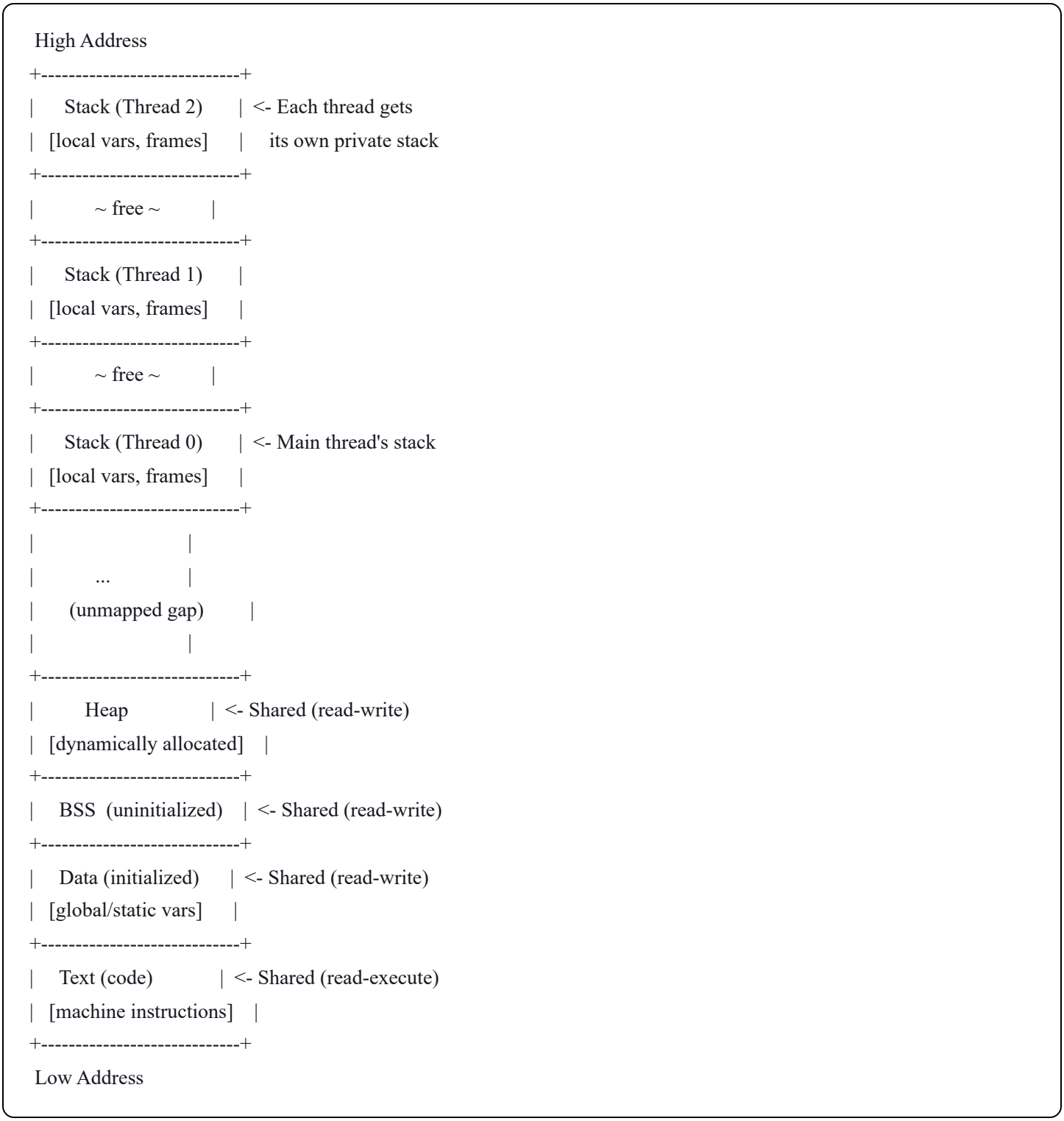
Memory Layout of a Multi-Threaded Process

Process Address Space (Example: 3 Threads)



What Is Shared and What Is Private

Segment	Permissions	Sharing Between Threads

```
+--------------+-------------+-----------------------------------+
|  Text        | r-x         | Shared -- no conflict (read-only) |
|  Data/BSS    | rw-         | Shared -- CONFLICT POSSIBLE       |
|  Heap        | rw-         | Shared -- CONFLICT POSSIBLE       |
|  Stack       | rw-         | Private to each thread            |
+--------------+-------------+-----------------------------------+
```

Why the Text Segment Is Safe to Share

The text segment contains the compiled machine instructions of the program. Its memory protection is set to **read-execute** (`r-x`) -- threads can read and execute the code, but no thread can modify it. Because no writes ever occur, any number of threads can execute the same code simultaneously without conflict.

Why Each Thread's Stack Is Private

Each thread is given its own stack, and the permission on every stack is **read-write** (`rw-`). In a well-structured program, however, each thread accesses only *its own* stack. Since no two threads write to the same stack, there is no conflict.

Local variables of a function -- including parameters and temporaries -- are allocated on the calling thread's stack. This means that **local variables are inherently private to the thread** that called the function. If two threads both call the same function, each gets its own independent copy of all the local variables in its own stack frame.

Why the Data Segment Is Dangerous

The data segment (and the BSS and heap) has **read-write** (`rw-`) permission, and it is shared among *all* threads in the process. Global variables and static variables live here. Every thread sees -- and can modify -- the exact same copy of each global variable.

This is where conflicts arise. If two threads read and write the same global variable concurrently, the result depends on the unpredictable interleaving of their instructions -- a **race condition**.

The Need for Arbitration: The Mutex

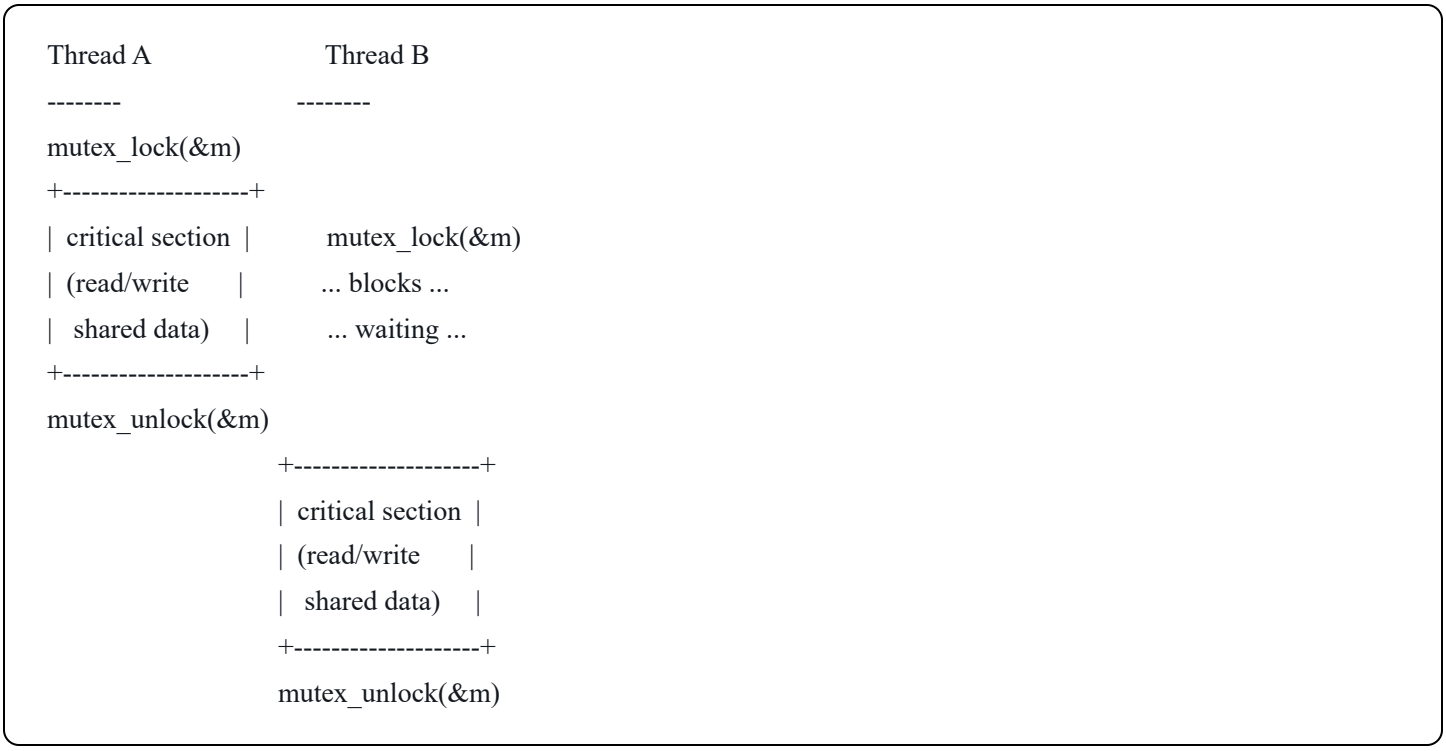
Because multiple threads can concurrently access and modify shared data, we need a mechanism for **arbitration** -- a way to ensure that only one thread at a time is operating on a shared variable or data structure.

The lowest-level programming mechanism for this arbitration is the **mutex** (short for *mutual-exclusion lock*). A mutex works as follows:

1. Before accessing shared data, a thread **locks** the mutex.
2. If the mutex is already locked by another thread, the requesting thread **blocks** (waits) until the mutex becomes available.

3. Once the thread is finished with the shared data, it **unlocks** the mutex, allowing another waiting thread to proceed.

This guarantees that the critical section -- the code that touches the shared data -- is executed by at most one thread at a time, eliminating race conditions.



Summary