Threads and Synchronization in xv6

Operating Systems CSL3030 : Mini-Project

Bhargav Narendra Shekokar (B23CS1008)
Namya Dhingra (B23CS1040)
Aashcharya Gorakh (B23ES1001)

November 16, 2025

Abstract

This report presents a series of kernel-level modifications to the xv6 operating system, designed to explore and implement fundamental concepts in process and thread management. The work details the design and implementation of four key components: (1) a precise `waitpid` system call for selective process reaping, (2) a kernel-level barrier for multi-process synchronization, (3) a lightweight, kernel-level threading system (`thread_create`, `thread_join`, `thread_exit`) sharing a common address space, and (4) an atomic, userspace spinlock for mutual exclusion. Each implementation is validated against provided test cases, and the report includes a detailed analysis of the technical challenges encountered, particularly regarding memory management and state transitions in a concurrent kernel environment.

Contents

1 Introduction

The primary objective of this project was to gain a practical understanding of how core synchronization and concurrency primitives are implemented at the operating system kernel level. The xv6 operating system, a modern re-implementation of Unix V6, was selected as the target environment. Its minimal yet complete design provides an ideal platform for kernel modification, offering a balance between functional complexity and pedagogical clarity.

This report bridges the gap between theoretical operating systems concepts and their practical application. While lectures cover the "what" and "why" of process lifecycles, concurrency, and mutual exclusion, this project addresses the "how." By directly modifying the `proc` structure, managing page tables, and implementing synchronization logic, the authors were able to observe the tangible effects of these high-level concepts within a monolithic kernel.

The project was divided into four distinct parts, each building upon the last:

- **Part A:** Implementing the `waitpid(pid)` system call.
- **Part B:** Introducing a single-use kernel barrier for process synchronization.
- **Part C:** Extending xv6 to support kernel-level threads inside a process.
- **Part D:** Creating userspace spinlocks using atomic x86 instructions.

All development was performed within the QEMU emulator, using a rapid edit-compile-debug cycle to analyze kernel behavior, including kernel panics and stack traces, to validate the implementations.

2 Part A: Implementing the `waitpid` System Call

Goal

The goal was to create a new system call, `int waitpid(int pid);`, capable of reaping **only the specified child process**. Return value semantics were required to match standard UNIX behavior: **PID** on successful reap, or **-1** if the specified child does not exist, is not a child of the caller, or if the caller has no children. A child reaped by `waitpid` must not be reaped again by a subsequent `wait()`.

Files Modified

- `syscall.h`, `syscall.c`: Added syscall number and table entry.
- `sysproc.c`: Implemented the `sys_waitpid` wrapper.
- `defs.h`: Added the kernel function prototype.
- `proc.c`: Implemented the core kernel-side `waitpid` logic.

Implementation Summary

The `waitpid` function was modeled after the existing `wait()` implementation, as both require iterating over the `ptable` with the `ptable.lock` held. The logic was modified to scan for a *specific* PID among the calling process's children. If the target child is found in the `ZOMBIE` state, its resources are freed, and its `proc` struct is set to `UNUSED`. If the child exists but is not a zombie, the parent calls `sleep()` on its own channel, to be woken up when any child (including, hopefully, the target) exits. This logic directly manipulates the xv6 process state model, managing the transition of a child from the 'ZOMBIE' state to 'UNUSED' and ensuring proper resource cleanup.

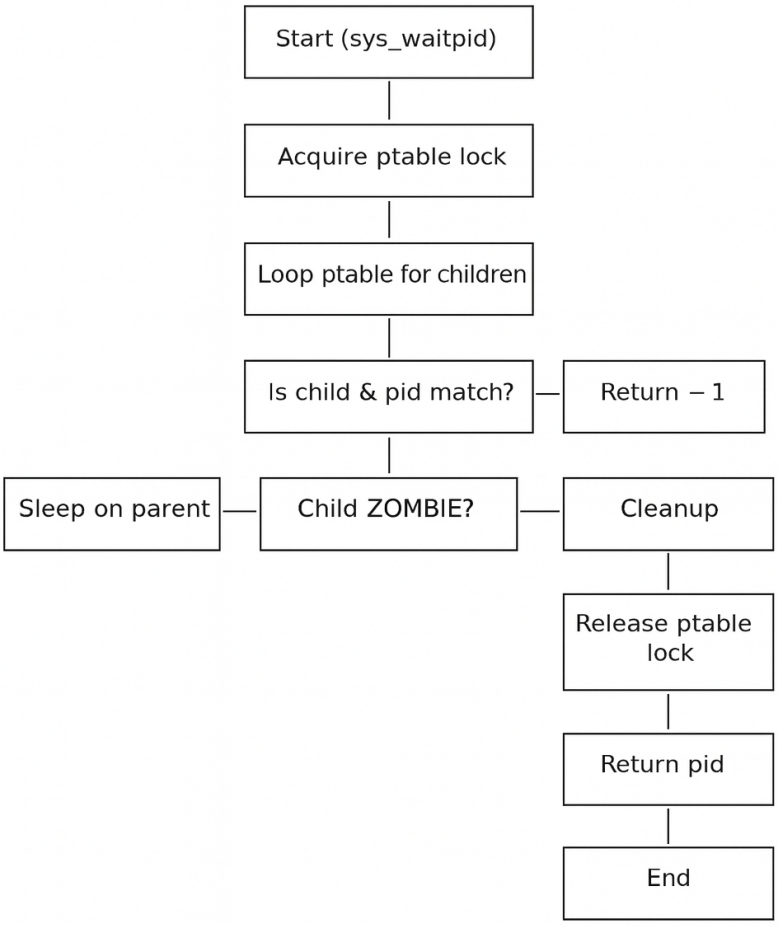

Figure 1: Flowchart of the `waitpid` kernel logic, showing the ptable scan and state check.

Test Output

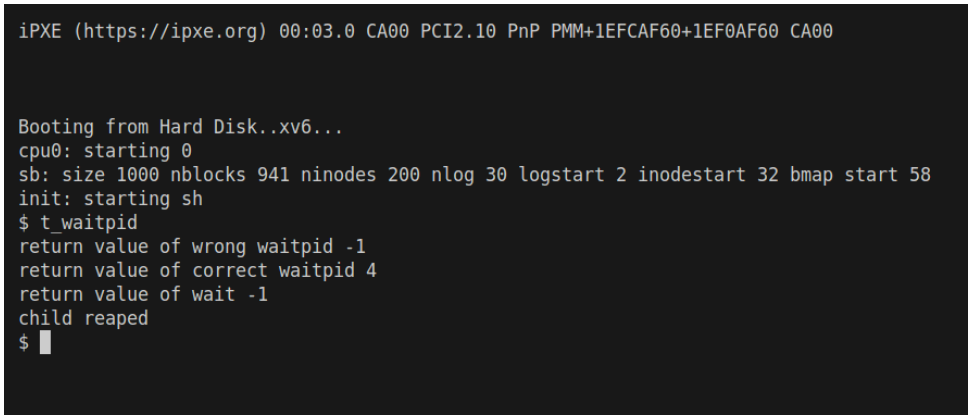

Figure 2: Execution of `t_waitpid`, validating correct PID-specific reaping.

The test output in Figure 1 confirms correct functionality. The call with an incorrect PID fails (-1), the call with the correct PID succeeds (returning 4), and the subsequent `wait()` fails (-1) as the child has already been reaped.

3 Part B: Kernel Barrier Implementation

Goal

This part required implementing two system calls for a single-use kernel barrier: `int barrier_init(int n)` and `int barrier_check(void)`. The barrier must block the first $N-1$ processes that call `barrier_check()`. Upon arrival of the N-th process, all blocked processes must be released simultaneously.

Files Modified

- `syscall.h`, `syscall.c`: Added entries for `SYS_barrier_init`, `SYS_barrier_check`.
- `sysproc.c`: Implemented `sys_barrier_init` and `sys_barrier_check` wrappers.
- `defs.h`: Added kernel function prototypes.
- `barrier.c` (new file): Implemented the core barrier logic.
- `Makefile`: Added `barrier.o` to the build objects.

Implementation Summary

A global barrier structure was defined in `barrier.c`, protected by a single spinlock to ensure atomicity.

- `barrier_init(n)`: Acquires the lock, sets the target count `n`, resets the current `arrival_count` to 0, and releases the lock.
- `barrier_check()`: Acquires the lock, increments `arrival_count`.

- If `arrival_count < target_count`, the process calls `sleep()` on a unique channel (the address of the barrier object), which atomically releases the lock and yields the CPU.
- If `arrival_count == target_count`, the process calls `wakeup()` on the same channel, moving all sleeping processes to the `RUNNABLE` state.

The process then releases the lock. This implementation provides a clear example of **blocking synchronization**, where waiting processes yield the CPU, contrasting with the busy-wait model implemented in Part D.

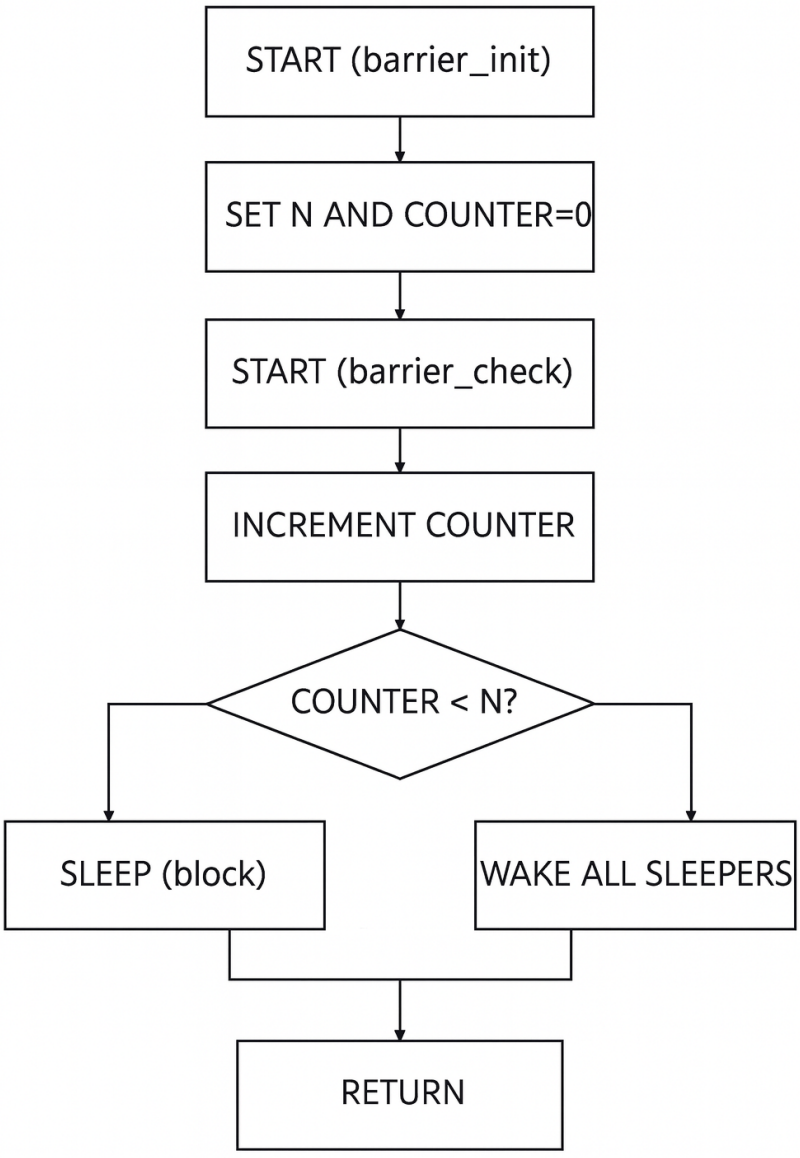

Figure 3: Flowchart of the kernel barrier, illustrating the sleep/wakeup synchronization logic.

Test Output

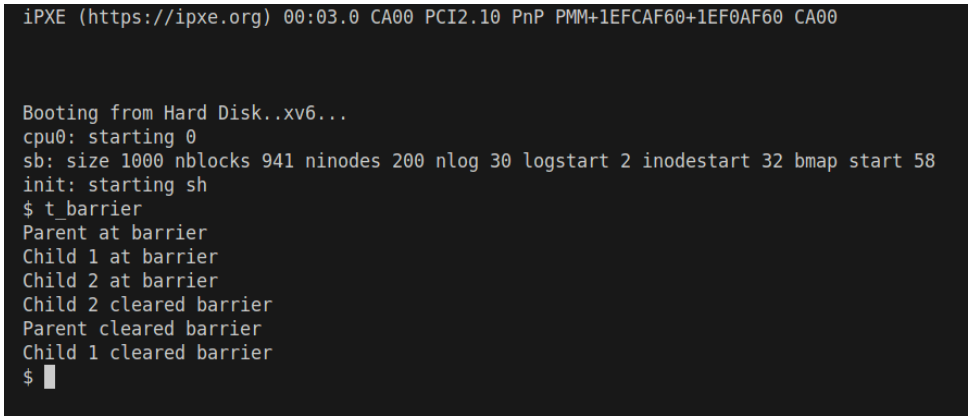

Figure 4: Execution of `t_barrier`, showing synchronized process release.

The test program (Figure 2) validates the barrier. Processes print their "arrived" message but are blocked. Only after the final (third) process arrives are all processes released, printing their "passed barrier" messages together.

4 Part C: Kernel-Level Threads in xv6

Goal

This section involved implementing a kernel-level threading system via three new system calls:

```
int thread_create(uint *tid, void *(*func)(void*), void *arg);
void thread_exit(void);
int thread_join(uint tid);
```

Threads were required to share the parent's address space (`pgdir`) but maintain independent user stacks.

Files Modified

- **proc.h**: Added `isthread` (flag), `mainthread` (pointer to main `struct proc`), and `ustack` (user stack base) fields to `struct proc`.
- **proc.c**: Implemented `thread_create`, `thread_exit`, and `thread_join`. Critically, `fork`, `exit`, and `wait` were modified to correctly handle processes containing threads.
- **syscall.h/c**: Added syscall numbers and table entries.
- **sysproc.c**: Implemented the system call wrappers.
- **user.h**: Added user-facing syscall definitions.

Implementation Summary

This was the most complex part, requiring careful modification of the process lifecycle.

- `thread_create`: This function mimics `fork` but with a crucial difference: it *does not* copy the address space. It calls `allocproc` to get a new `proc` struct, but sets the new thread's `pgdir` to point to the parent's `pgdir`. It allocates a single new page for the thread's user stack, sets the `mainthread` pointer, and manipulates the new thread's trap frame (`tf`) to set `eip` to the function pointer and `esp` to the top of the new stack.
- `thread_exit`: Marks the thread as `ZOMBIE` and awakens the `mainthread` (which may be sleeping in `thread_join`). It does *not* free the address space.
- `thread_join`: Scans the `ptable` for a child thread (a `proc` with `isthread==1` and `mainthread` pointing to the caller). If a `ZOMBIE` child thread is found, it frees the thread's user stack page (`deallocuvm`) and kernel stack, resets the `proc` struct, and returns.
- **Modified `exit` and `wait`**: `exit` was modified to only work for main threads (redirecting others to `thread_exit`). `wait` was modified to reap all child threads along with the main child process being reaped.

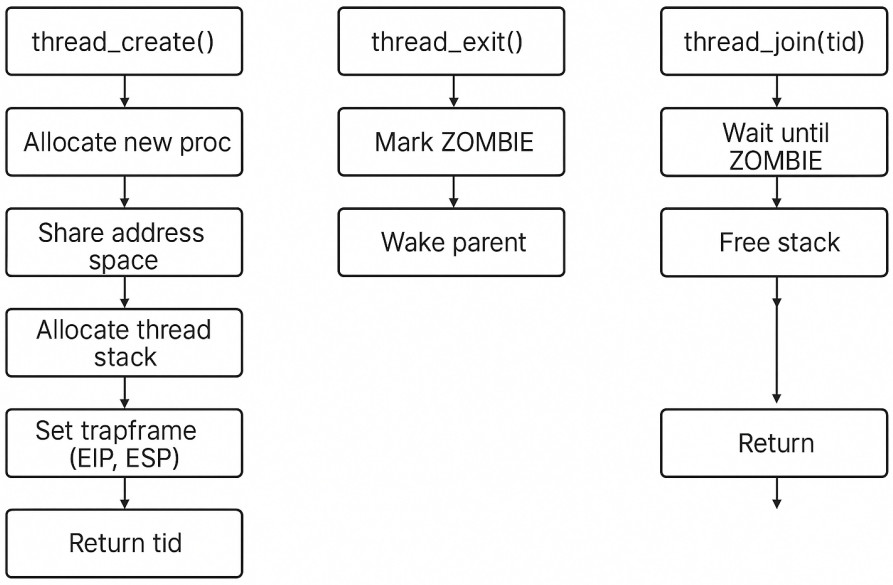

Figure 5: Logic flow for the `thread_create`, `thread_exit`, and `thread_join` functions, highlighting the state management and resource ownership (e.g., stack).

In-Depth Analysis: Kernel Debugging

This implementation was fraught with complex bugs, the analysis of which was highly instructive.

1. Double Free Bug A critical bug was introduced by freeing the thread's user stack page in both `thread_exit` and `thread_join`. This led to page table corruption. **Solution:** A clear ownership model was established: **Only `thread_join` is responsible for freeing the user stack**. `thread_exit` merely transitions the thread's state to `ZOMBIE`.

2. Kernel Panic: "panic: remap" A recurring and difficult-to-diagnose kernel panic, `lapicid 0: panic: remap`, was encountered (Figure 3). This panic indicates a fatal inconsistency in page table mappings.

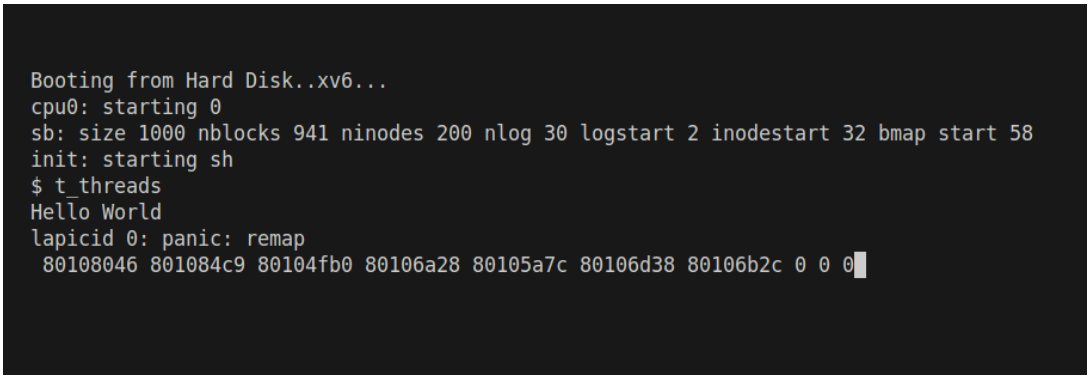

Figure 6: "panic: remap" encountered while running `t_threads`

After extensive debugging, the root cause was traced to incorrect memory management and trapframe initialization. The double-free bug was one cause. Another was that the new thread's trapframe was not correctly re-initialized, causing it to briefly use an incorrect stack pointer (`esp`), which corrupted memory. **Solution:** Resolving this required ensuring `thread_join` has exclusive responsibility for freeing the user stack, and manually setting the new thread's `tf->eip` and `tf->esp` in `thread_create` to guarantee it starts execution on its new stack correctly.

This debugging process was highly instructive, reinforcing the critical importance of careful memory management and state transitions in a kernel environment, where simple errors can lead to system-wide failure.

Test Output

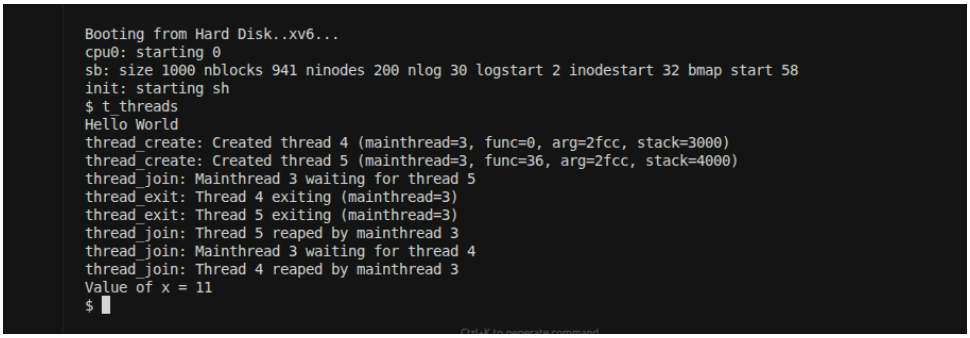

Figure 7: Execution of `t_threads`

The test case (Figure 4) runs successfully, demonstrating that threads are created, share the parent's address space (both incrementing the global `x`), and are correctly reaped by `thread_join`.

5 Part D: Userspace Spinlocks

Goal

The final part was to implement user-level spinlocks using the atomic x86 `xchg` instruction, allowing the threads from Part C to safely access shared data.

Files Modified

- **user.h**: Modified the `struct lock` definition.
- **ulib.c**: Implemented `initiateLock`, `acquireLock`, and `releaseLock`.

Implementation Summary

This implementation was built entirely in userspace.

- `initiateLock`: Initializes the `lockvar` to 0 (unlocked).
- `acquireLock`: Enters a busy-wait loop, repeatedly calling the atomic `xchg` instruction to swap the value 1 with the `lockvar`. The loop terminates when `xchg` returns 0, indicating the lock was successfully acquired.
- `releaseLock`: Atomically sets the `lockvar` back to 0.

The **atomicity** of 'xchg' is the fundamental requirement for building mutual exclusion, as it guarantees that the "test" and "set" of the lock variable is an indivisible operation, preventing race conditions.

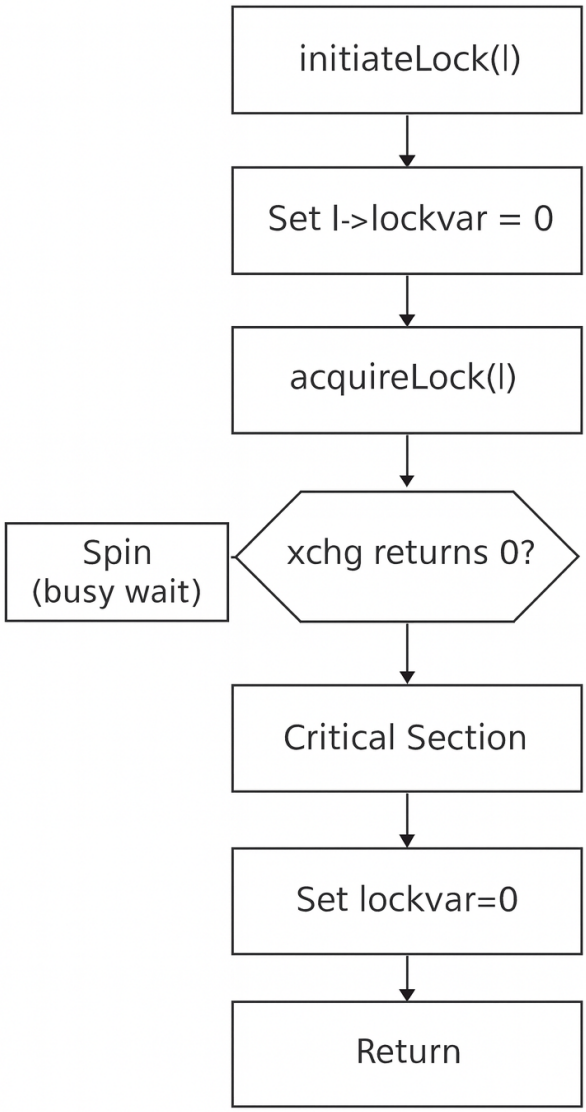

Figure 8: Logic flow for the userspace spinlock, demonstrating the busy-wait loop based on the atomic `xchg` instruction.

Test Output

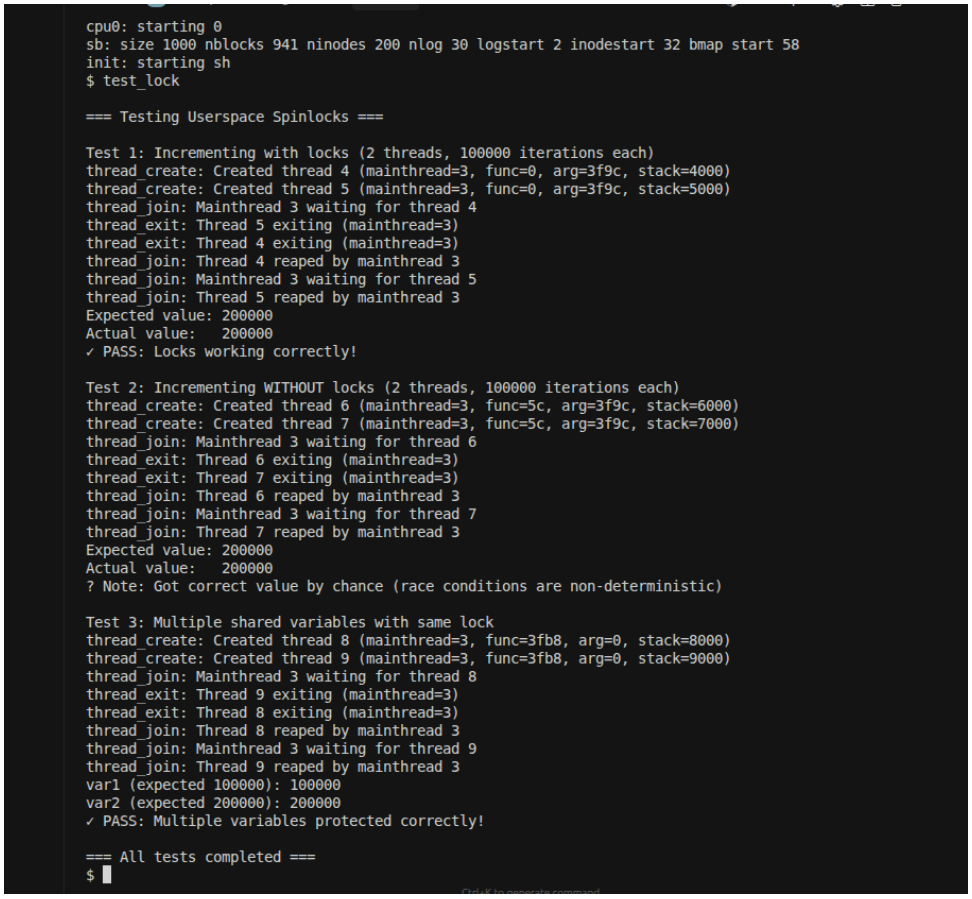

Figure 9: Execution of `t_lock` with 2,000,000 increments.

The test program `t_lock` creates multiple threads that all increment a shared counter. Without a lock, this would result in a race condition. As shown in Figure 5, the final value is exactly 2,000,000, proving that the spinlock correctly guarantees mutual exclusion and protects the critical section.

6 Conclusion and Key Takeaways

This project successfully demonstrated the practical implementation of key operating systems primitives, moving them from theoretical concepts to functional kernel code. The process of building, debugging, and validating each component provided a deep and lasting understanding of the complexities of process and thread management.

More than just completing the tasks, this project provided a clear synthesis of several key OS concepts:

- **Process vs. Thread Memory Model:** The contrast between `fork` (Part A) and `thread_create` (Part C) was profound. While `fork` requires a full address space copy (`copyuvm`), `thread_create` simply shares the parent's `pgdir`. This highlighted the performance benefits of threads but also underscored the debugging challenge of shared memory, as demonstrated by the `panic: remap` issue.
- **Kernel vs. Userspace Synchronization:** The project provided a direct comparison between two synchronization models. The kernel barrier (Part B) used **blocking synchronization** (`sleep`/`wakeup`), which is efficient as it yields the CPU. In contrast, the userspace spinlock (Part D) used **busy-waiting**, which is simpler to implement but consumes CPU cycles. This illustrated the fundamental trade-off between implementation complexity and system performance.
- **The Criticality of Atomicity:** Both the kernel barrier (protecting its counters with a `spinlock`) and the userspace lock (using `xchg`) were fundamentally dependent on an underlying atomic operation. The debugging of `t_threads` further reinforced this: a concurrent system is only as stable as its smallest atomic unit, and errors in state management (like the double-free bug) are catastrophic.

Ultimately, this project was a valuable exercise in system-level debugging, forcing a meticulous approach to memory management, state transitions, and concurrency.

7 References

References

[1] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 2018. (Chapters on xv6 and Concurrency).

[2] Frans Kaashoek, Robert Morris, and Russ Cox. *xv6, a simple, Unix-like teaching operating system*. MIT CSAIL. Available: `https://github.com/mit-pdos/xv6-public`

[3] Operating Systems : CSL3030 (Class Notes) and *Mini Project: Threads and Synchronization in xv6*. Assignment Handout, 2025.