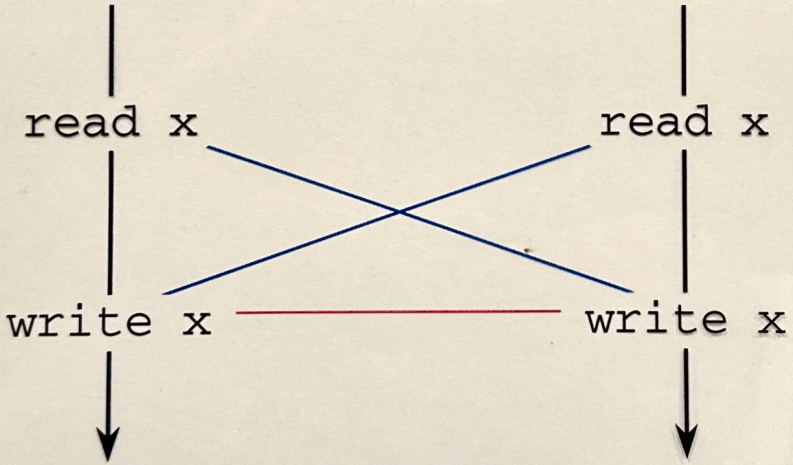
# Efficient Detection of Determinacy Races in Cilk Programs

Charles E. Leiserson
*MIT Laboratory for Computer Science*

Mingdong Feng
*National University of Singapore*
*and*
*MIT Laboratory for Computer Science*

# Determinacy Races

A Cilk program contains a determinacy race if two logically parallel threads access the same shared location, and one of the accesses is a write.

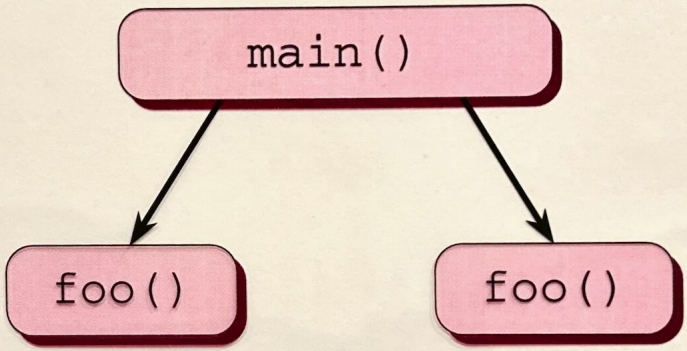



*read/write race*

*write/write race*

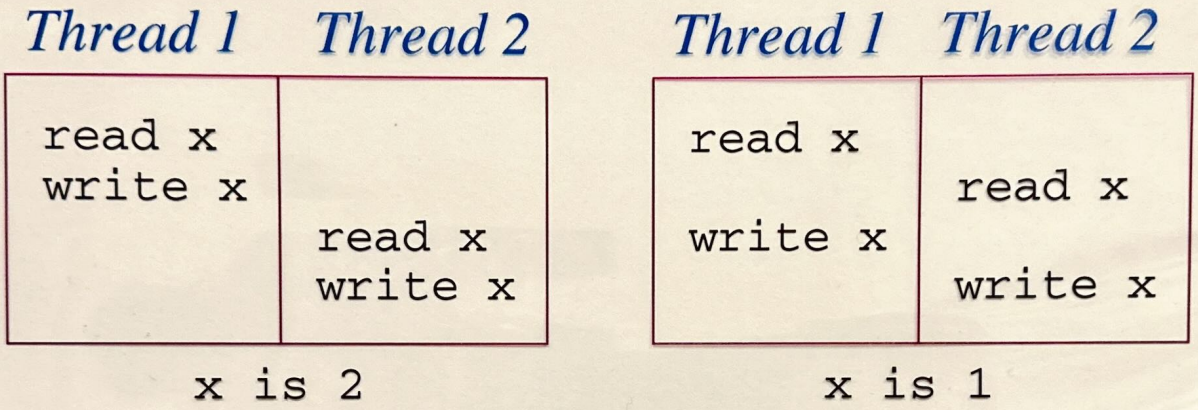
# A Cilk Program with a Determinacy Race

```
int x;

cilk void foo()
{
   x++;
   return;
}

cilk int main()
{
   x = 0;
   spawn foo();
   spawn foo();
   sync;
   printf("x is %d\n", x);
   return 1;
}
```




*spawn tree*

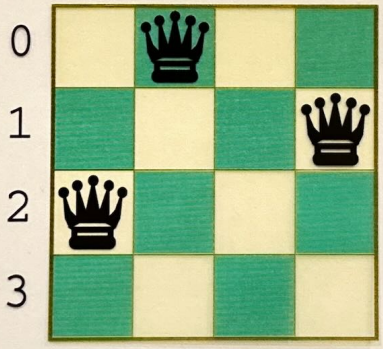
# The Effect of a Determinacy Race

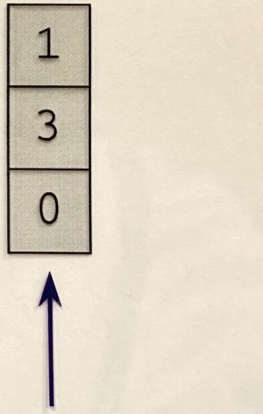



- A determinacy race can cause a Cilk program to behave *nondeterministically*.
- A determinacy race is usually a bug.

# Races in N-queens Puzzle

```
cilk char *nqueens(char *board, int n, int row)
{ char *new_board;
  ...
  new_board = malloc(row+1);
  memcpy(new_board, board, row);
  for (j = 0; j < n; j++)
  { ...
    new_board[row] = j;
    spawn nqueens(new_board, n, row+1);
    ...
  }
  sync;
  ...
}
```

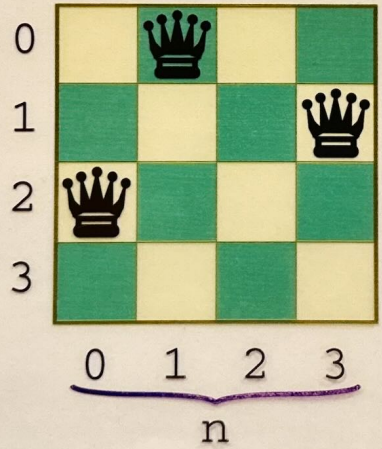


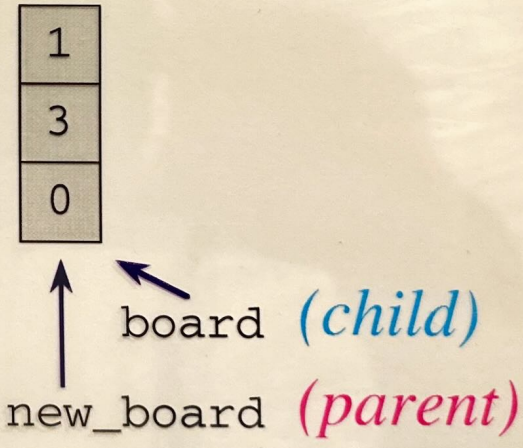



# Races in N-queens Puzzle

```
cilk char *nqueens(char *board, int n, int row)
{ char *new_board;
  ...
  new_board = malloc(row+1);
  memcpy(new_board, board, row);
  for (j = 0; j < n; j++)
  { ...
    new_board[row] = j;
    spawn nqueens(new_board, n, row+1);
    ...
  }
  sync;
  ...
}
```

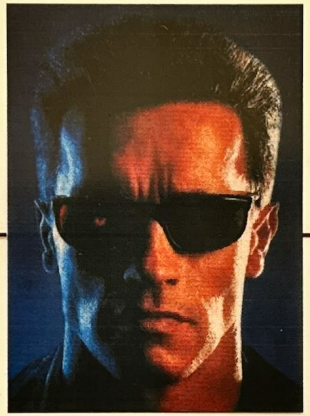
*Race between child reading & parent writing*



# The Nondeterminator

*Cilk program + Input data set*



FAIL

Information localizing a determinacy race.

PASS

*Every* scheduling produces the same result.

---

*A debugging tool, not a verifier.*

# Provable Performance

***Theorem.*** For a Cilk program that runs in $T$ time serially and uses $v$ shared-memory locations, the Nondeterminator runs in $O(T\,\alpha(v,v))$ time, where $\alpha$ is Tarjan's functional inverse of Ackermann's function.

---

- The Nondeterminator is a serial program.
- As a practical matter, $\alpha(v,v) \leq 4$.
- The Nondeterminator uses $O(v)$ space.

# Related Work

| Algorithm | Time/Access | Space |
|---|---|---|
| *English-Hebrew* [Nudler/Rudolph 1986] | $O(pt)$ | $O(vt + \min(np, vtp))$ |
| *Task recycling* [Dinning/Schonberg 1990] | $O(t)$ | $O(vt + t^2)$ |
| *Offset-span labeling* [Mellor-Crummey 1991] | $O(p)$ | $O(v + \min(np, vp))$ |
| *SP-bags* [Feng/Leiserson 1996] | $O(\alpha(v,v))$ amortized | $O(v)$ |

$n$ = number of threads
$v$ = number of shared locations
$t$ = max number of parallel threads
$p$ = max depth of nested parallelism

# Outline

- Series-Parallel Dags
- Tarjan's Least Common Ancestors Algorithm
- The SP-Bags Algorithm
- Empirical Results
- Conclusion

- Series-Parallel Dags

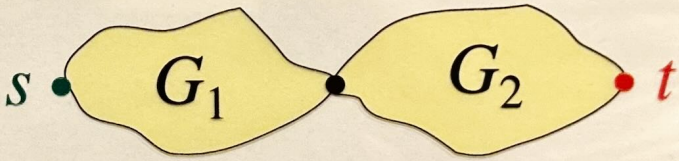
# Series-Parallel Dags

***Base graph:***

- source $s$
- sink $t$

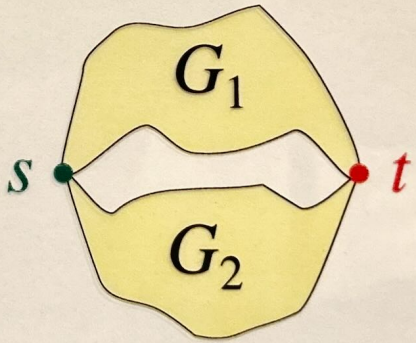

***Series composition:***

- $s = s_1$
- $t_1 = s_2$
- $t = t_2$

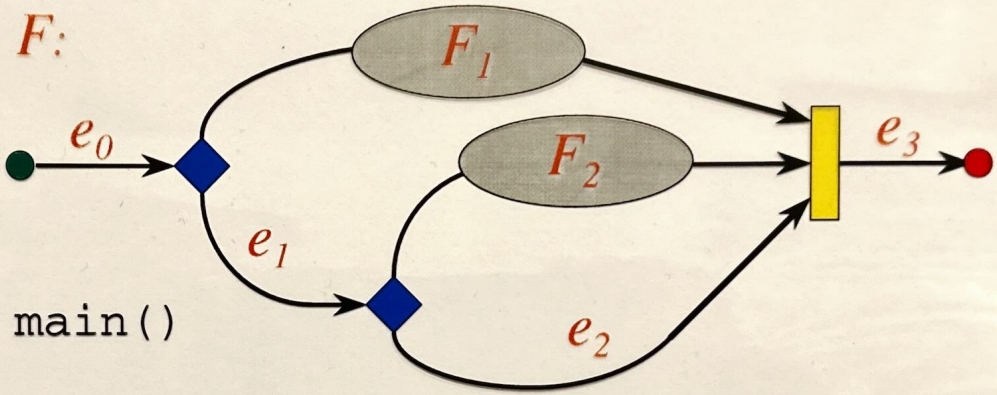



***Parallel composition:***

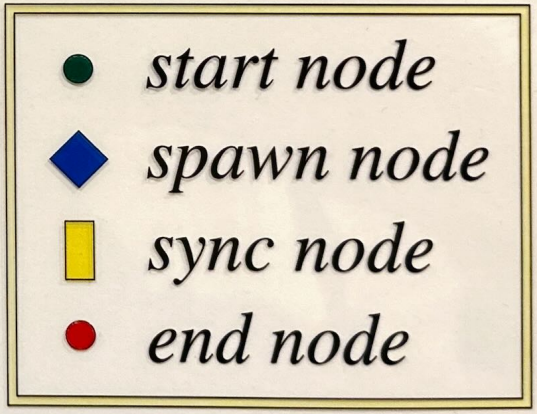
- $s = s_1 = s_2$
- $t = t_1 = t_2$



# A Cilk Dag is Series-Parallel



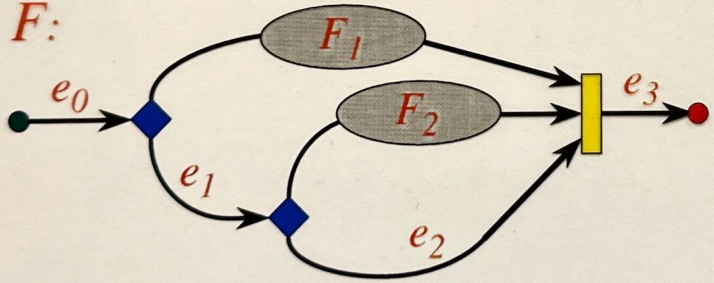

```
F:  cilk int main()
    {
e0:     x = 0;
F1:     spawn foo();
e1:
F2:     spawn foo();
e2:
        sync;
e3:     printf("%d", x);
        return 1;
    }
```

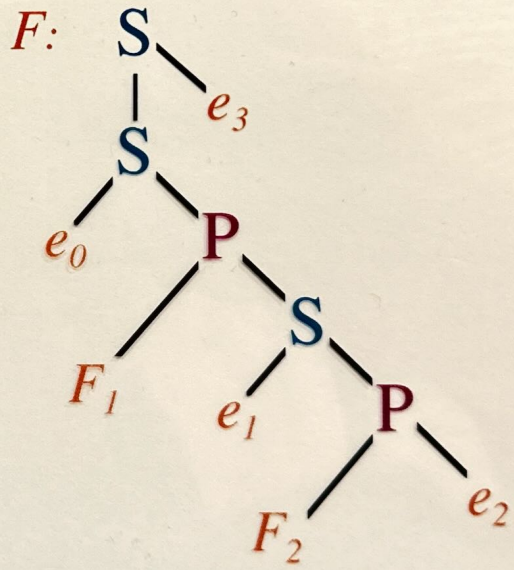


# Series-Parallel Parse Trees

Every series-parallel dag has a ***parse tree***. The ***least common ancestor*** of two threads determines whether the threads are logically in series or in parallel.

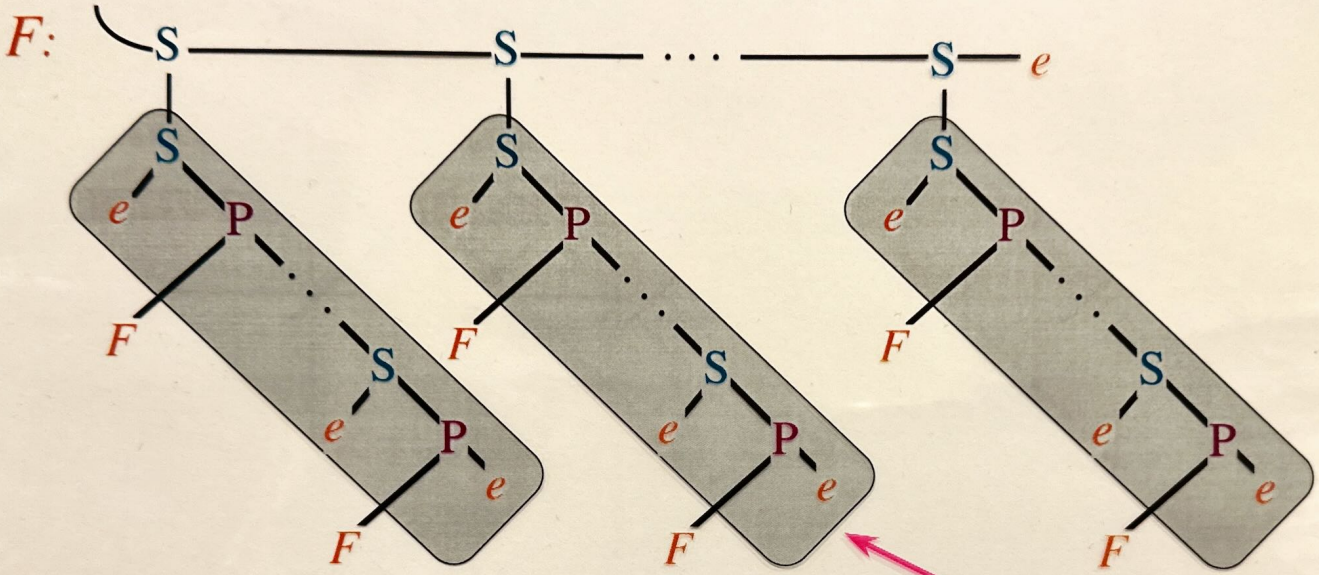
- $e \prec e'$ if LCA($e$,$e'$) is S and $e$ is left of $e'$.
- $e \parallel e'$ if LCA($e$,$e'$) is P.




*A treewalk visits threads in serial execution order.*

# Canonical Cilk Parse Tree




*e*; spawn *F*; *e*; spawn *F*; … *e*; sync; ← ***sync block***

*e*; spawn *F*; *e*; spawn *F*; … *e*; sync;

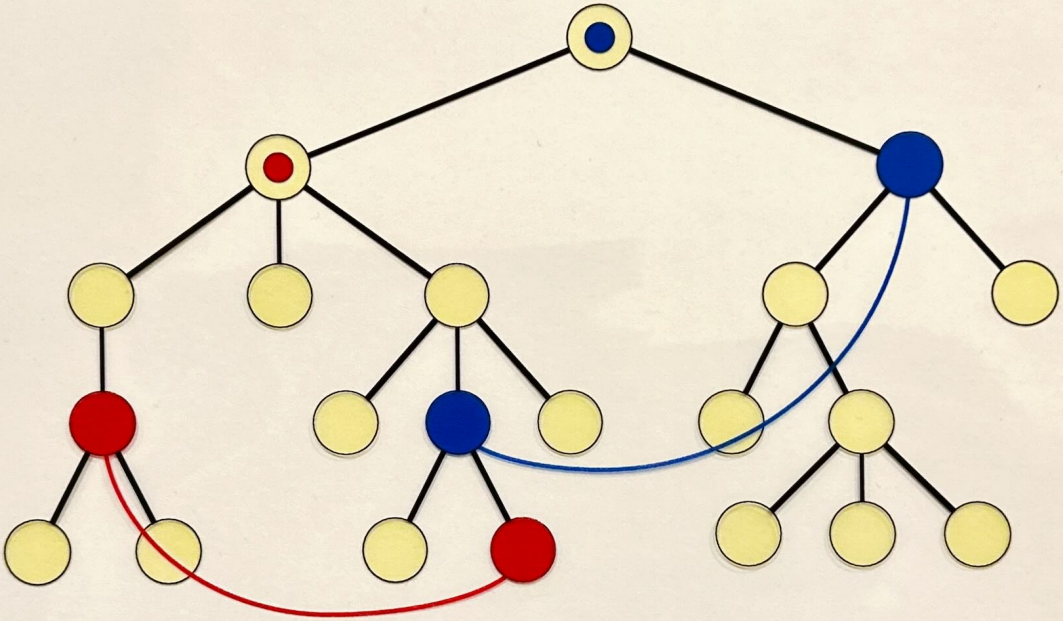
⋮

*e*; spawn *F*; *e*; spawn *F*; … *e*; sync;

*e*; return;

- Tarjan's Least Common Ancestors Algorithm

# Least Common Ancestors



***Definition.*** The *least common ancestor* of two nodes in a rooted tree is the node on the path between them that is closest to the root.

# Disjoint-Set Data Structure

$\Sigma$ is a collection of *disjoint* sets.

- $X, Y \in \Sigma$ implies $X \cap Y = \emptyset$.

---

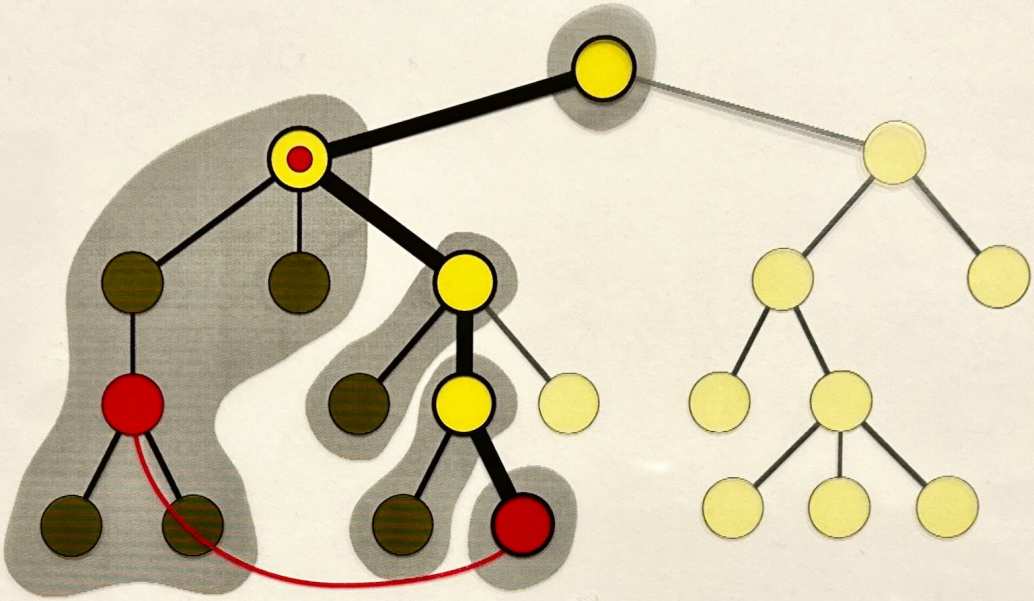
Three operations:

- MAKE-SET($e$): $\Sigma \leftarrow \Sigma \cup \{\{e\}\}$.
- UNION($X,Y$): $\Sigma \leftarrow \Sigma - \{X,Y\} \cup \{X \cup Y\}$.
- FIND-SET($e$): returns $X \in \Sigma$ such that $e \in X$.

---

Any sequence of $m$ operations on $n$ sets can be performed in $O(m\ \alpha(m,n))$ time [Tarjan 1975].

# Tarjan's LCA Algorithm



***Depth-first treewalk:***

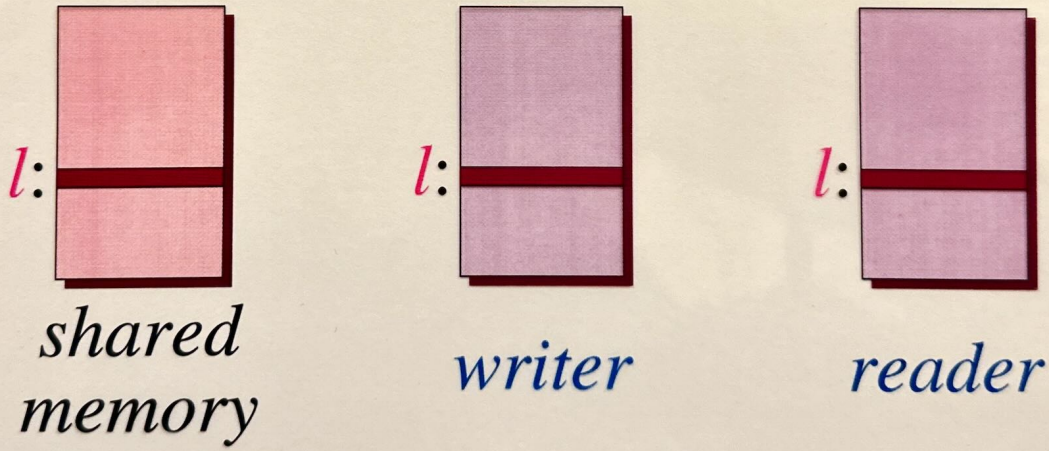
- Visit node $v$: $S[v] \leftarrow$ MAKE-SET($v$)
- Return to $u$ from $v$: $S[u] \leftarrow$ UNION($S[u]$,$S[v]$)
- Encounter edge $(u,v)$ for the second time at $v$: LCA($u$,$v$) = FIND-SET($u$)

- THE SP-BAGS ALGORITHM

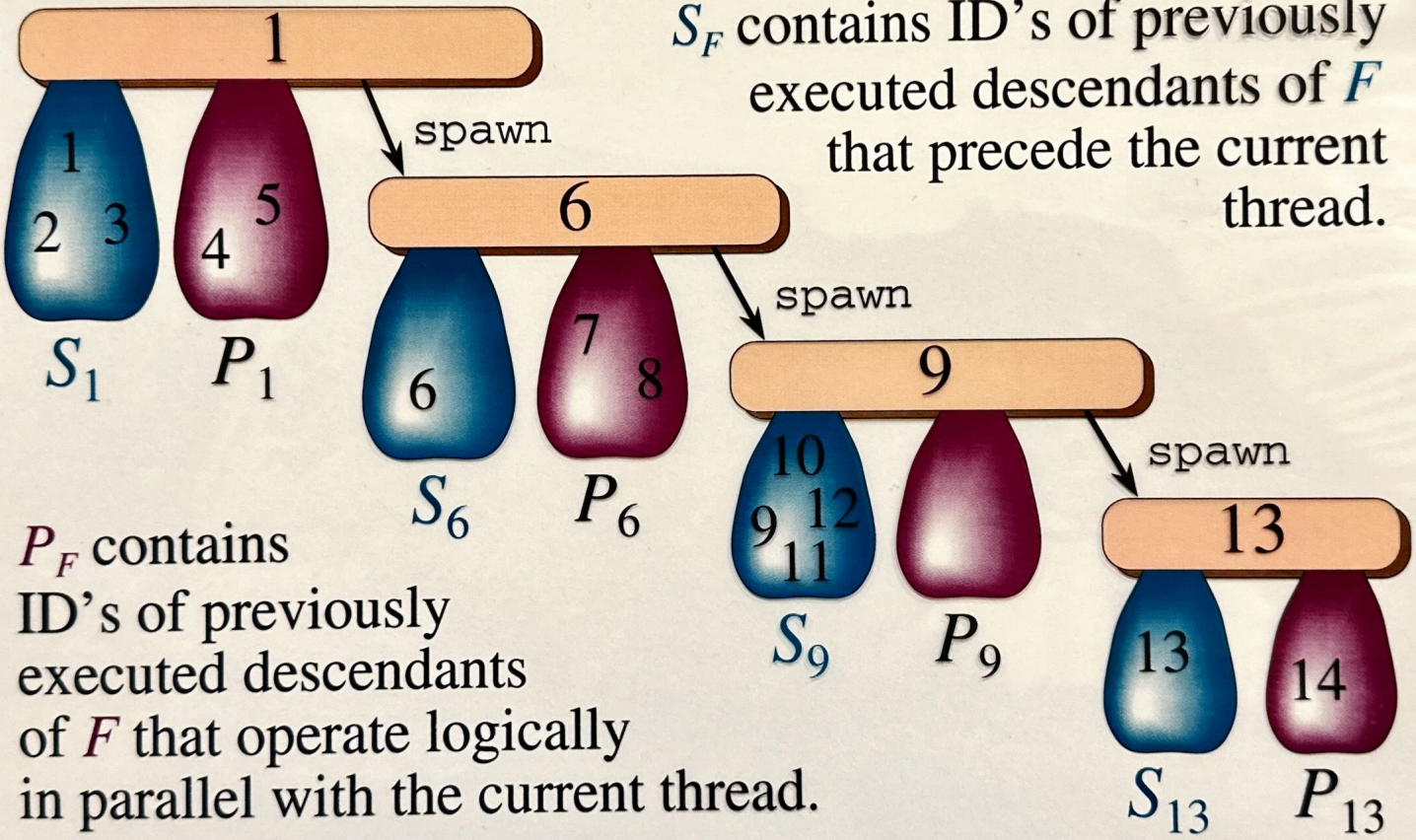
# Shadow Spaces

Each shared-memory location $l$ has two corresponding shadow locations that are updated by the SP-bags algorithm as the Cilk program executes:

- *writer*[$l$]: ID of a procedure that wrote $l$.
- *reader*[$l$]: ID of a procedure that read $l$.



# S-Bags and P-Bags




$S_F$ contains ID's of previously executed descendants of $F$ that precede the current thread.

$P_F$ contains ID's of previously executed descendants of $F$ that operate logically in parallel with the current thread.

# The SP-Bags Algorithm

`spawn` procedure $F$: $S_F \leftarrow$ MAKE-SET($F$); $P_F \leftarrow \emptyset$

`sync` in a procedure $F$: $S_F \leftarrow$ UNION($S_F$,$P_F$); $P_F \leftarrow \emptyset$

`return` from $F'$ to $F$: $P_F \leftarrow$ UNION($P_F$,$S_{F'}$)

`write` location $l$ by a procedure $F$:
  **if** FIND-SET($reader[l]$) is a P-bag
    or FIND-SET($writer[l]$) is a P-bag
    **then** a determinacy race exists
  $writer[l] \leftarrow F$

`read` location $l$ by a procedure $F$:
  **if** FIND-SET($writer[l]$) is a P-bag
    **then** a determinacy race exists
  **if** FIND-SET($reader[l]$) is an S-bag
    **then** $reader[l] \leftarrow F$

# Correctness of SP-Bags

***Lemma***. Suppose threads $e_1$, $e_2$, and $e_3$ execute in order in the normal serial execution. Then

- $e_1 \prec e_2$ and $e_1 \parallel e_3$ implies $e_2 \parallel e_3$;
- (*Pseudotransitivity*)
  $e_1 \parallel e_2$ and $e_2 \parallel e_3$ implies $e_1 \parallel e_3$.