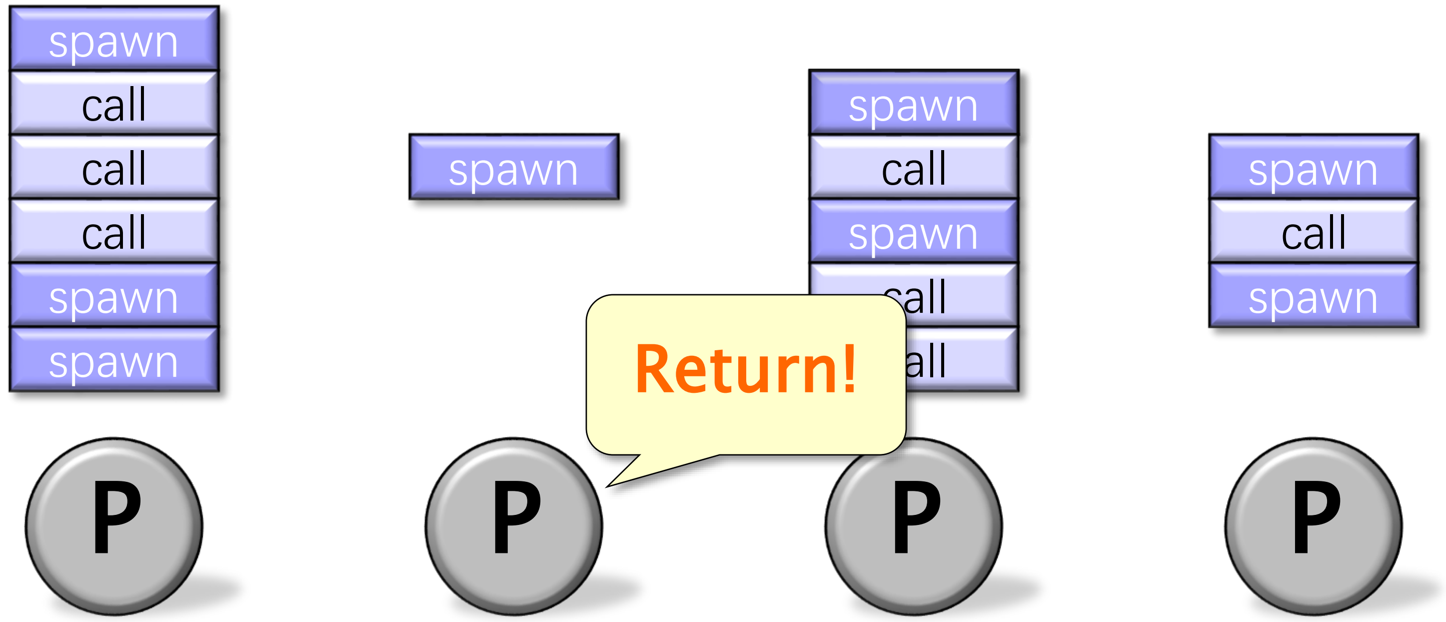
# Cilk Runtime System

Each worker (processor) maintains a work deque of ready strands, and it manipulates the bottom of the deque like a stack [MKH90, BL94, FLR98].



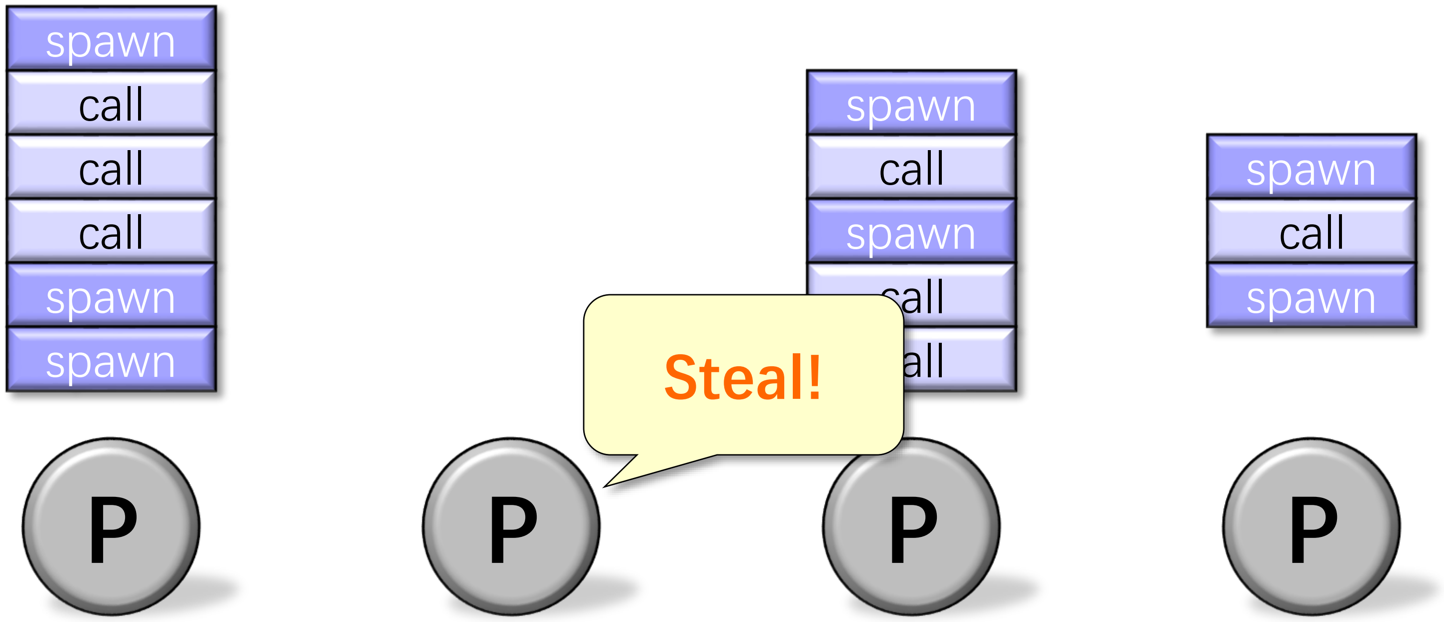
# Cilk Runtime System

Each worker (processor) maintains a work deque of ready strands, and it manipulates the bottom of the deque like a stack [MKH90, BL94, FLR98].



# Cilk Runtime System

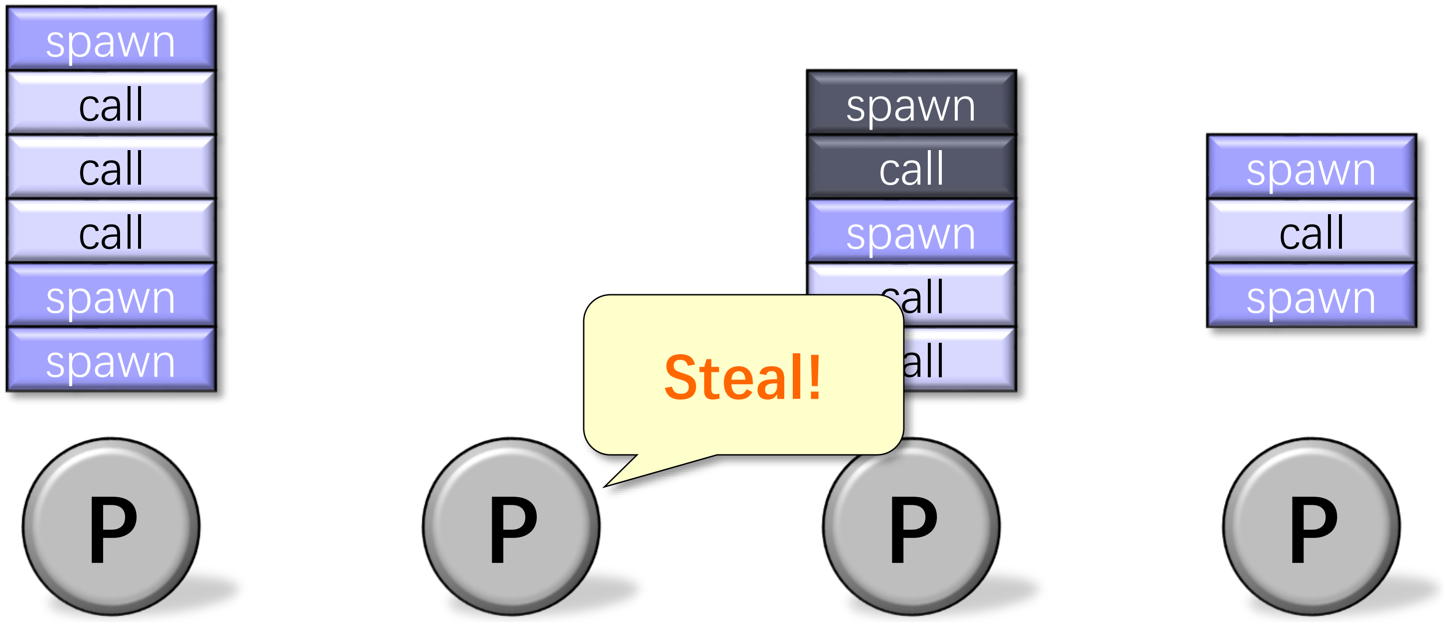
Each worker (processor) maintains a work deque of ready strands, and it manipulates the bottom of the deque like a stack [MKH90, BL94, FLR98].




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

# Cilk Runtime System

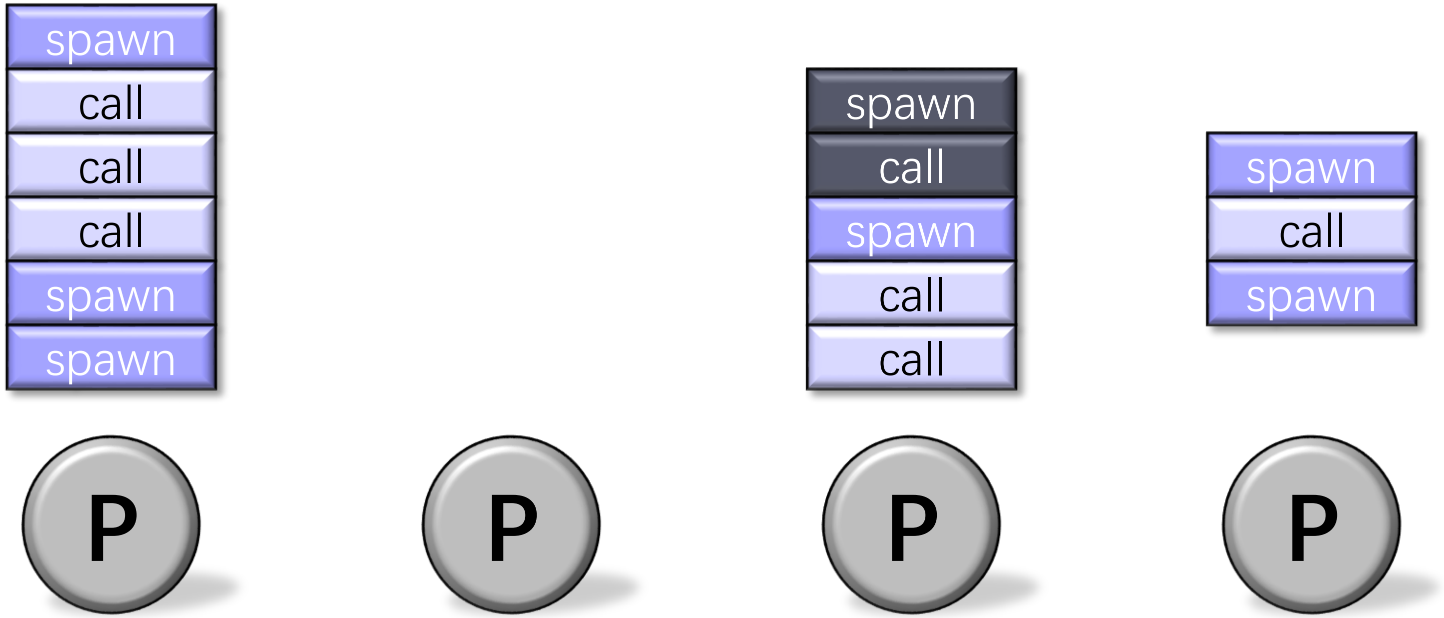
Each worker (processor) maintains a work deque of ready strands, and it manipulates the bottom of the deque like a stack [MKH90, BL94, FLR98].




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

# Cilk Runtime System

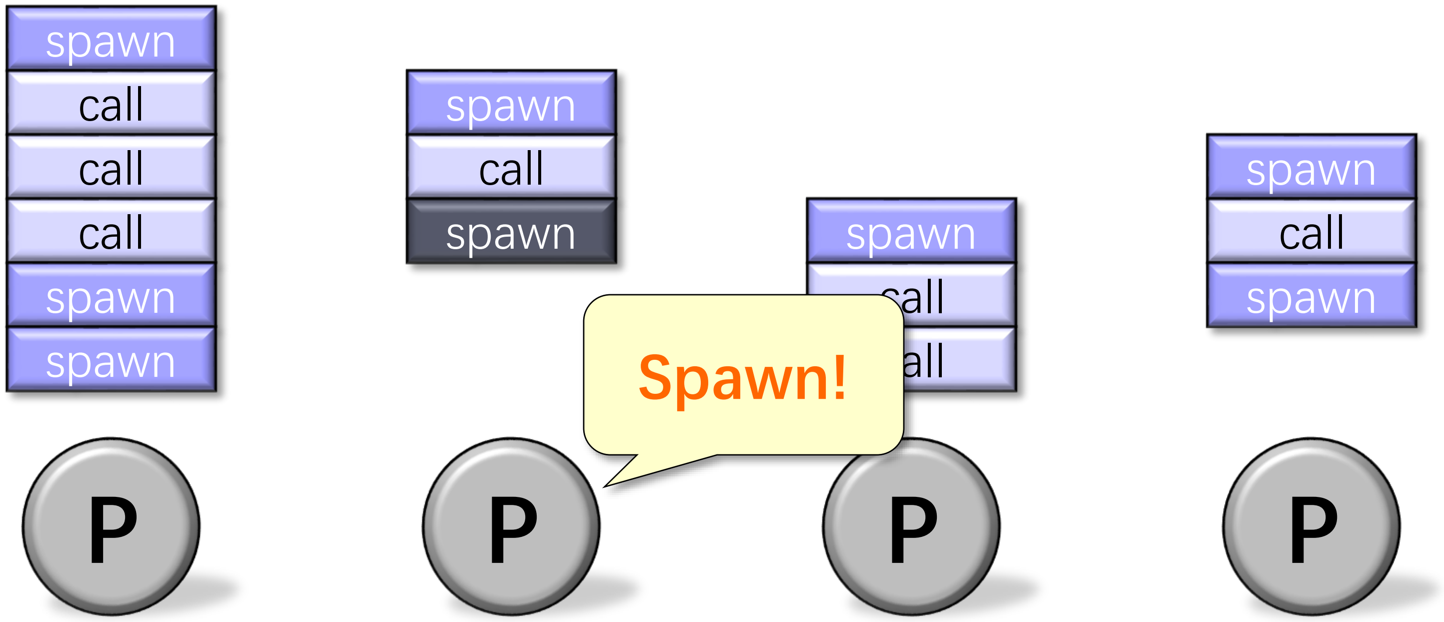
Each worker (processor) maintains a work deque of ready strands, and it manipulates the bottom of the deque like a stack [MKH90, BL94, FLR98].




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

# Cilk Runtime System

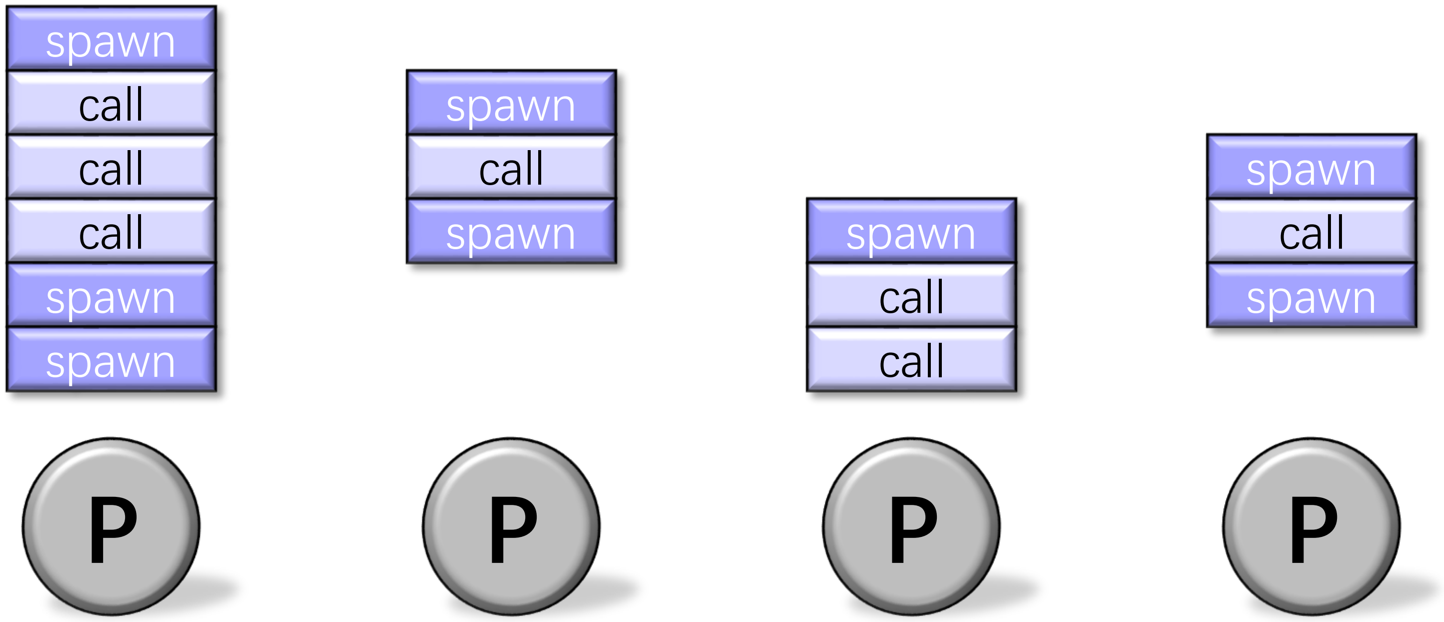
Each worker (processor) maintains a work deque of ready strands, and it manipulates the bottom of the deque like a stack [MKH90, BL94, FLR98].




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

# Cilk Runtime System

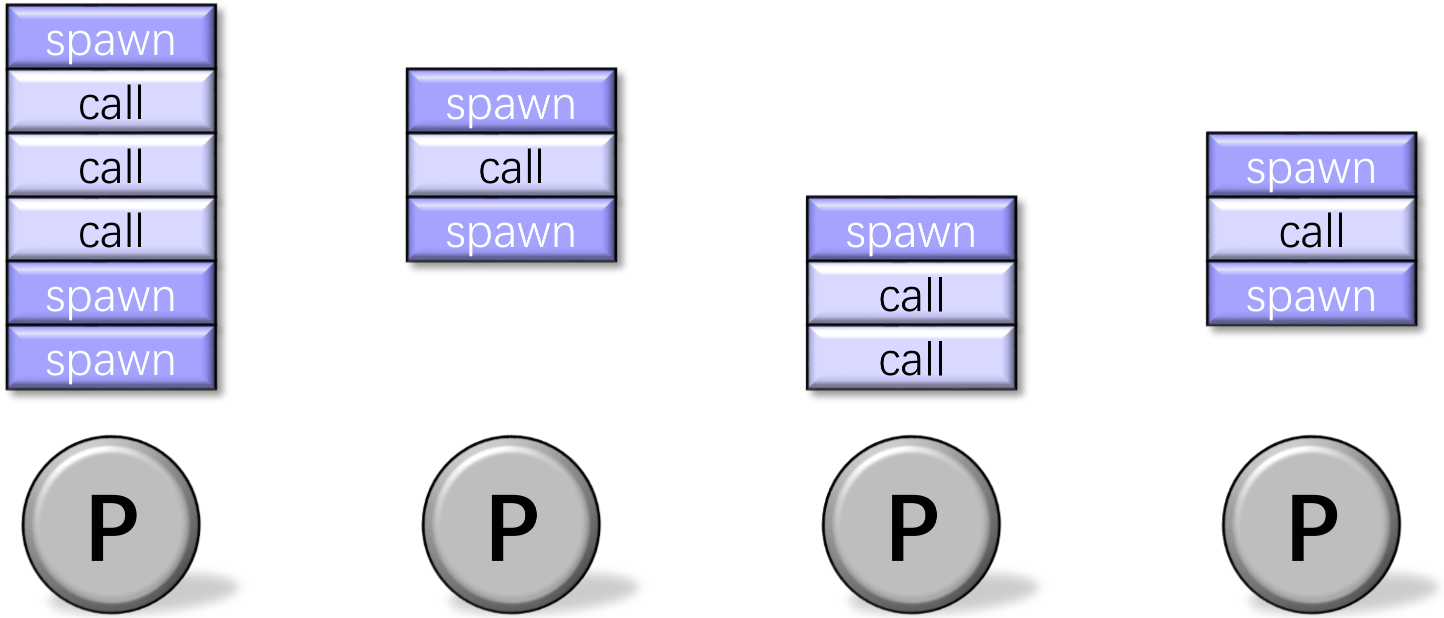
Each worker (processor) maintains a work deque of ready strands, and it manipulates the bottom of the deque like a stack [MKH90, BL94, FLR98].




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

# Cilk Runtime System

Each worker (processor) maintains a work deque of ready strands, and it manipulates the bottom of the deque like a stack [MKH90, BL94, FLR98].




[Link to Cilk implementation]

Theorem [BL94]: With sufficient parallelism, workers steal infrequently $\Rightarrow$ linear speed-up.

# Work-Stealing Bounds

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

$$T \approx W/P + O(D)$$

on $P$ processors.

*Pseudoproof.*

A processor is either working or stealing.

The total time all processors spend working is $T$.

Each steal has a $1/P$ chance of reducing the span by $1$.

Thus, the expected cost of all steals is $O(PD)$.

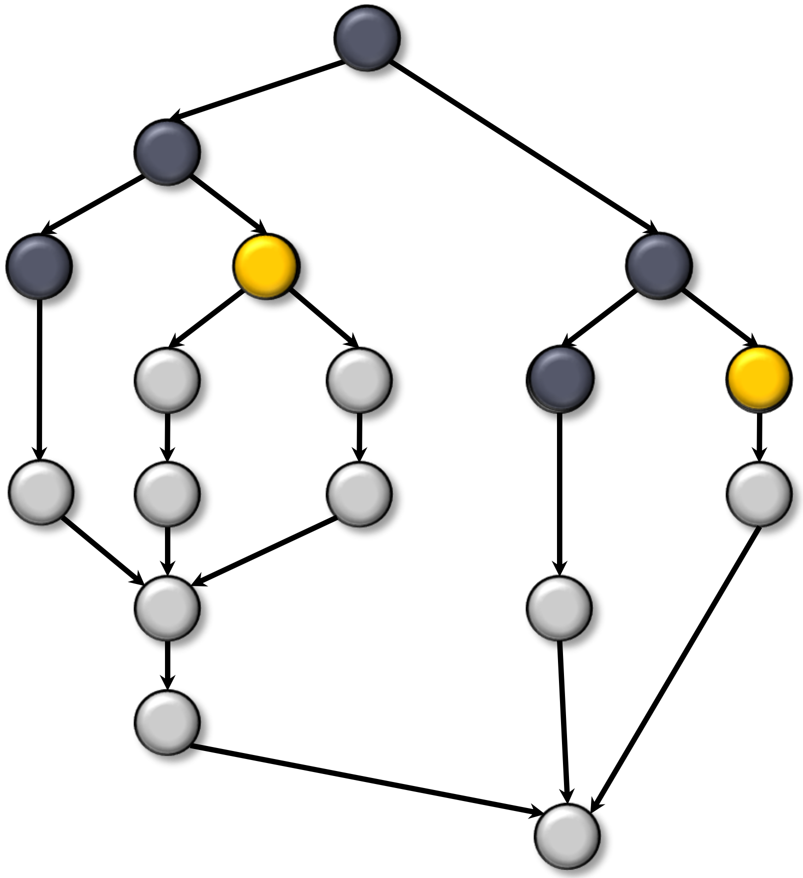
Since there are $P$ processors, the expected time is

$$(W + O(PD))/P = W/P + O(D) . \quad \blacksquare$$

# Overhead of work-stealing scheduler

Bound the number of steals (whp):

$$O(pD)$$



Running time (whp):

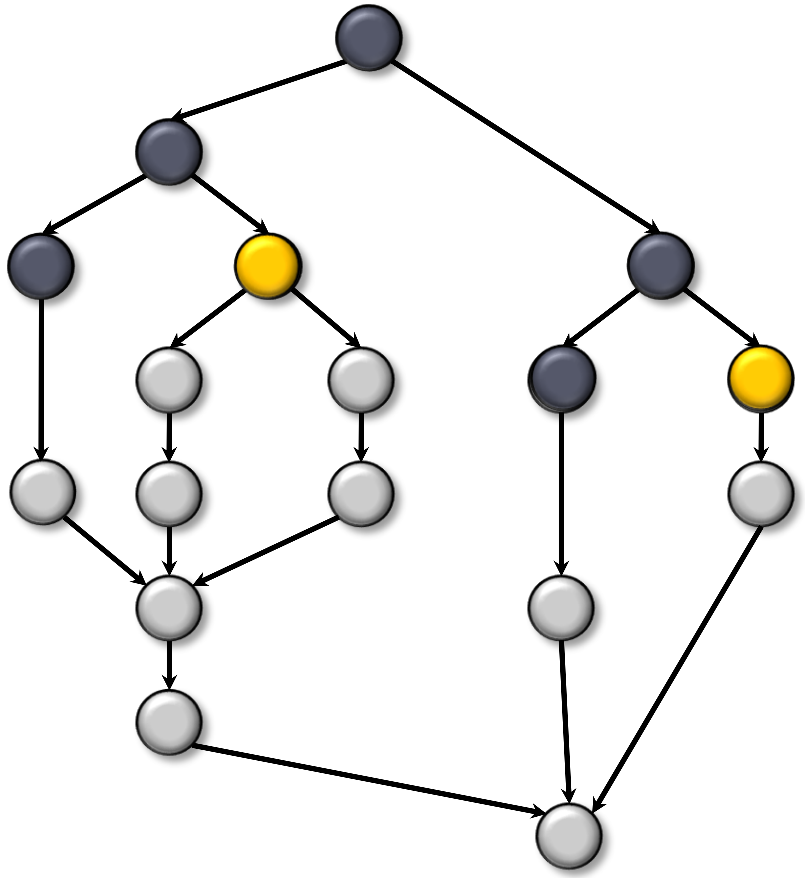
$$T = \frac{W + O(pD)}{p} = \frac{W}{p} + O(D)$$

[Link to a simple proof]

# Successful steals can be expensive

Bound the number of steals (whp):

$$O(pD)$$

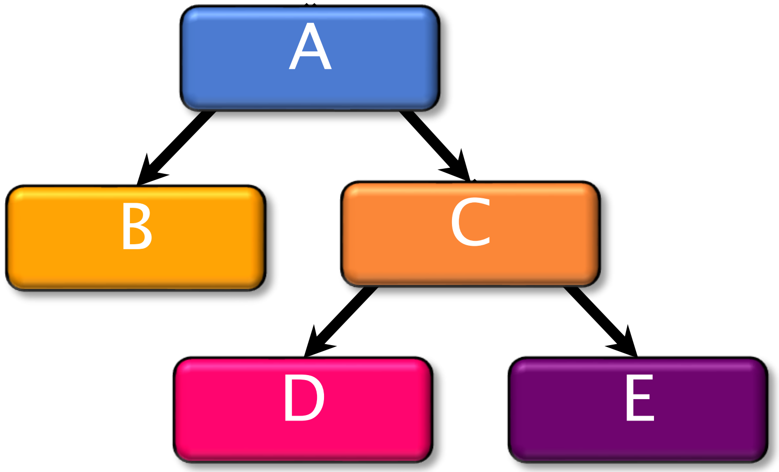


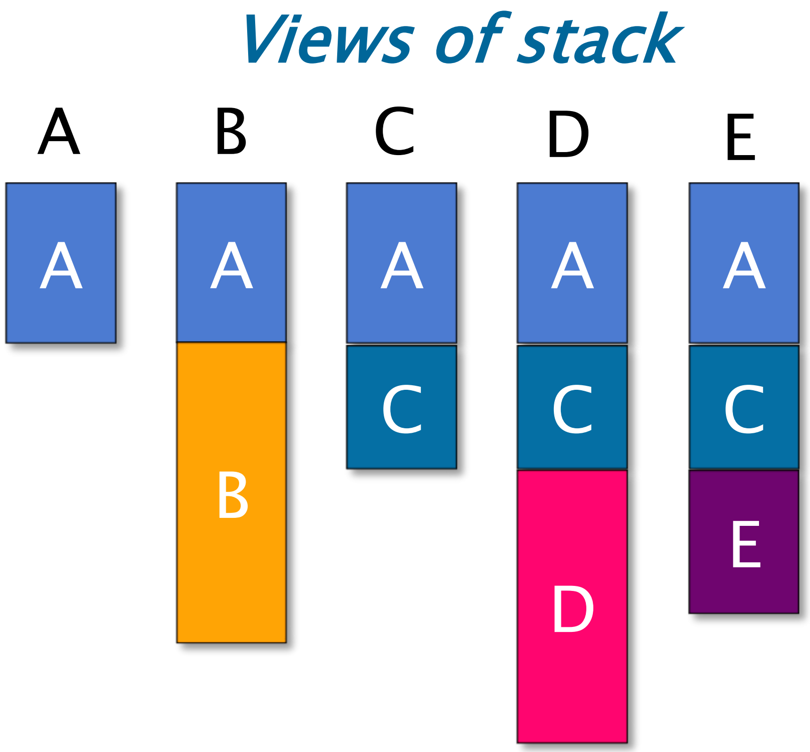
- **Physical communication between two processors**
- **Can lead to considerably more cache misses**

- Coarsening will not increase #SuccSteal

# 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



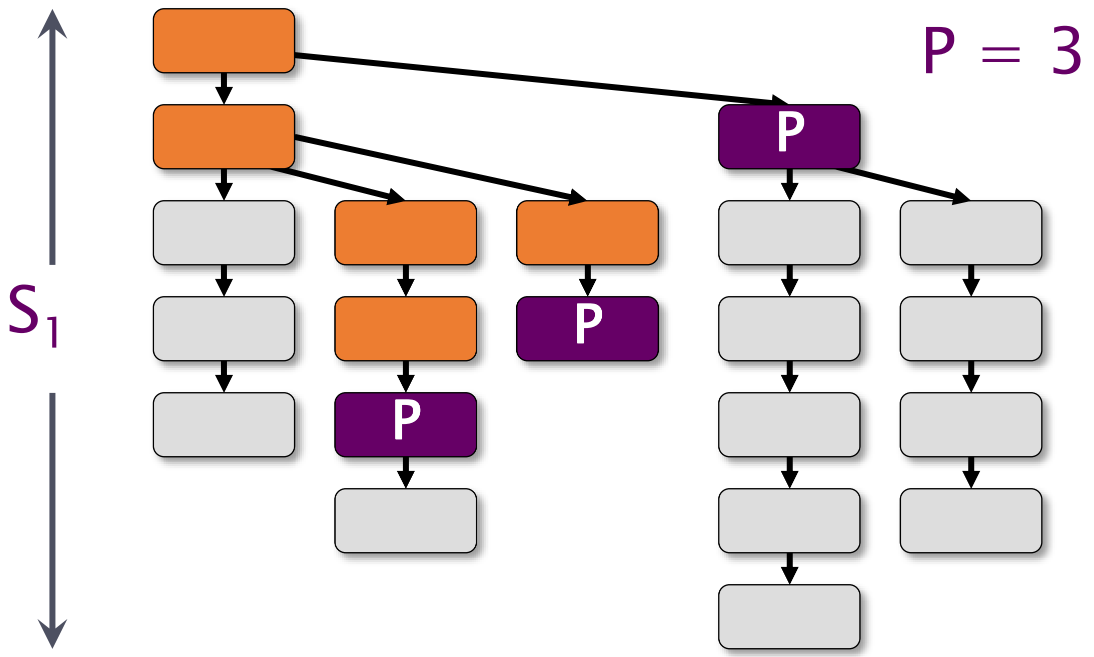




Cilk's cactus stack supports multiple views in parallel.

# Bound on Stack Space

Theorem. Let $S_1$ be the stack space required by a serial execution of a Cilk program. Then the stack space required by a $P$-processor execution is at most $S_P \leq PS_1$.

*Proof* (by induction). The work-stealing algorithm maintains the busy-leaves property: Every extant leaf activation frame has a worker executing it. ■



# CS260: Algorithm Engineering Lecture 8

Fork-Join Parallelism

Greedy Scheduler

Work-Stealing Scheduler

# Design and Analysis of Parallel Algorithms

- Work $W$, depth $D$, I/O cost $Q$ (sequential / random)
- Parallelism for work: $\frac{W}{P}$
- Time for I/O: $\max\left(\frac{Q}{P}, \frac{Q}{B_{max}}\right)$
- Number of steals: $O(PD)$
- Most combinatorial algorithms are I/O bottlenecked