

Layout Thrashing Explained

```
const ele = document.getElementById("layoutThrashing");
let val = ele.getBoundingClientRect().width;
for (let index = 0; index < 100; index++) {
    val = val + 1;
    ele.style.width = val + 'px';
}
```

here during js execution, `ele.getBoundingClientRect().width` gets the width from last render. then `val` is a variable that gets incremented, and `width` is increased. here browser makes this element as dirty and 100 times dirty and it will batch and in next render cycle, change the layout to the last value and print.

In below layout thrashing code:

```
const ele = document.getElementById("layoutThrashing");
for (let index = 0; index < 100; index++) {
    ele.style.width = ele.getBoundingClientRect().width + 1 + 'px';
}
```

same like above, js code execution happens but here read and write is happening in loop. Now explain `ele.getBoundingClientRect().width` why is expensive? I understand `ele` is marked as dirty, but why fetching a width of a dirty element is damn expensive?

Great question — you already have most of the mental model right. The missing piece is **what the browser must do when you *read layout* from a “dirty” tree**.

Let's walk through it carefully.

Key concept: “dirty” means *invalid*, not *unknown*

When you do:

```
js
ele.style.width = val + 'px';
```

the browser does **not** immediately recompute layout. It marks parts of the **render tree as dirty**:

- style is invalid
- layout (geometry) is invalid
- paint may be invalid

This is an **optimization**: browsers delay expensive work until they *must* do it.

Case 1: Your first example (cheap)

```
js

let val = ele.getBoundingClientRect().width;
for (let i = 0; i < 100; i++) {
  val = val + 1;
  ele.style.width = val + 'px';
}
```

What happens internally

1. `getBoundingClientRect()`

- Browser computes layout **once**
- Returns a valid width

2. Loop:

- 100 writes to `style.width`
- Browser marks layout as dirty **once**
- No reads → no reason to recompute layout

3. Next render frame:

- Browser applies **final width only**
- Single layout + paint

One layout calculation total

Case 2: Layout thrashing (expensive)

```
js

for (let i = 0; i < 100; i++) {
  ele.style.width = ele.getBoundingClientRect().width + 1 + 'px';
}
```

This is where things change dramatically.

Why `getBoundingClientRect()` is expensive here

Important rule

Layout reads must return *correct, up-to-date values*

When you call:

```
js

ele.getBoundingClientRect().width
```

the browser **cannot** return a cached value if layout is dirty.

If it did, your JavaScript would observe **wrong geometry**, which would break correctness.

What the browser is forced to do (every iteration)

Let's look at one loop iteration:

```
js  
ele.style.width = ...
```

→ Marks layout as dirty

```
js  
ele.getBoundingClientRect()
```

- Browser must flush layout
- Recalculate styles
- Recalculate layout
- Possibly recalculate ancestors & descendants
- Return accurate geometry

Now repeat that **100 times**.

💥 This is called **forced synchronous layout**.

Why layout recalculation is “damn expensive”

Because layout is **global**, not local.

To compute `ele`'s width, the browser may need to:

- Resolve styles (CSS cascade)
- Recalculate layout for:
 - Parent elements
 - Sibling elements
 - Children (depending on layout mode)
- Handle:
 - Flexbox / Grid constraints
 - Percentage sizes
 - Intrinsic sizing
 - Text reflow

Even if you *only* want one element's width, the browser may need to walk **large portions of the render tree**.

📌 That's why layout is one of the most expensive steps in rendering.

Why the browser can't "just give the last value"

Because **JavaScript is synchronous and observable**.

This must always be true:

```
js  
  
ele.style.width = '200px';  
console.log(ele.getBoundingClientRect().width); // MUST be 200
```

So when you read layout:

- The browser **must be correct**
- Even if that means blocking JS execution

Summary (the core answer)

Why `getBoundingClientRect()` is expensive on a dirty element

1. Writing to layout marks the render tree as **invalid**
2. Reading layout **forces the browser to synchronize**
3. Synchronization requires:
 - Style recalculation
 - Layout (reflow)
4. Layout is expensive because it may affect **many elements**
5. Doing this in a loop causes **layout thrashing**

Mental model to keep

Writes are cheap (deferred).
Reads are expensive (force sync).
Mixing them repeatedly is catastrophic.

Best practice

Batch reads and writes separately:

```
js  
  
let width = ele.getBoundingClientRect().width;  
  
for (let i = 0; i < 100; i++) {  
  width++;  
}
```

```
ele.style.width = width + 'px';
```

Or use `requestAnimationFrame` to separate phases.

If you want, I can also explain:

- Why `offsetWidth` behaves the same
- Why `transform: scaleX()` avoids layout
- How browsers track “layout dirtiness” internally

Just tell me 

Browser must flush layout

- Recalculate styles
- Recalculate layout
- Possibly recalculate ancestors & descendants
- Return accurate geometry

so by this u mean that the browser will do layout calculations that it does after RAF is run immediately? This means i am breaking the cycle? if yes, then what happens when browser comes to layout stage after thrashing?

Yes — you’re thinking in the **right direction**, and this is exactly the subtle point most explanations gloss over. Let’s tighten the mental model and answer both parts precisely.

Short answer

-  **Yes**, a layout read (`getBoundingClientRect`, `offsetWidth`, etc.) **forces the browser to run layout immediately**, even if it was planning to do it later in the normal render cycle.
-  You are **not “breaking” the render pipeline**, you are **forcing it to run early and repeatedly**.
-  When the browser later reaches the normal layout stage (after `rAF`), **there is little or nothing left to do**, because layout was already flushed during JS execution.

The real rendering model (important correction)

Think of the render pipeline **not as a fixed loop**, but as a **lazy system**:

The browser does **as little work as possible**, *until* correctness forces it to do more.

Normal (ideal) frame

```
mathematica
  JS
  ↓
  Style calculation
  ↓
  Layout
  ↓
  Paint
  ↓
  Composite
```

But **style/layout/paint** are NOT mandatory steps — they only happen if something *needs* them.

What “flush layout” actually means

When you do this:

```
js
ele.getBoundingClientRect()
```

and layout is dirty, the browser must do:

```
vbnet
JS execution pauses
↓
Style recalculation (only what's needed)
↓
Layout (only what's needed)
↓
Return value to JS
↓
JS resumes
```

This is called a **synchronous layout flush**.

Key point

This work happens **inside the JS call stack**, not at `requestAnimationFrame`.

Are you breaking the render cycle?

No

You are **short-circuiting it**

The browser still *has* a render cycle, but you're forcing parts of it to execute **early** and **many times**.

Timeline comparison

Normal (batched) case

```
text

JS (writes)
JS (writes)
JS (writes)
↓
rAF
↓
Layout (once)
↓
Paint
```

Layout thrashing case

```
text

JS write
JS read → layout flush
JS write
JS read → layout flush
JS write
JS read → layout flush
...
↓
rAF
↓
(no layout work left)
↓
Paint
```

You didn't eliminate layout — you **multiplied it**.

What happens at the browser's normal layout stage after thrashing?

Usually:

- **✓ Nothing**
- Or only **paint / composite**

Why?

Because:

- Layout is already clean
- Geometry is already correct
- No further writes happened after the last forced flush

So the browser says:

"Layout is up-to-date. Skip."

Why this is so bad for performance

1. Layout is expensive

As discussed:

- Tree traversal
- Constraint solving (flex/grid)
- Text reflow
- Dependency resolution

2. It blocks JavaScript

Because:

- JS is single-threaded
- Layout happens **synchronously**
- No other JS or rendering can proceed

3. It destroys batching

Browsers are optimized to:

- Batch multiple writes
- Coalesce layout work

You're disabling that optimization.

Important subtlety: partial vs full layout

Browsers try to be smart:

- Sometimes only a **subtree** is recalculated
- Sometimes layout is **fully global**

But **the browser must assume worst case**, especially with:

- Percent widths
- Flexbox
- Grid
- **auto** sizes
- Fonts

So you should assume:

Any forced layout may cost $O(N)$ of the render tree.

Correct mental model (final)

`requestAnimationFrame` is when the browser *chooses* to render.

Layout thrashing is when JavaScript *forces* rendering work immediately.

One-line takeaway

You're not breaking the cycle — you're repeatedly pulling expensive work *into* the JS execution path.

If you want, I can next explain:

- Why `transform` doesn't trigger layout
- How browsers track "dirty bits"
- Why Chrome DevTools says "Forced Reflow"
- How this interacts with `async`, `await`, and microtasks

Just say the word.