# Review & Objectives

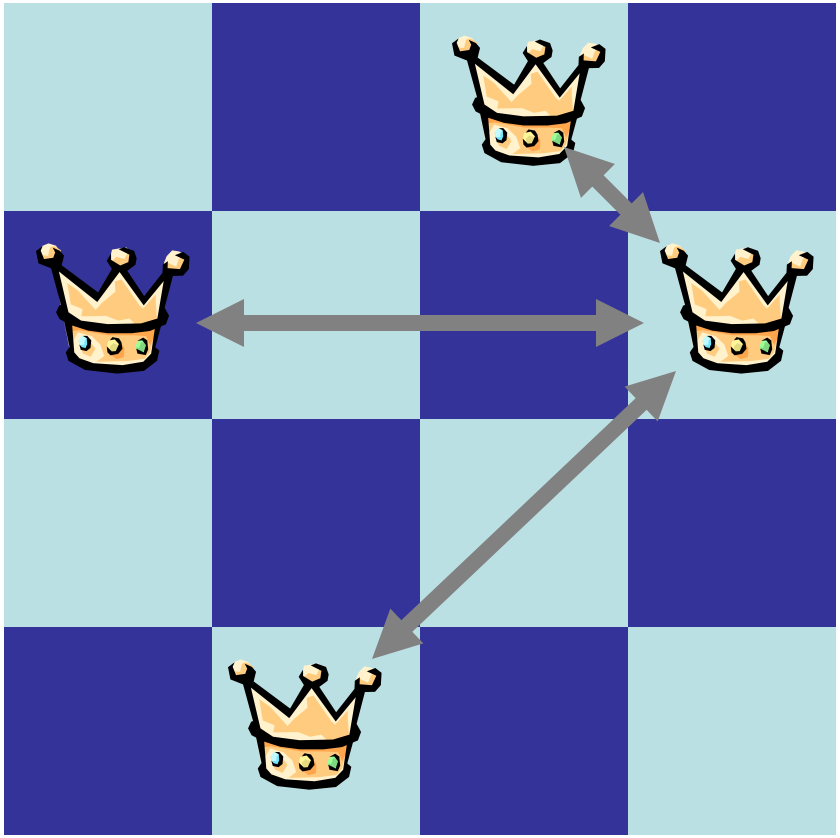
Previously:

- Described how the OpenMP `task` pragma is different from the `for` pragma
- Showed how to code task decomposition solutions for `while` loop and recursive tasks, with the OpenMP `task` construct

At the end of this part you should be able to:

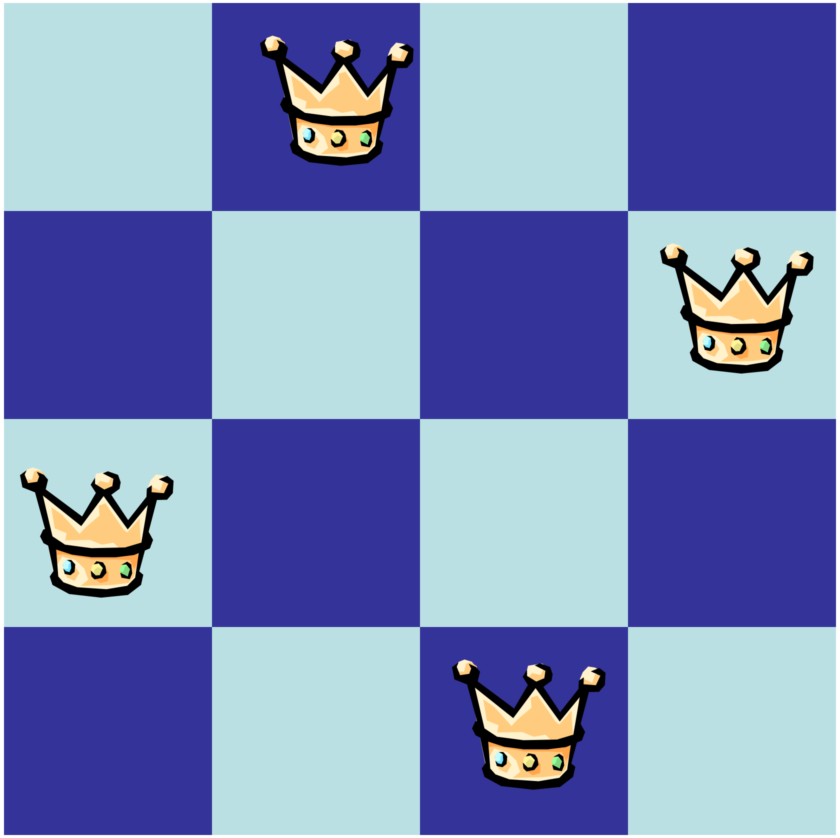
- Design and implement a task decomposition solution

# Case Study: The N Queens Problem

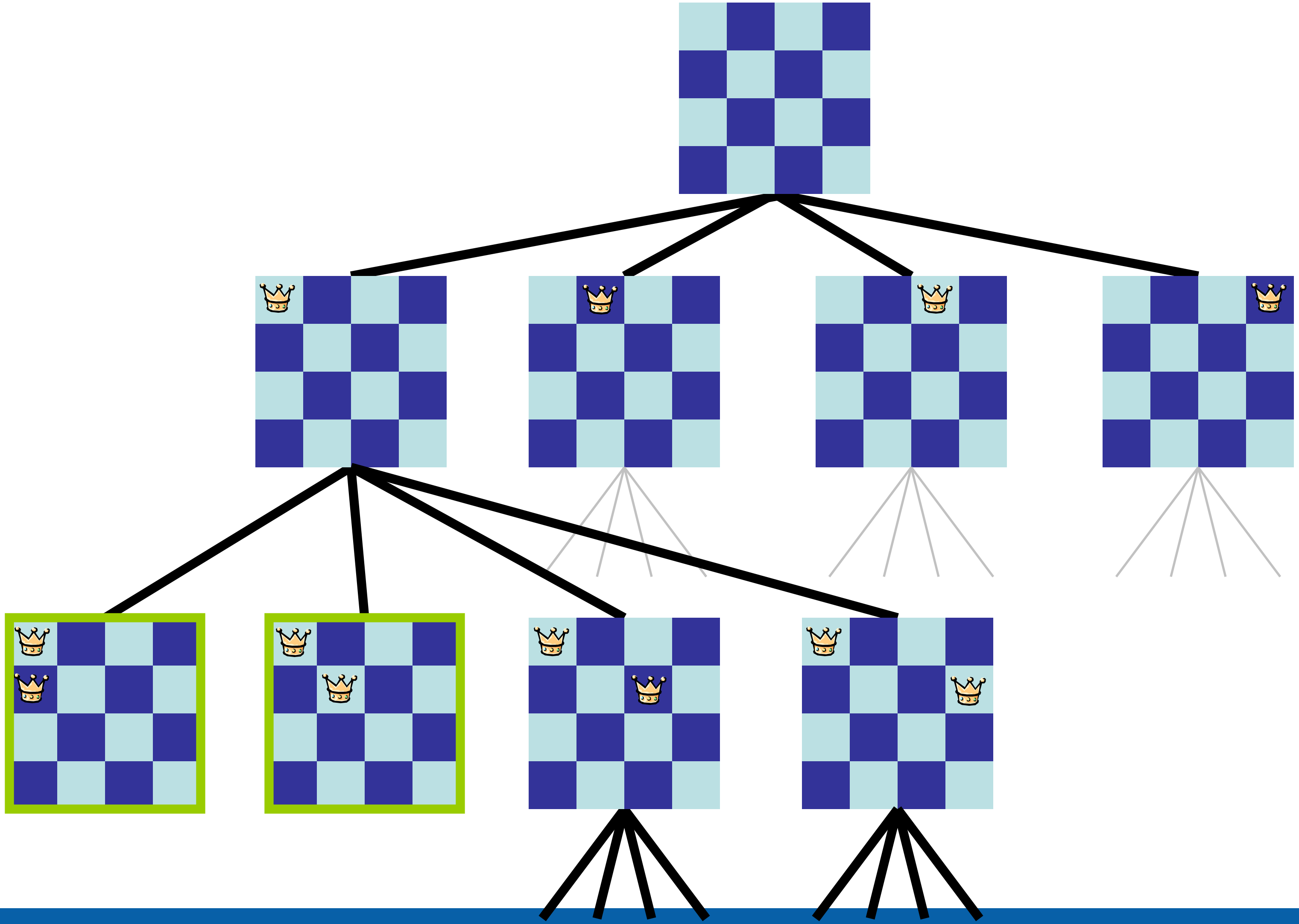


Is there a way to place N queens on an N-by-N chessboard such that no queen threatens another queen?

# A Solution to the 4 Queens Problem



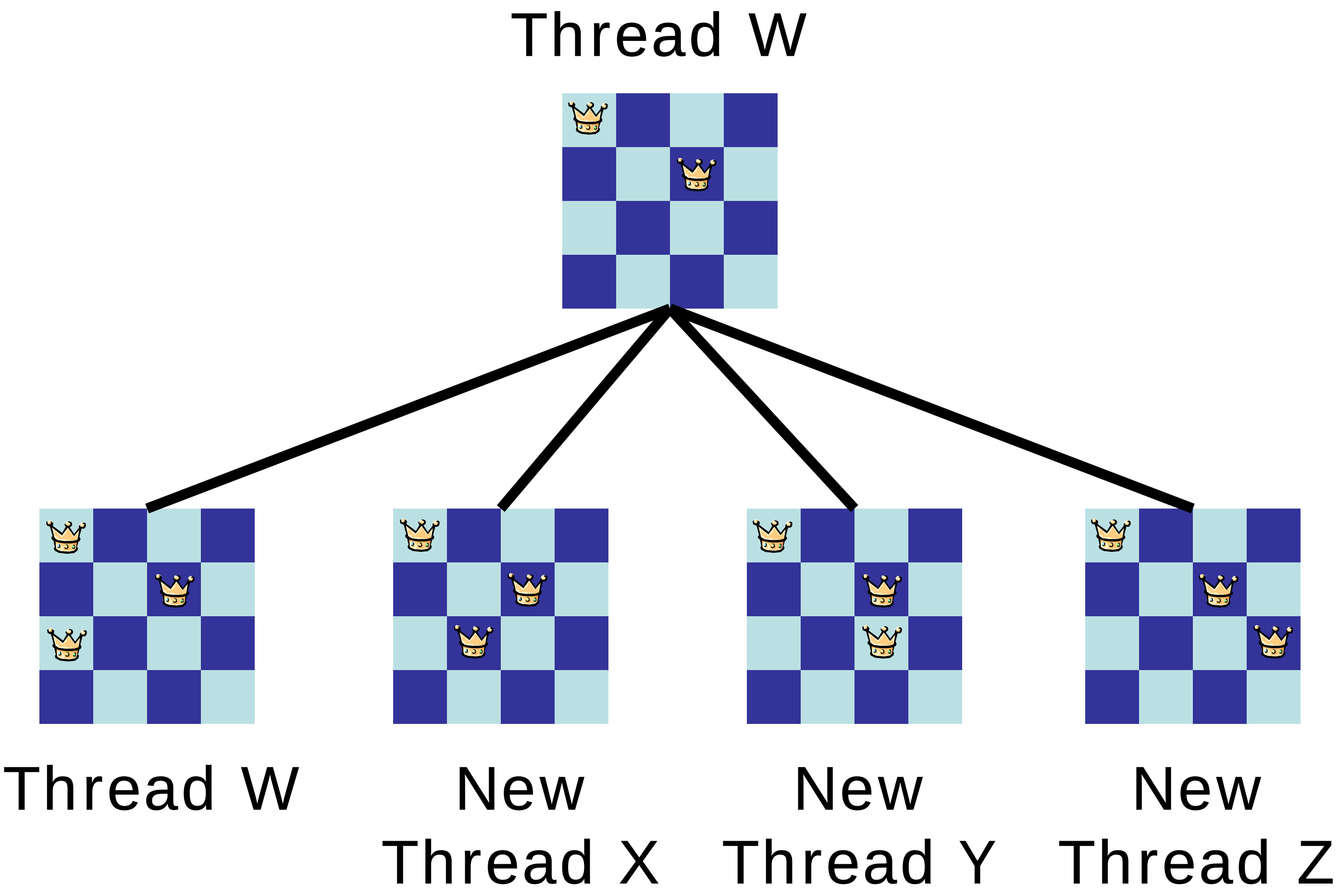
# Exhaustive Search



# Design #1 for Parallel Search

- Create threads to explore different parts of the search tree simultaneously
- If a node has children
  - The thread creates child nodes
  - The thread explores one child node itself
  - Thread creates a new thread for every other child node

# Design #1 for Parallel Search



# Pros and Cons of Design #1

Pros

- Simple design, easy to implement
- Balances work among threads

Cons

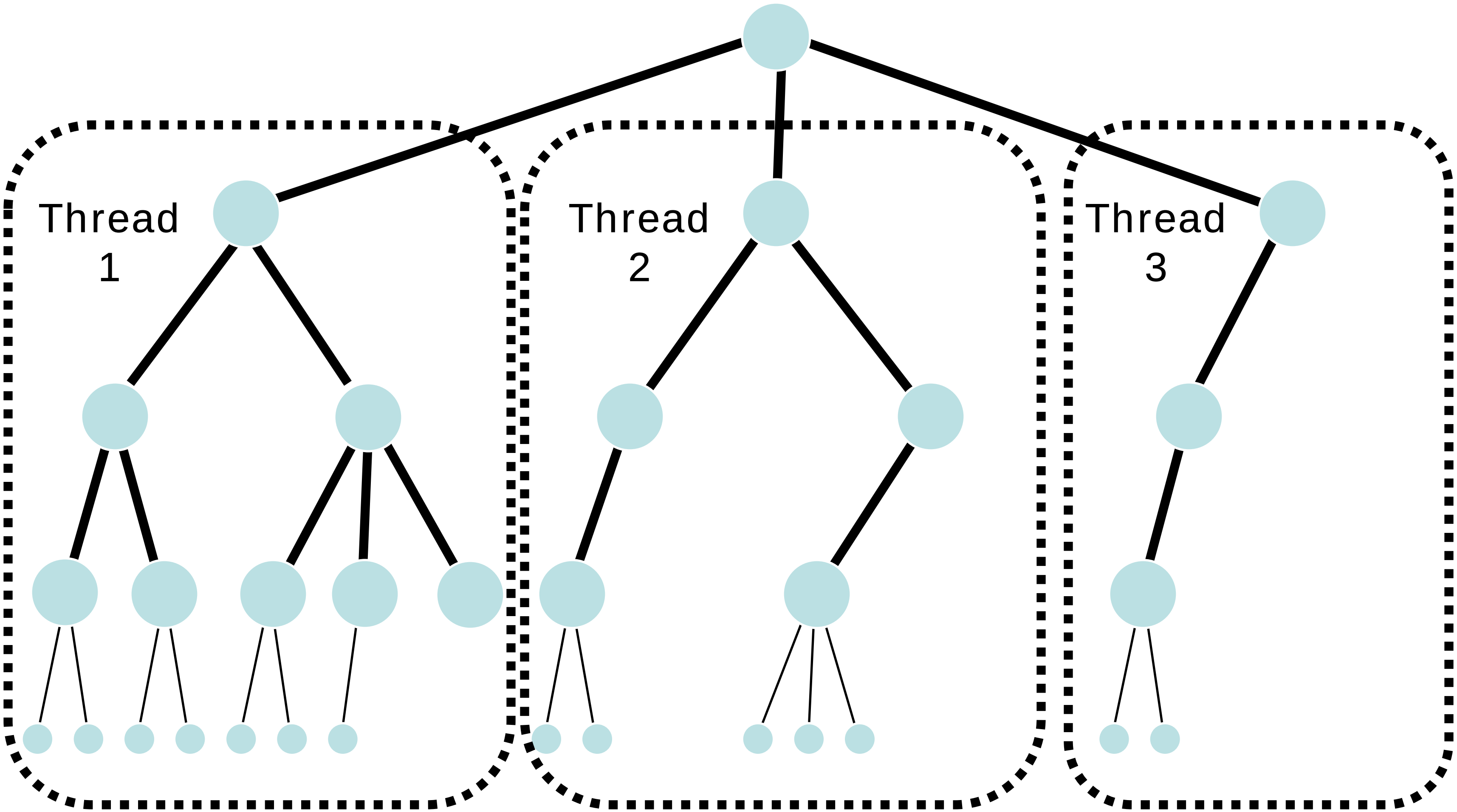
- Too many threads created
- Lifetime of threads too short
- Overhead costs too high

# Design #2 for Parallel Search

One thread created for each subtree rooted at a particular depth

Each thread sequentially explores its subtree

# Design #2 in Action



# Pros and Cons of Design #2

Pros

- Thread creation/termination time minimized

Cons

- Subtree sizes may vary dramatically
- Some threads may finish long before others
- Imbalanced workloads lower efficiency

# Design #3 for Parallel Search

Main thread creates work pool—list of subtrees to explore

Main thread creates finite number of co-worker threads

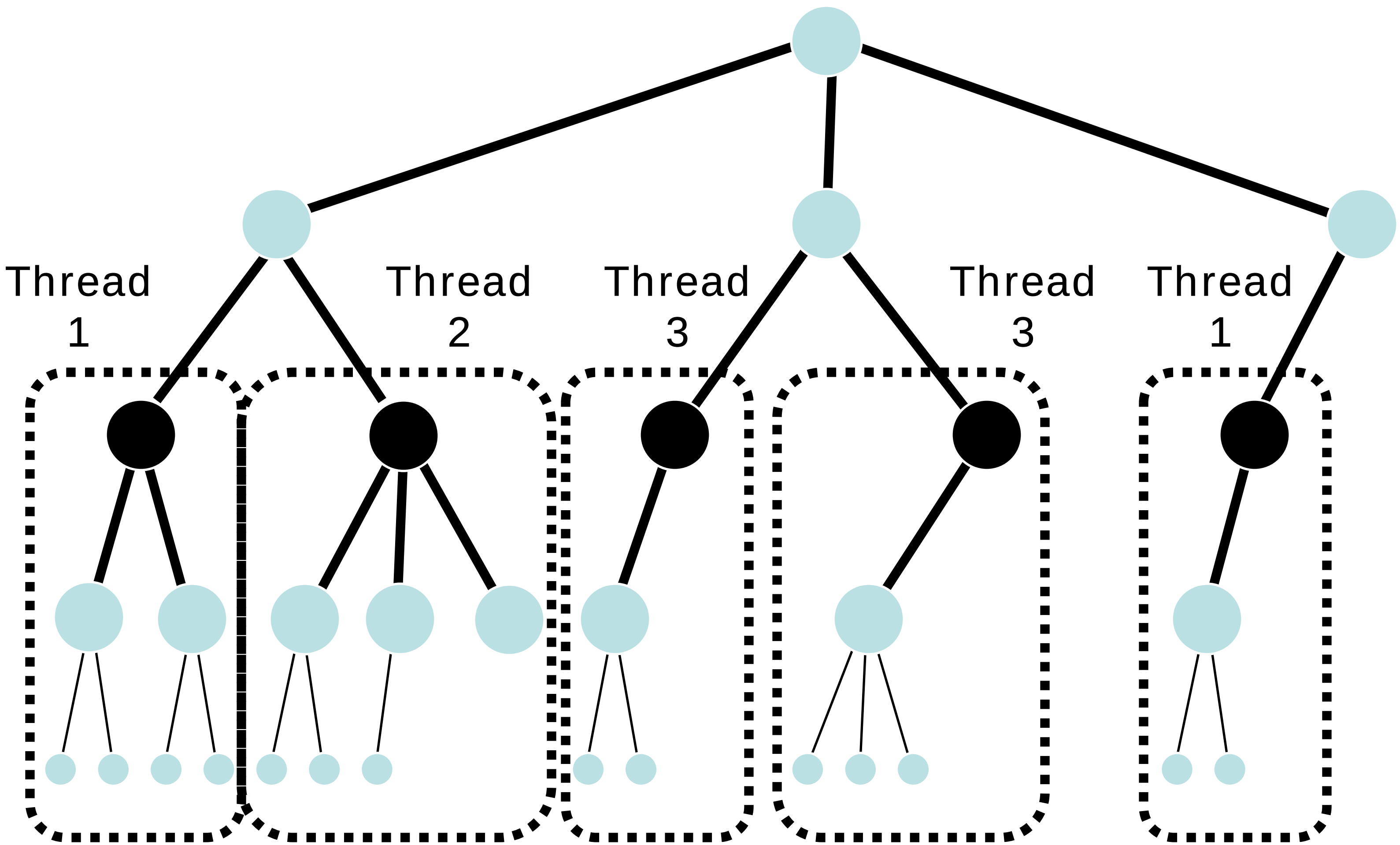
Each subtree exploration is done by a single thread

Inactive threads go to pool to get more work

# Work Pool Analogy

- More rows than workers
- Each worker takes an unpicked row and picks the crop
- After completing a row, the worker takes another unpicked row
- Process continues until all rows have been harvested

# Design #3 in Action



# Pros and Cons of Strategy #3

Pros

- Thread creation/termination time minimized
- Workload balance better than strategy #2

Cons

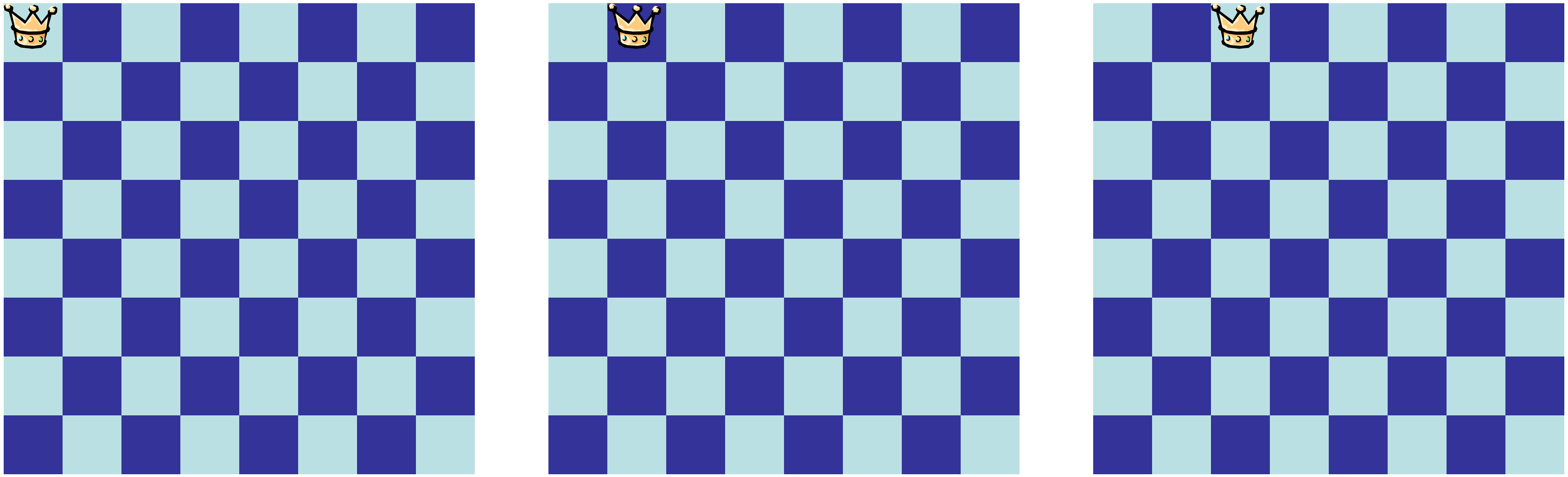
- Threads need exclusive access to data structure containing work to be done, a sequential component
- Workload balance worse than strategy #1

Conclusion

- Good compromise between designs 1 and 2

# Implementing Strategy #3 for N Queens

Work pool consists of N boards representing N possible placements of queen on first row



# Parallel Program Design

- One thread creates list of partially filled-in boards
- Fork: Create one thread per core
- Each thread repeatedly gets board from list, searches for solutions, and adds to solution count, until no more board on list
- Join: Occurs when list is empty
- One thread prints number of solutions found

# Search Tree Node Structure

```
/*    The ‘board’ struct contains information about a
      node in the search tree; i.e., partially filled-
      in board. The work pool is a singly linked
      list of ‘board’ structs. */
struct board {
   int pieces;           /* # of queens on board*/
   int places[MAX_N];    /* Queen’s pos in each row */
   struct board *next;   /* Next search tree node */
};
```

# Key Code in main Function

```
struct board *stack;
...
stack = NULL;
for (i = 0; i < n; i++) {
  initial=(struct board *)malloc(sizeof(struct board));
  initial->pieces = 1;
  initial->places[0] = i;
  initial->next = stack;
  stack = initial;
}
num_solutions = 0;
search_for_solutions (n, stack, &num_solutions);
printf ("The %d-queens puzzle has %d solutions\n", n,
            num_solutions);
```

# Insertion of OpenMP Code

```
struct board *stack;
...
stack = NULL;
for (i = 0; i < n; i++) {
  initial=(struct board *)malloc(sizeof(struct board));
  initial->pieces = 1;
  initial->places[0] = i;
  initial->next = stack;
  stack = initial;
}
num_solutions = 0;

#pragma omp parallel
 search_for_solutions (n, stack, &num_solutions);

printf ("The %d-queens puzzle has %d solutions\n", n,
            num_solutions);
```

# Original C Function to Get Work

```
void search_for_solutions (int n,
   struct board *stack, int *num_solutions)
{
   struct board *ptr;
   void search (int, struct board *, int *);

   while (stack != NULL) {
      ptr = stack;
      stack = stack->next;
      search (n, ptr, num_solutions);
      free (ptr);
   }
}
```

# C/OpenMP Function to Get Work

```
void search_for_solutions (int n,
   struct board *stack, int *num_solutions)
{
   struct board *ptr;
   void search (int, struct board *, int *);

   while (stack != NULL) {
  #pragma omp critical
  { ptr = stack; stack = stack->next; }
     search (n, ptr, num_solutions);
     free (ptr);
   }
}
```

# Original C Search Function

```
void search (int n, struct board *ptr,
             int *num_solutions)
{
   int i;
   int no_threats (struct board *);

   if (ptr->pieces == n) {
      (*num_solutions)++;
   } else {
      ptr->pieces++;
      for (i = 0; i < n; i++) {
         ptr->places[ptr->pieces-1] = i;
         if (no_threats(ptr))
      search (n, ptr, num_solutions);
      }
      ptr->pieces--;
   }
}
```

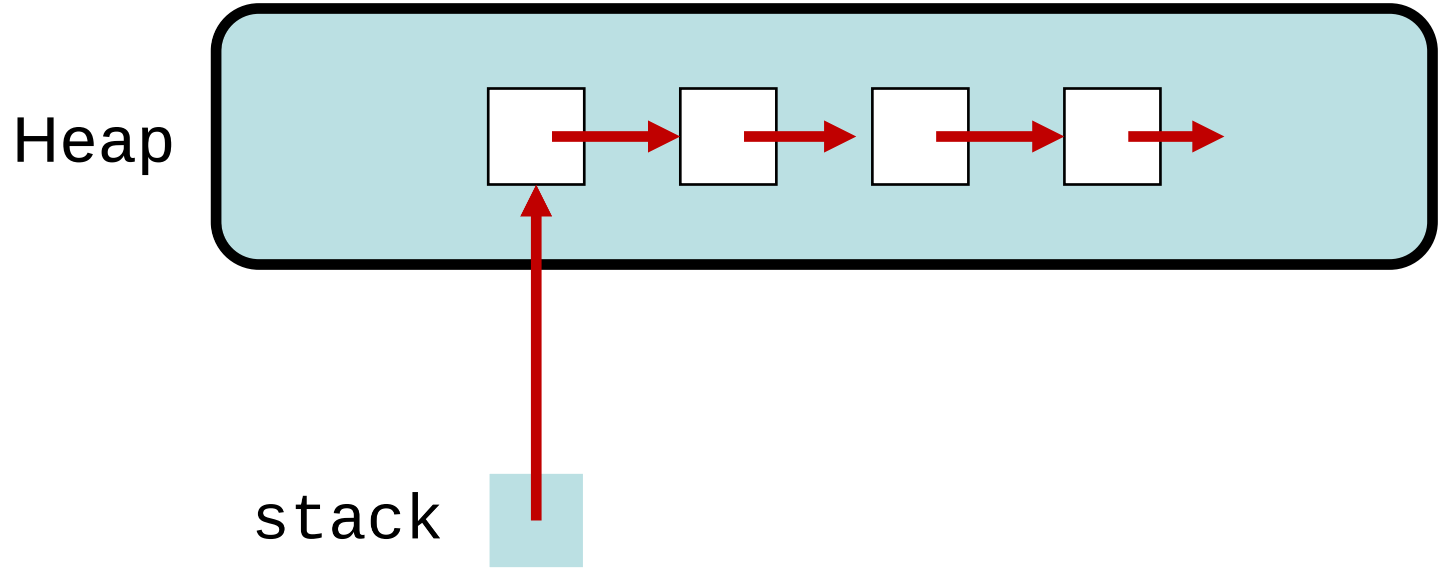
# C / OpenMP Search Function

```
void search (int n, struct board *ptr,
             int *num_solutions)
{
   int i;
   int no_threats (struct board *);

   if (ptr->pieces == n) {
     #pragma omp critical
      (*num_solutions)++;
   } else {
      ptr->pieces++;
      for (i = 0; i < n; i++) {
         ptr->places[ptr->pieces-1] = i;
         if (no_threats(ptr))
      search (n, ptr, num_solutions);
      }
      ptr->pieces--;
   }
}
```

# Only One Problem: It Doesn't Work!

- OpenMP program throws an exception
- Culprit: Variable stack

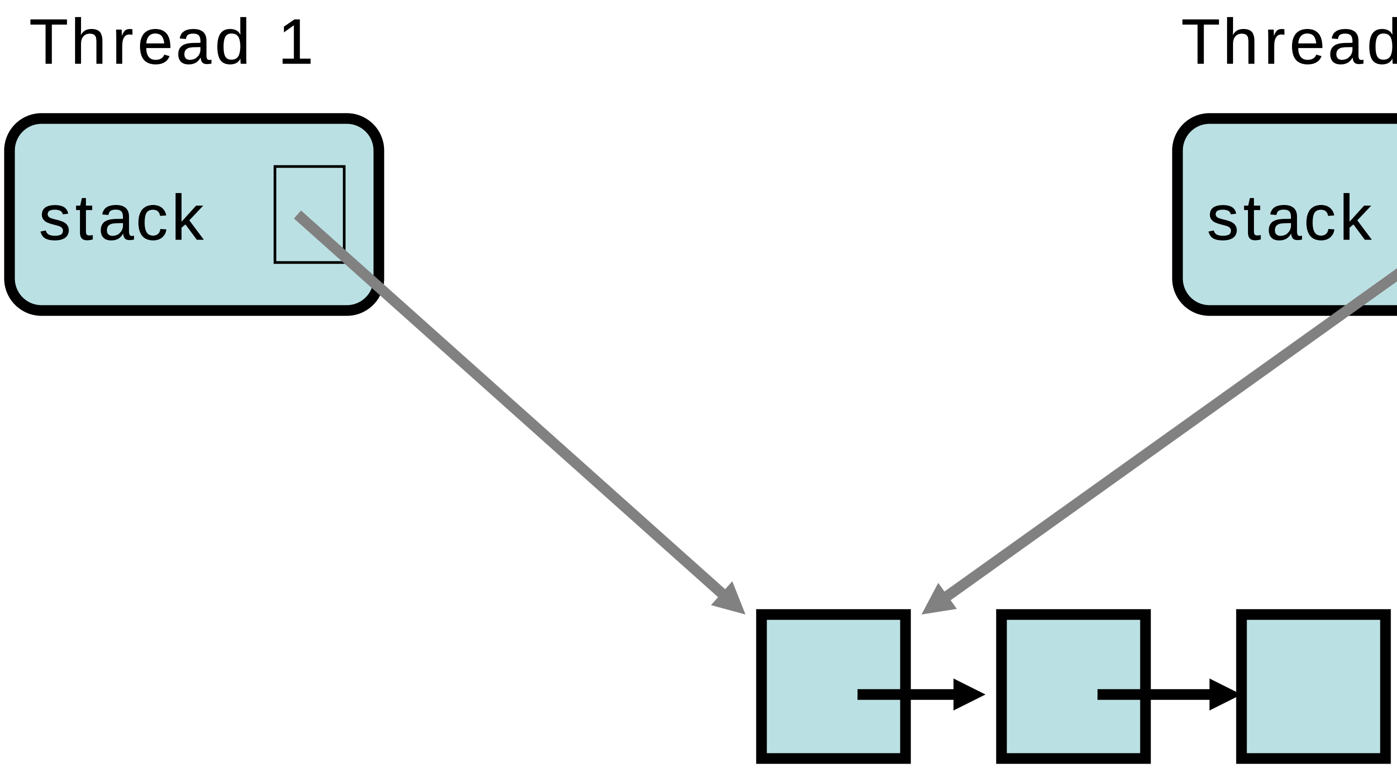


# Problem Site

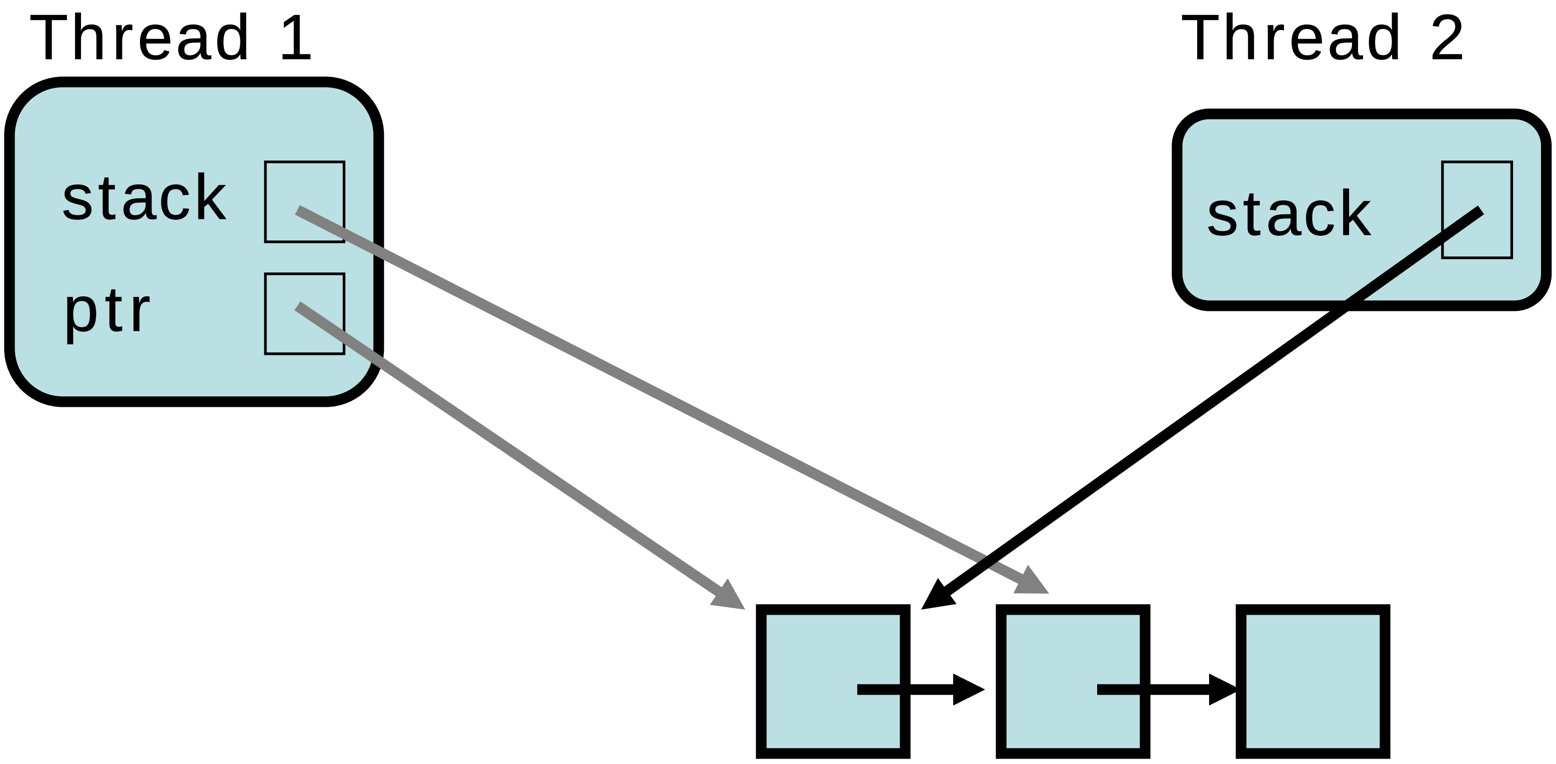
```
int main ()
{
   struct board *stack;
   ...
  #pragma omp parallel
    search_for_solutions(n, stack, &num_solutions);
   ...
}

void search_for_solutions (int n,
   struct board *stack, int *num_solutions)
{
   ...
   while (stack != NULL) ...
```

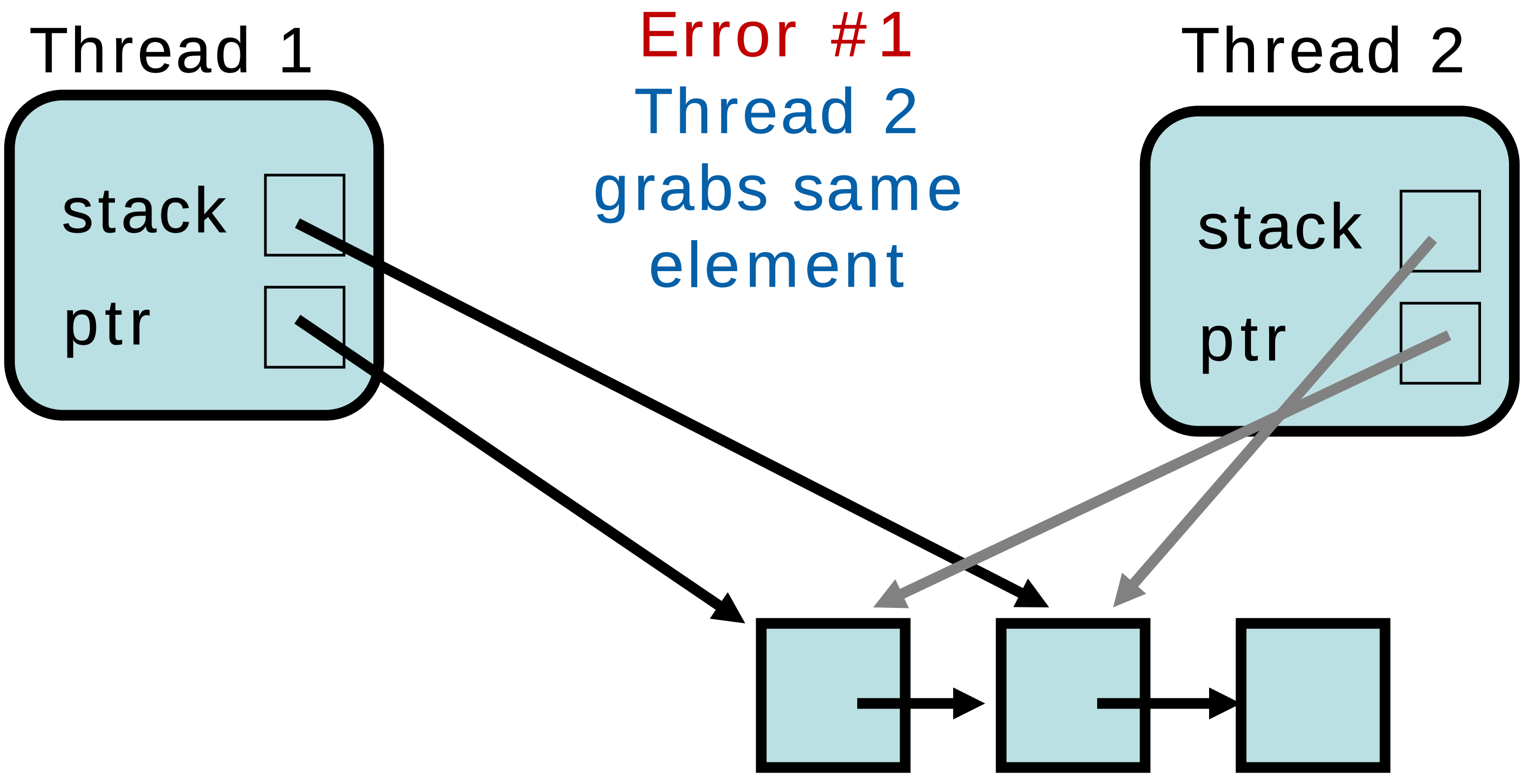
# 1. Both Threads Point to Top



# 2. Thread 1 Grabs First Element



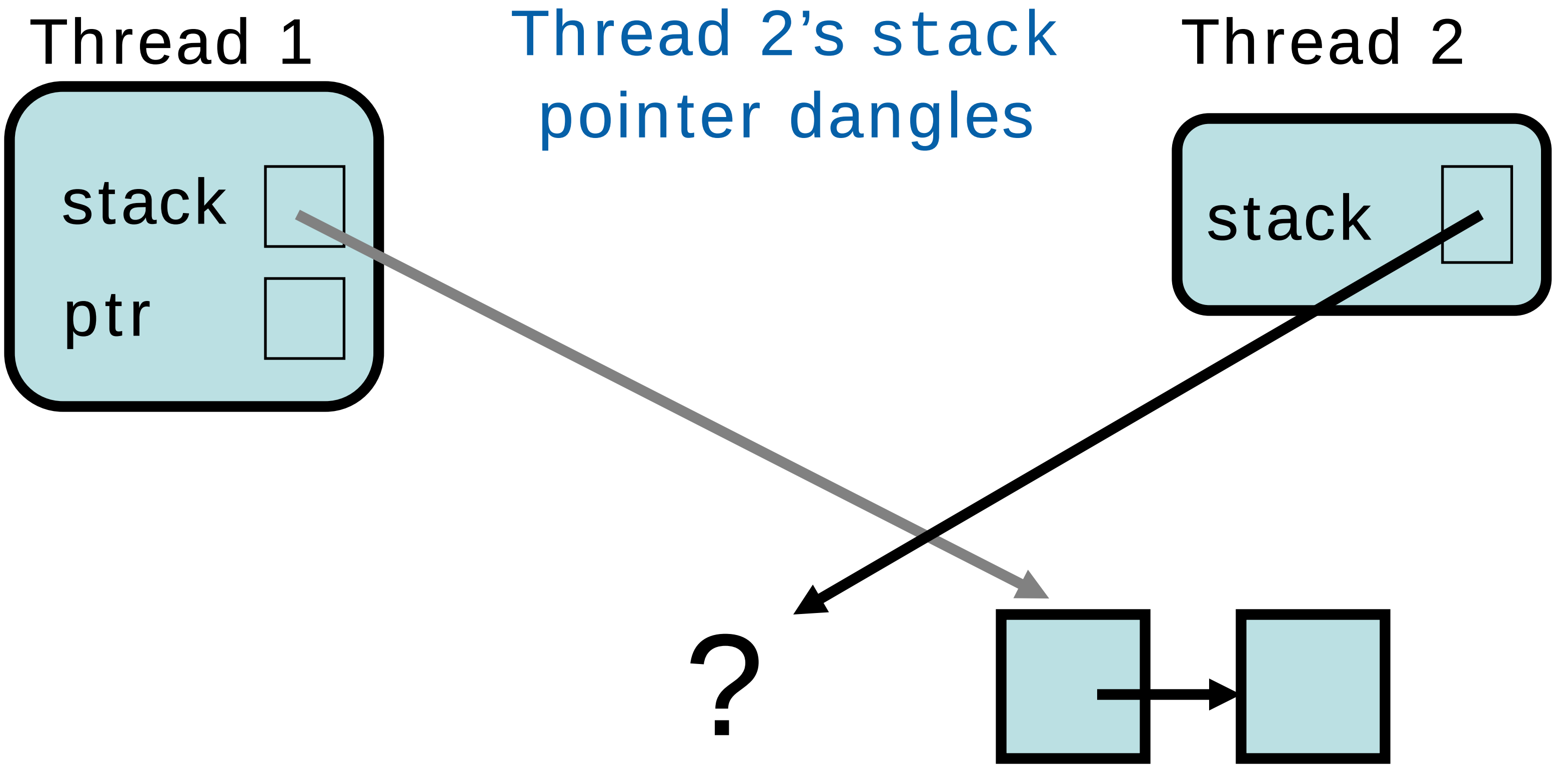
# 3. Thread 2 Grabs “Next” Element



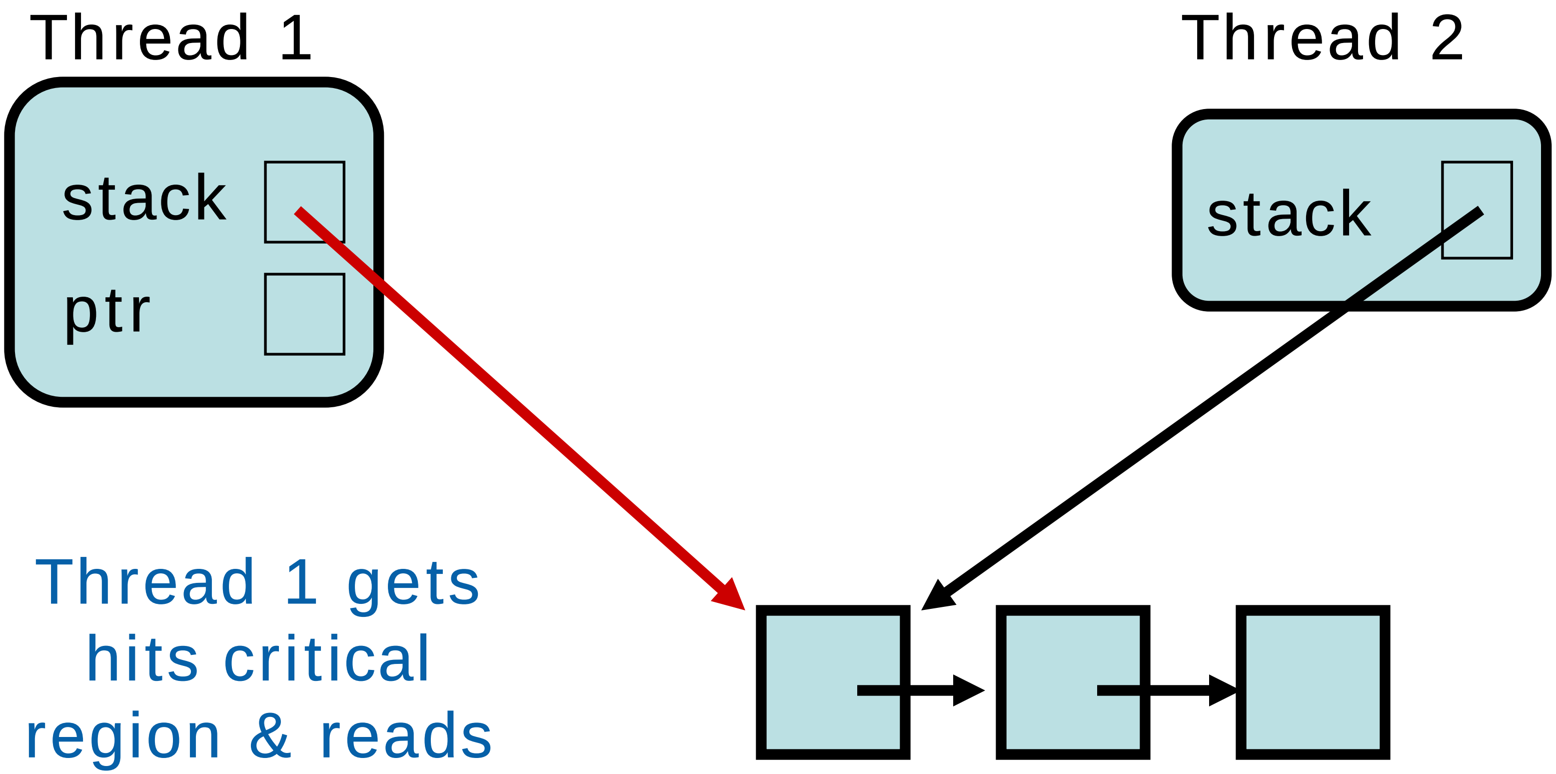
# 4. Thread 1 Deletes Element

Error #2

Thread 2's stack pointer dangles

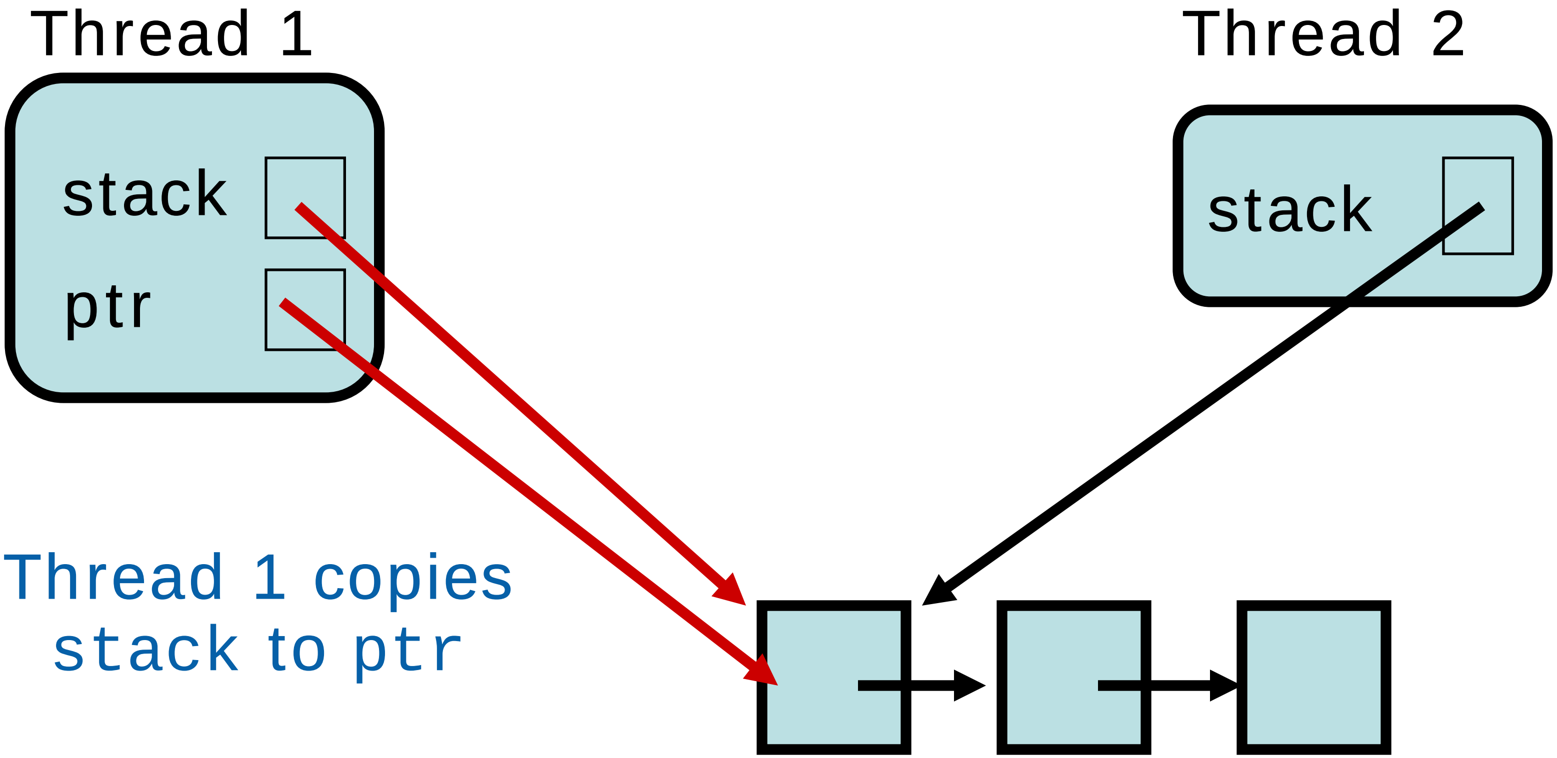


# Demonstrate error #2



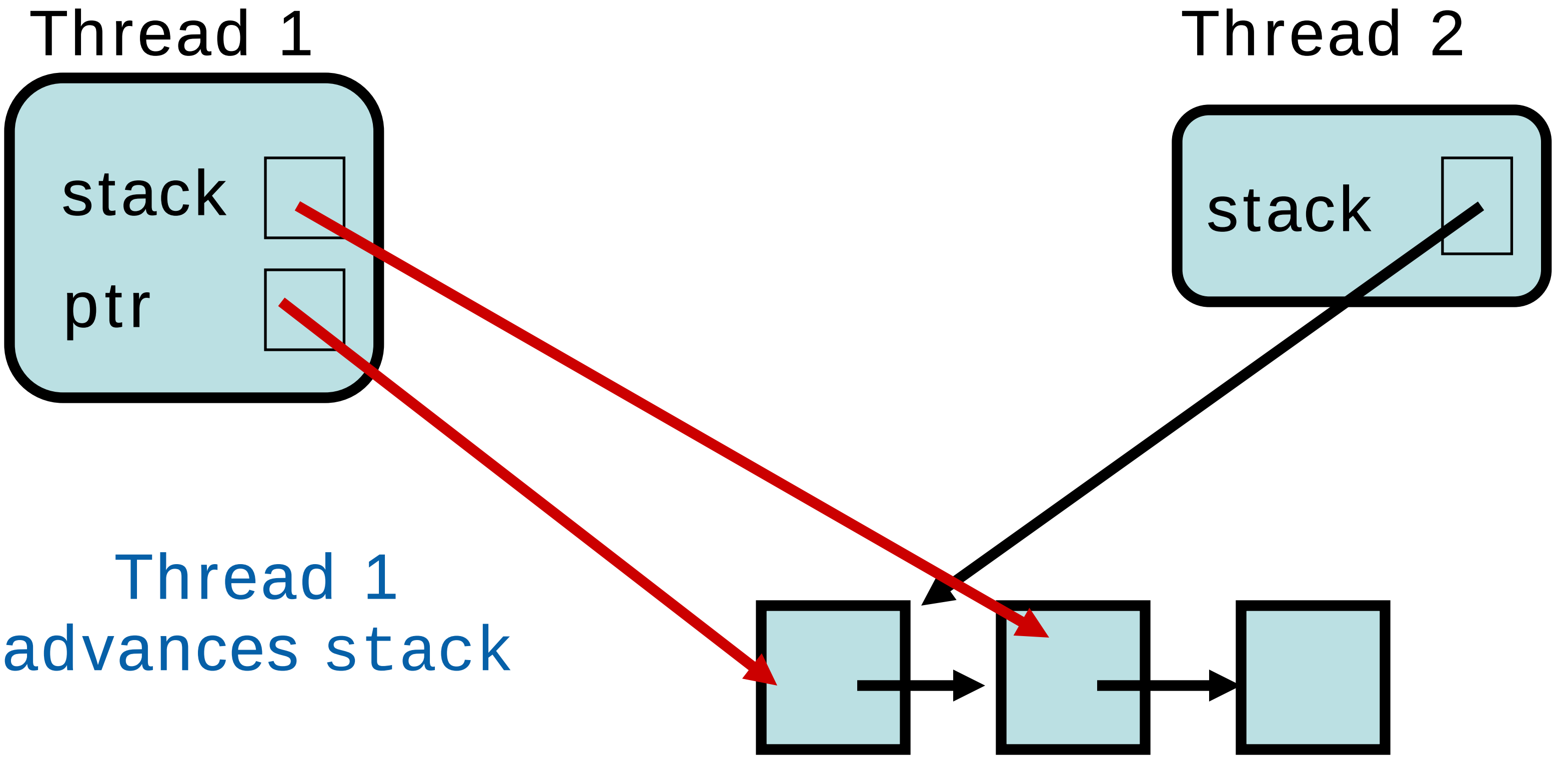

Thread 1 gets hits critical region & reads stack

# Demonstrate error #2



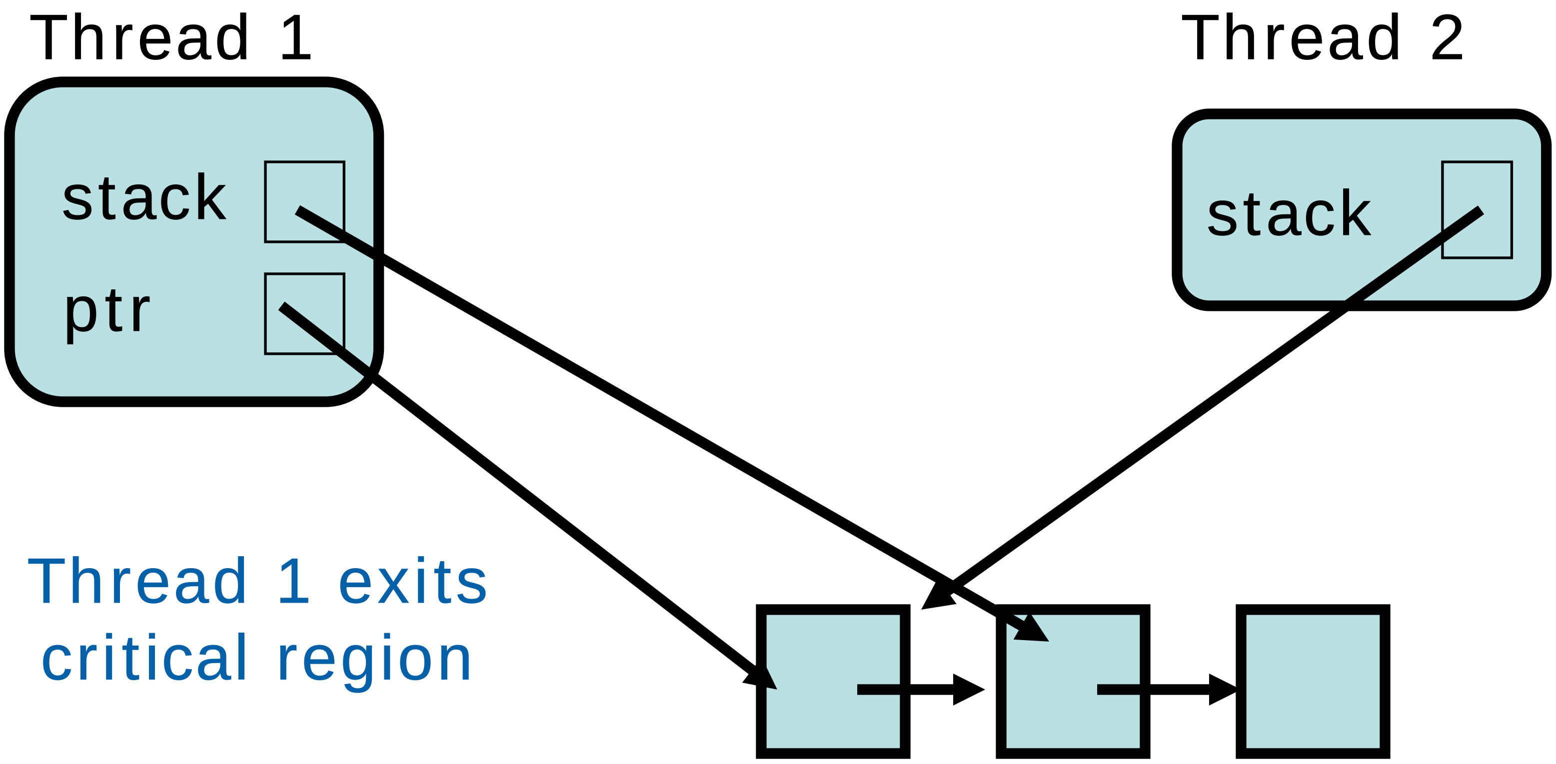

Thread 1 copies
stack to ptr

# Demonstrate error #2



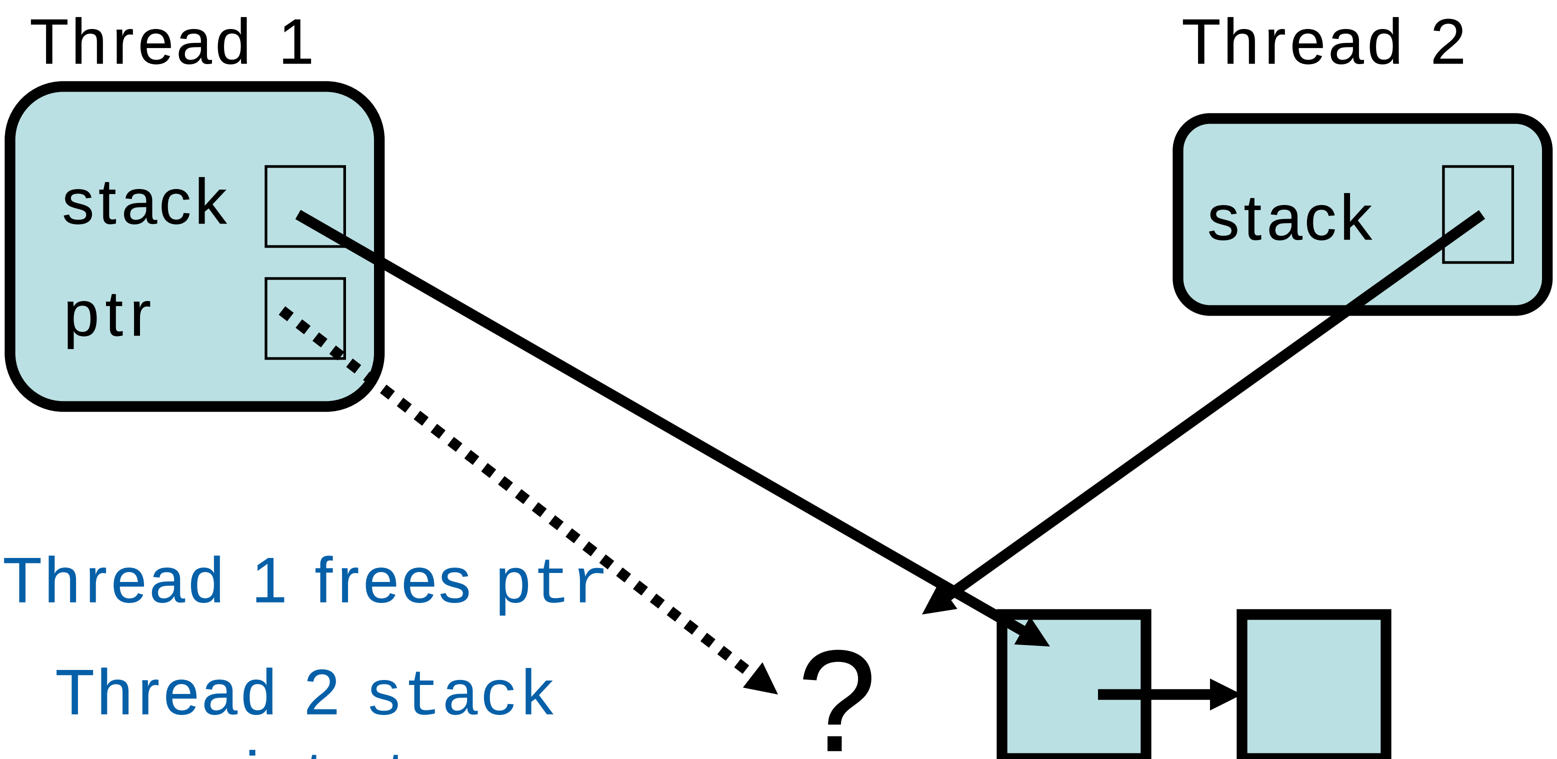

Thread 1
advances stack

# Demonstrate error #2




Thread 1 exits
critical region

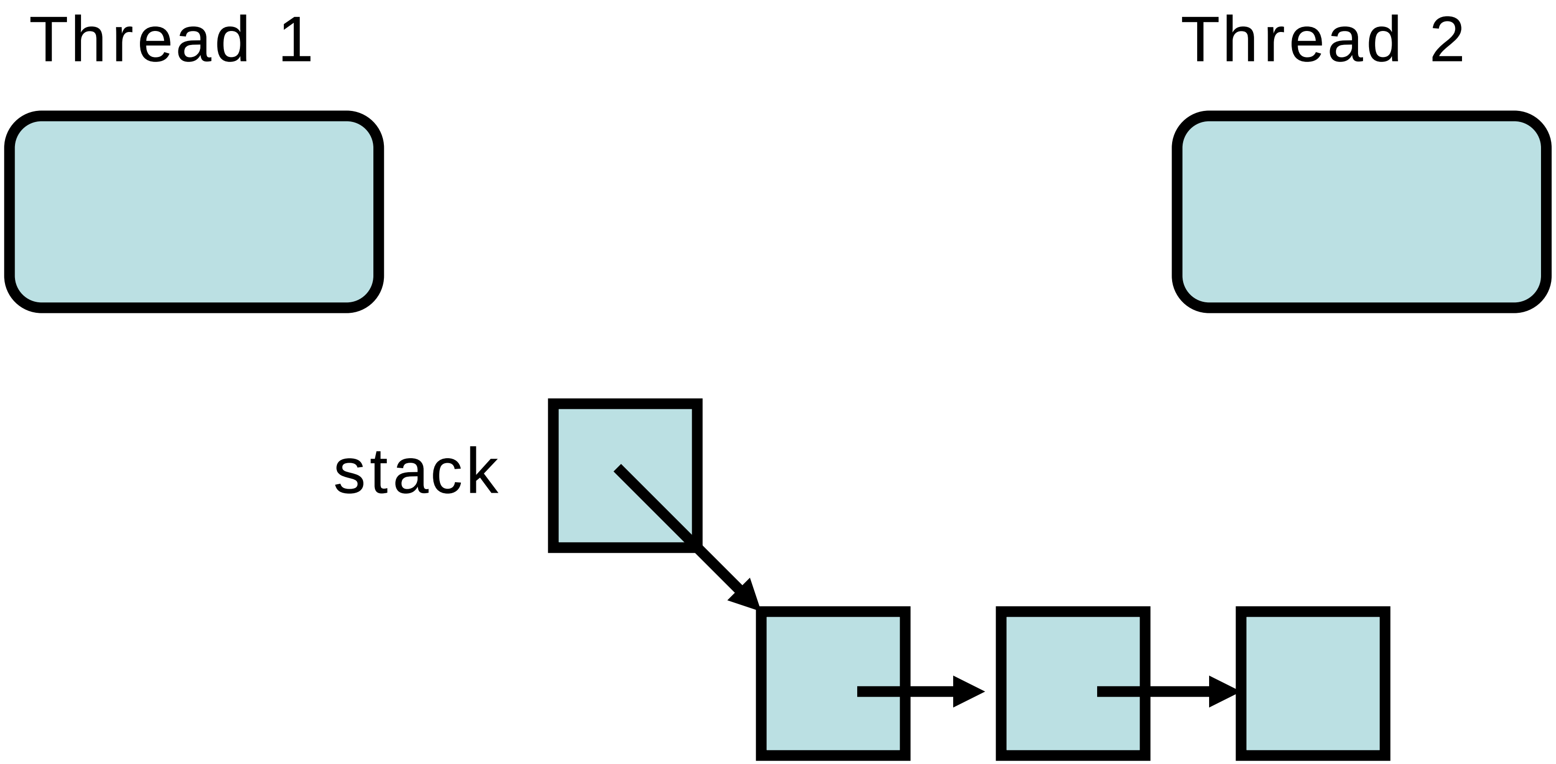
# Demonstrate error #2



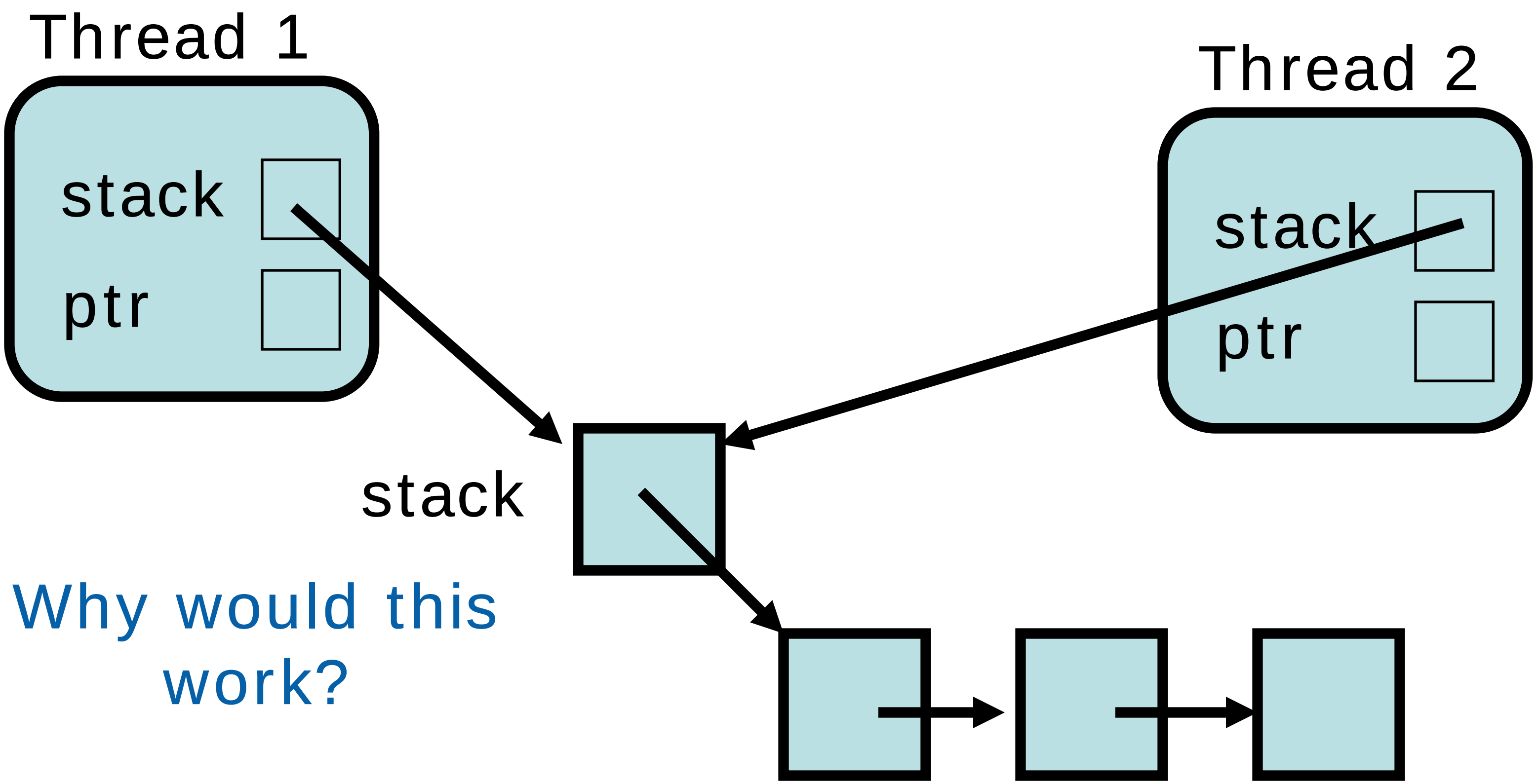

Thread 1 frees ptr

Thread 2 stack points to undefined value

# Remedy 1: Make stack Static

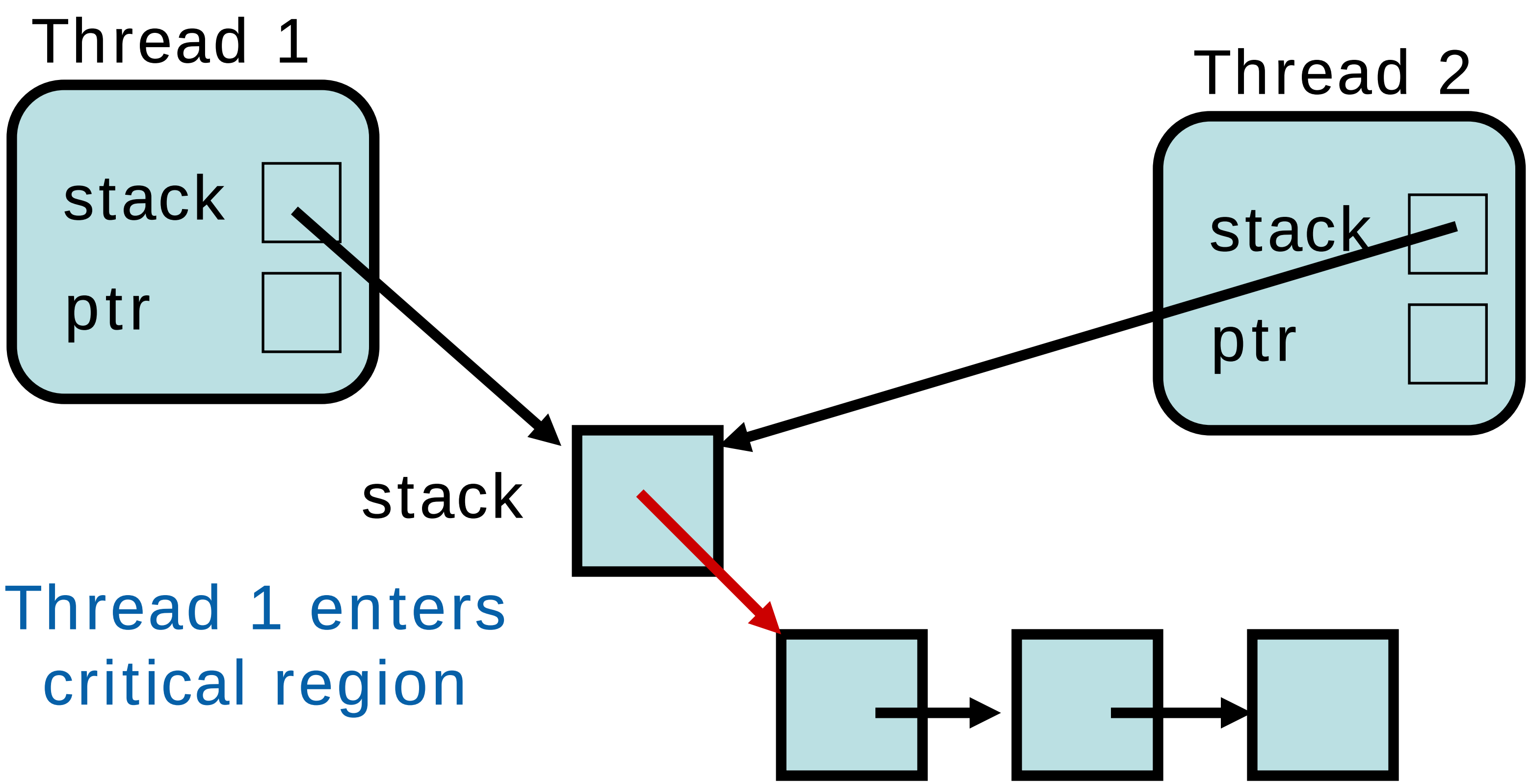


# Remedy 1: Make stack Static



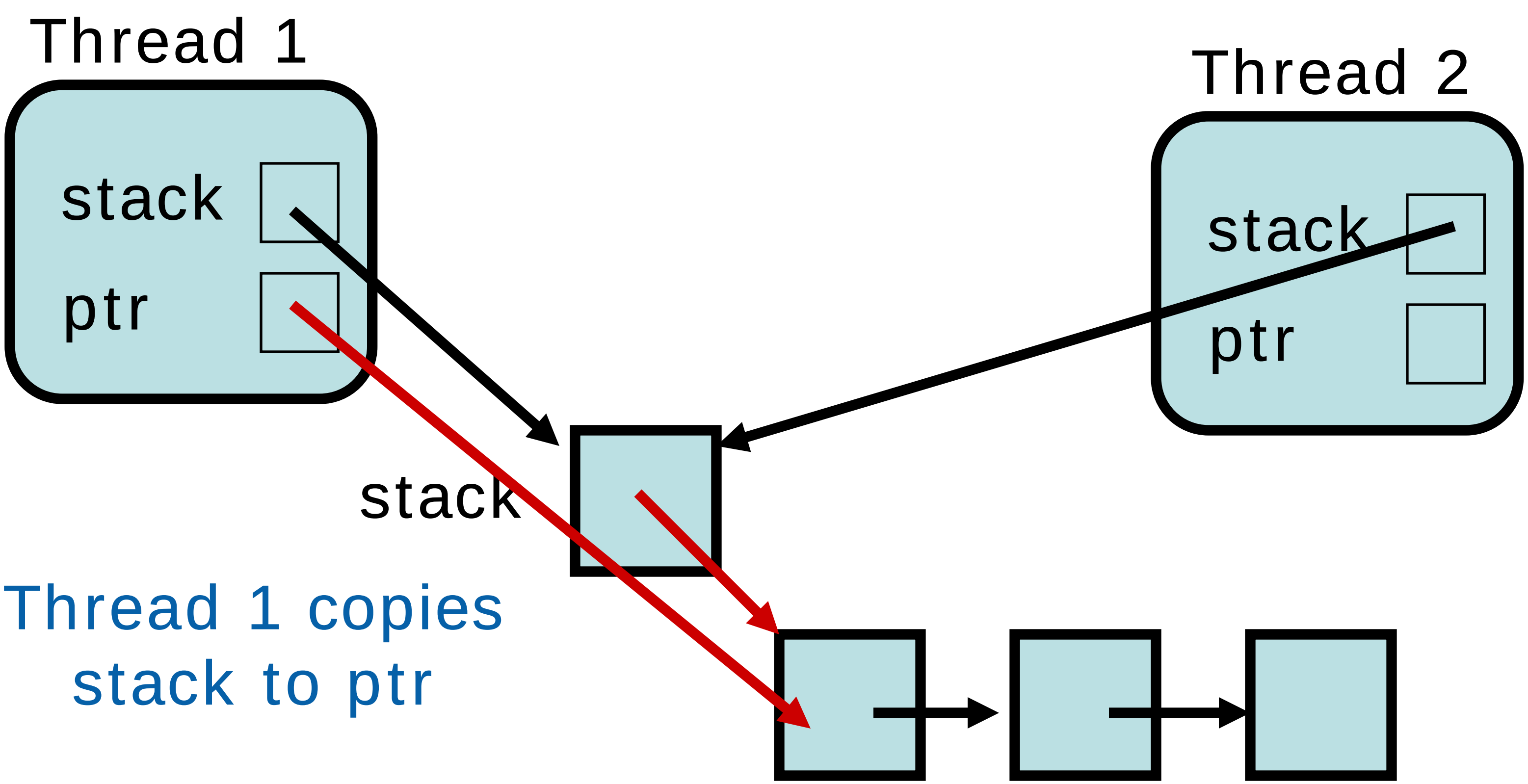

Why would this work?

# Remedy 1: Make stack Static



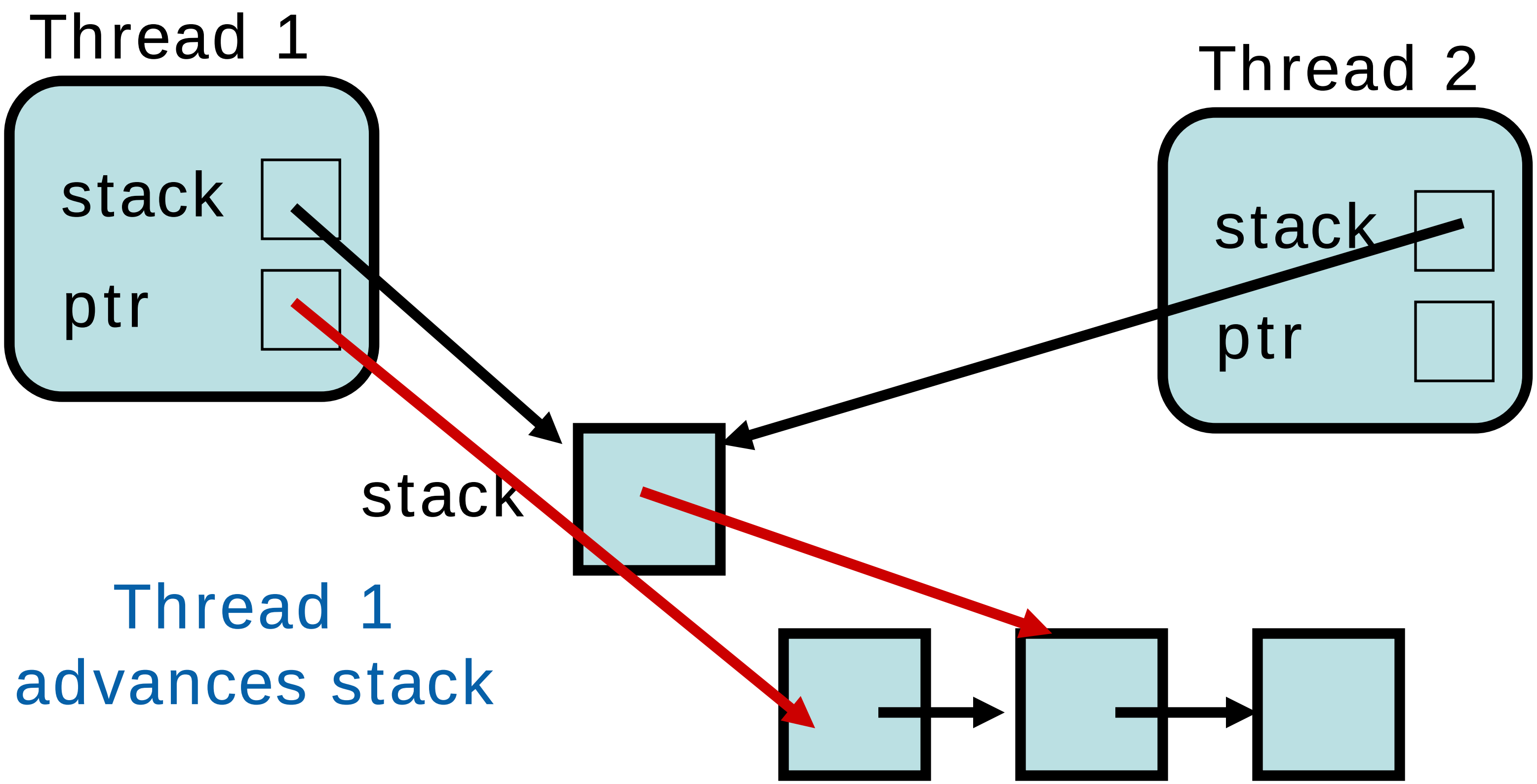

Thread 1 enters critical region

# Remedy 1: Make stack Static



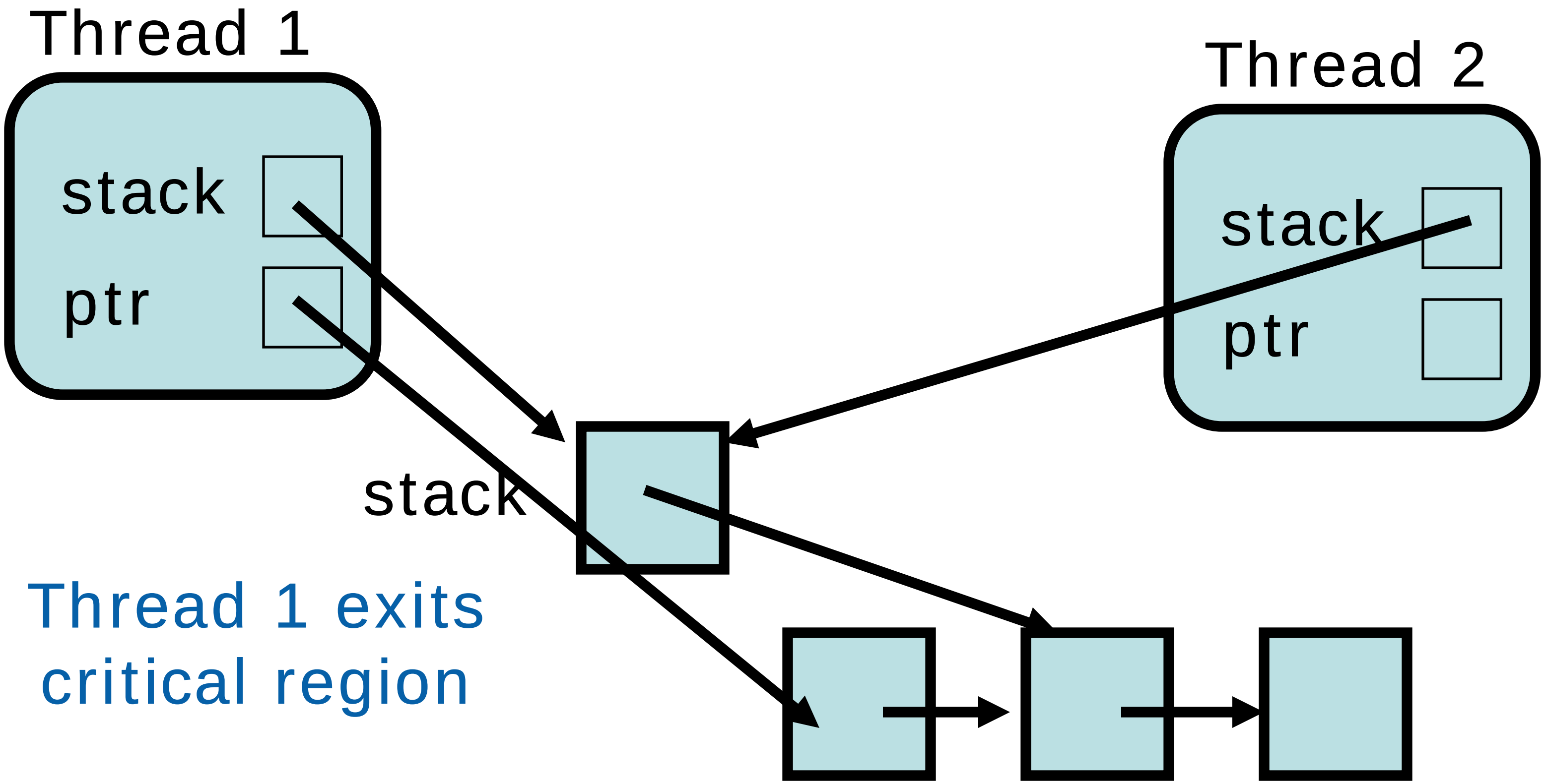

Thread 1 copies
stack to ptr

# Remedy 1: Make stack Static



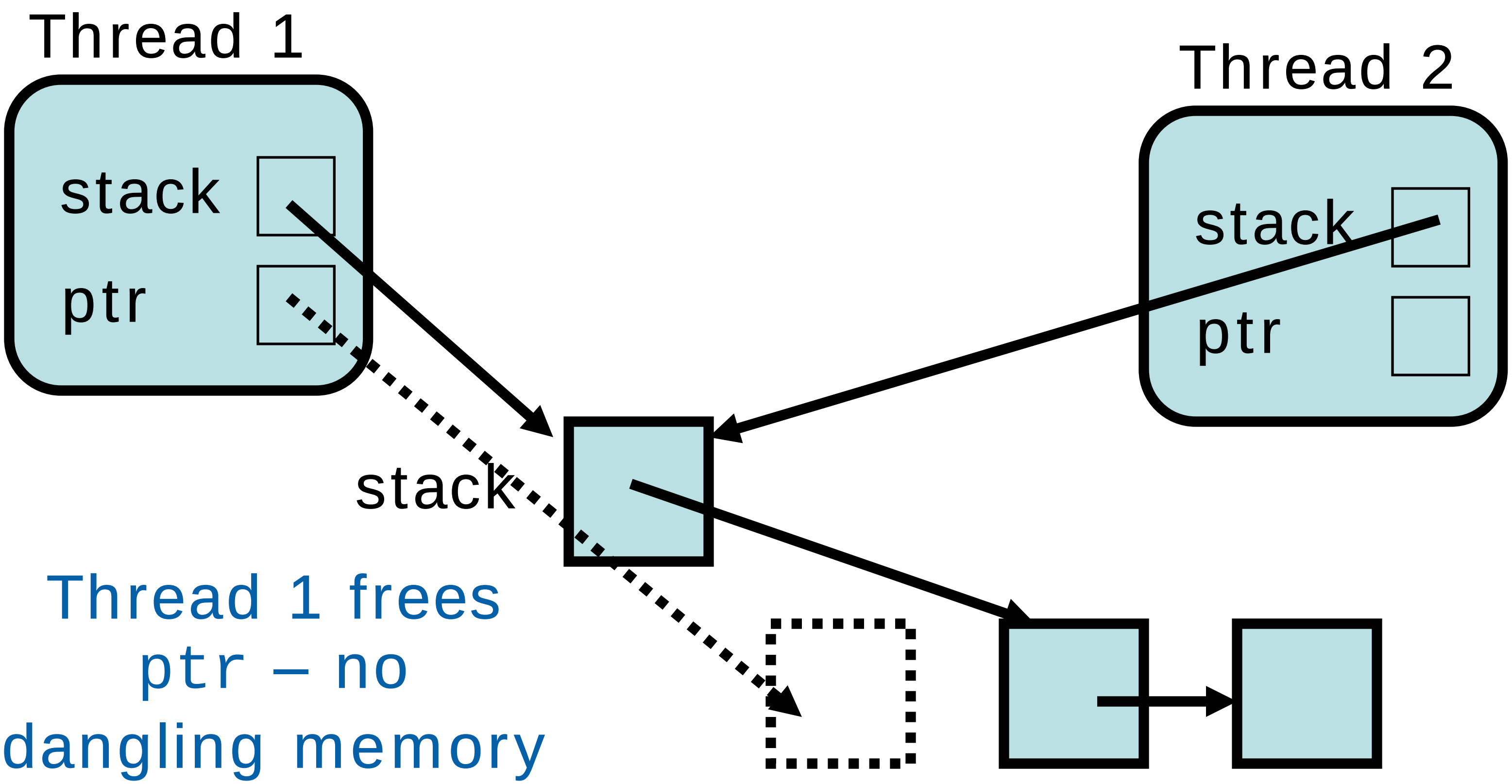

Thread 1
advances stack

# Remedy 1: Make stack Static



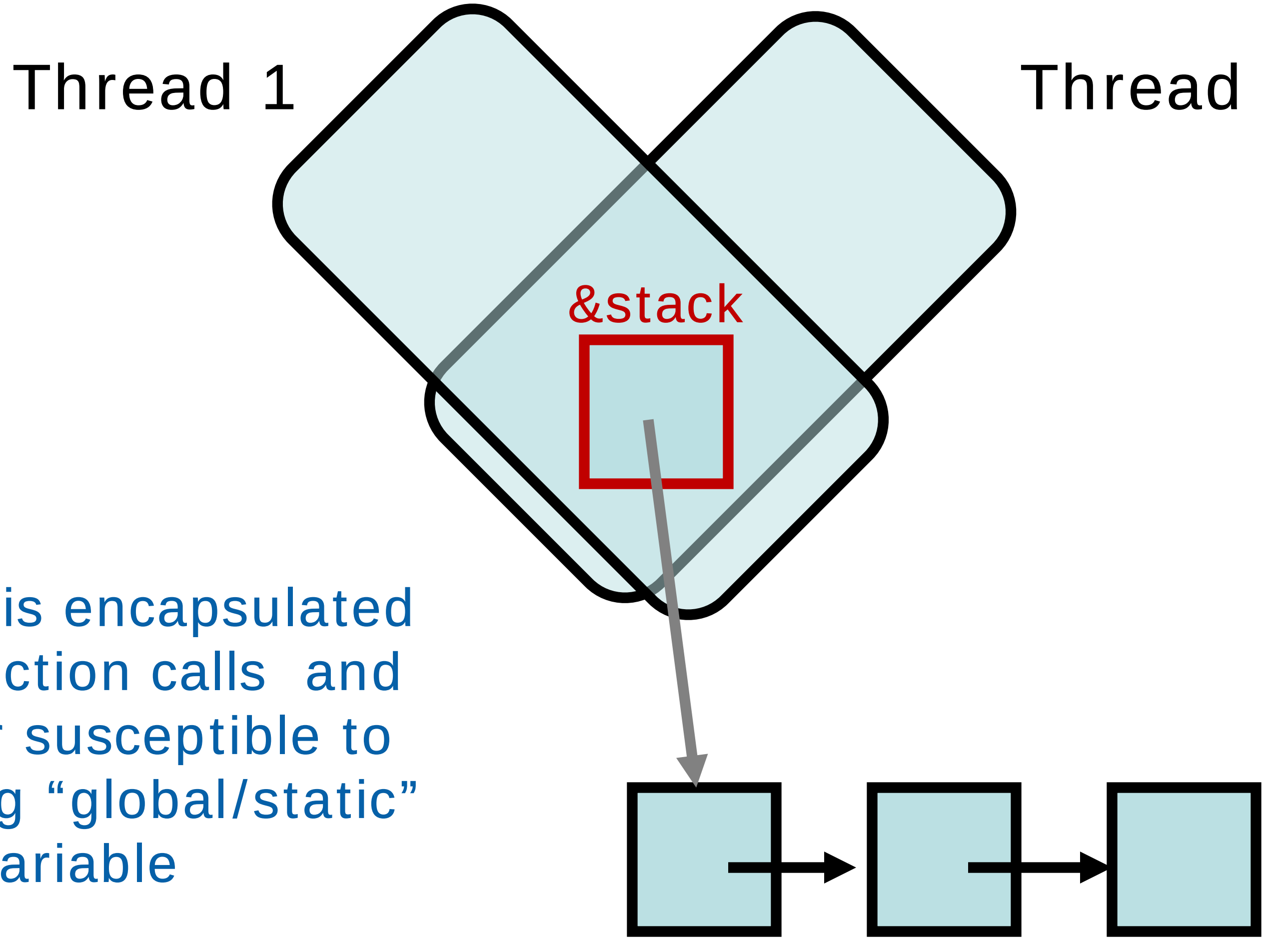

Thread 1 exits
critical region

# Remedy 1: Make stack Static




Thread 1 frees ptr – no dangling memory

# Remedy 2: Use Indirection (Best choice)




Now data is encapsulated inside function calls and no longer susceptible to overwriting "global/static" variable

# Corrected main Function

```
struct board *stack;
...
stack = NULL;
for (i = 0; i < n; i++) {
  initial=(struct board *)malloc(sizeof(struct board));
  initial->pieces = 1;
  initial->places[0] = i;
  initial->next = stack;
  stack = initial;
}
num_solutions = 0;
#pragma omp parallel
 search_for_solutions (n, &stack, &num_solutions);
printf ("The %d-queens puzzle has %d solutions\n", n,
            num_solutions);
```

# Corrected Stack Access Function

```
void search_for_solutions (int n,
   struct board **stack, int *num_solutions)
{
   struct board *ptr;
   void search (int, struct board *, int *);

   while (*stack != NULL) {
  #pragma omp critical
  { ptr = *stack;
        *stack = (*stack)->next; }
     search (n, ptr, num_solutions);
     free (ptr);
   }
}
```

# References

Rohit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon, *Parallel Programming in OpenMP*, Morgan Kaufmann (2001).

Barbara Chapman, Gabriele Jost, Ruud van der Pas, *Using OpenMP: Portable Shared Memory Parallel Programming*, MIT Press (2008).

Michael J. Quinn, *Parallel Programming in C with MPI and OpenMP*, McGraw-Hill (2004).

intel®