Chapter 5a: Memory Layout of a Multi-Threaded Process

Gene Cooperman

Copyright © 2026 Gene Cooperman, gene@ccs.neu.edu
This text may be copied as long as the copyright notice remains and no text is modified.

THIS IS A WORK IN PROGRESS.

1 Memory Layout of a Multi-Threaded Process

1.1 Threads vs. Processes

A **process** is an instance of a running program. It has its own address space (text, data, heap, and stack), its own file descriptors, and its own process ID (PID). When a program is launched, the operating system creates a process with a single thread of execution – this is the **primary thread** (also called the main thread).

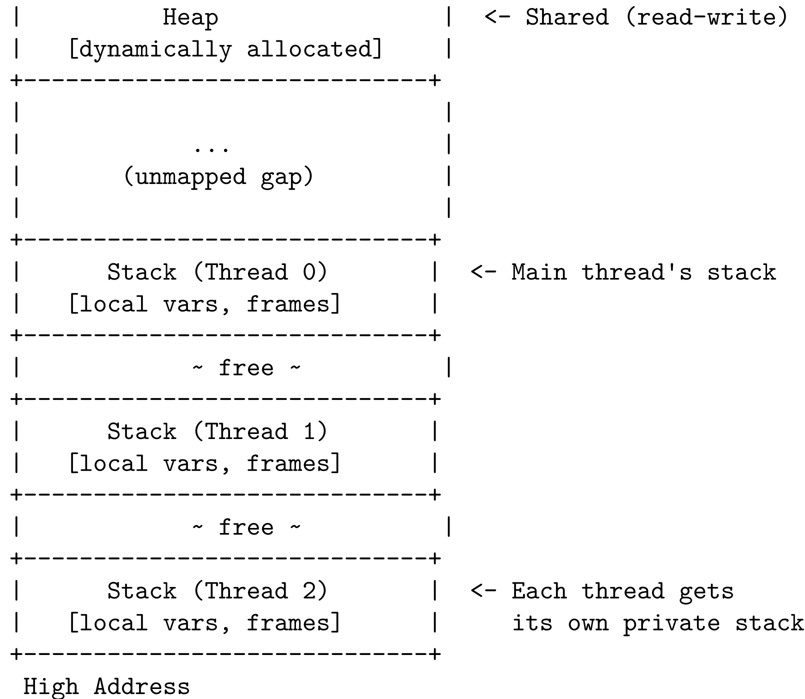
A **thread** is an independent flow of execution *within* a process. All threads in the same process share the same address space – the same text segment, the same data segment, and the same heap. However, each thread has its own stack, its own program counter, and its own set of CPU registers.

The primary thread is, in effect, the process itself. It is the thread that begins executing at `main()` (in C/C++) and it is the thread whose termination typically ends the entire process. Any additional threads created from within the process are **secondary threads**. They run concurrently with the primary thread, sharing the same address space.

In UNIX/Linux, the kernel's **process table** (also called the task list) contains entries for both processes and threads. Each entry is a `task_struct`, and every thread – whether it is a primary thread or a secondary thread – gets its own entry. Primary threads (which represent the process as a whole) and secondary threads are distinguished by their thread group ID (TGID): the primary thread's PID and TGID are the same, while secondary threads share the TGID of their primary thread but have their own unique PID (often called a TID, or thread ID). This is why tools like `ps -eLf` or `top -H` can show individual threads alongside processes.

1.2 Process Address Space (Example: 3 Threads)

```
 Low Address
+------------------------------+
|     Text (code)              |  <- Shared (read-execute)
|   [machine instructions]     |
+------------------------------+
|     Data (initialized)       |  <- Shared (read-write)
|   [global/static vars]       |
+------------------------------+
|     BSS  (uninitialized)     |  <- Shared (read-write)
+------------------------------+
```



1.3 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

1.4 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.

1.5 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.

1.6 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**.

1.7 The Need for Arbitration: The Mutex Lock

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.

```
Thread A                          Thread B
--------                          --------
mutex_lock(&m)
+---------------------+
|  critical section  |            mutex_lock(&m)
|  (read/write       |            ... blocks ...
|   shared data)     |            ... waiting ...
+---------------------+
mutex_unlock(&m)
                                  +---------------------+
                                  |  critical section  |
                                  |  (read/write       |
                                  |   shared data)     |
                                  +---------------------+
                                  mutex_unlock(&m)
```

1.8 Summary

```
+--------------------------+-----------------------------+--------------------------+
| Variable Type            | Stored In                   | Thread Safety            |
+--------------------------+-----------------------------+--------------------------+
```

```
| Local variables          | Stack (private per thread) | Safe -- no sharing        |
| Global / static vars     | Data segment (shared)      | UNSAFE -- needs a mutex   |
| Dynamic (heap) allocs    | Heap (shared)              | UNSAFE -- needs a mutex   |
+--------------------------+----------------------------+---------------------------+
```