

Mini Project: Threads and Synchronization in xv6

CSL-3030 Operating Systems

Bhargav Narendra Shekokar (B23CS1008)

Namya Dhingra (B23CS1040)

Aascharya Gorakh (B23ES1001)

Abstract

This mini-project explores the implementation of core synchronization and concurrency mechanisms inside the xv6 operating system. The work includes adding a new system call `waitpid`, implementing a kernel-level userspace barrier, building a basic thread system (`thread_create`, `thread_exit`, `thread_join`), and designing userspace spinlocks using atomic instructions. The modifications span `proc.c`, `sysproc.c`, `barrier.c`, `ulib.c`, and related headers.

1 Introduction

This project revolved around understanding how synchronization primitives are actually implemented under the hood of an OS kernel. xv6, being a small yet fully-functional Unix-like OS, offered enough structure to work with while still being minimal enough to understand.

The goals can be summarised as:

- Implementing a `waitpid(pid)` system call.
- Introducing a single-use kernel barrier accessible via system calls.
- Extending xv6 to support threads inside a process, sharing memory but not stacks.
- Creating userspace spinlocks using atomic x86 instructions.

All work was done inside QEMU, which emulates x86 hardware and boots xv6 from the source tree.

Setting up QEMU and xv6

The setup process was fairly straightforward, but getting the environment right was important:

- Installed QEMU using the package manager

```
(git clone https://gitlab.com/qemu-project/qemu.git  
cd qemu  
.configure  
make  
sudo apt install qemu-system-x86  
pip install sphinx==6.2.1 and pip install sphinx-rtd-theme==1.2.2)
```

- Extracted the xv6 lab bundle and copied the patched starter files provided.

```
tar -zxvf cs236-xv6-linux.tgz
```

- Compilation was done using `make`; xv6 was launched using `make qemu-nox` whenever I preferred a terminal-only workflow.

During development, I typically kept two terminals:

1. One running `make qemu-nox` with xv6 booted.
2. Another editing code and re-running make after each change.

This workflow made debugging system calls and kernel panics smoother (once you get used to reading xv6 stack traces).

2 Part A: Implementing the waitpid System Call

Goal

Create a syscall:

```
int waitpid(int pid);
```

that reaps **only the specified child**. Return:

- **PID** — if the child exists and has exited.
- **-1** — if:

- no such child exists,
- or the supplied pid is not a child of the calling process,
- or the calling process has no children at all.

Once a child is reaped via `waitpid`, it should **not** reappear in a subsequent `wait()`.

This matches standard UNIX semantics.

Files Modified

- `syscall.h`: Added `SYS_waitpid`.
- `syscall.c`: Added syscall table entry and extern declaration.
- `sysproc.c`: Implemented `sys_waitpid`.
- `defs.h`: Added prototype `int waitpid(int);`.
- `proc.c`: Implemented kernel-side `waitpid` logic.

Implementation Notes

I modeled `waitpid` after xv6's `wait()`, but added a PID check. The logic is:

1. Scan ptable for children of the calling process.
2. If no child exists or pid doesn't match any child → return -1.
3. If the child exists but is not ZOMBIE:
 - sleep on the parent until the child changes state.
4. If the child is ZOMBIE:
 - free kernel stack,
 - free address space,
 - mark slot UNUSED,
 - return child's pid.

Since `waitpid` requires access to `ptable`, the implementation lives inside `proc.c`.

Test Output

```
iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1EFCAF60+1EF0AF60 CA00

Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ t_waitpid
return value of wrong waitpid -1
return value of correct waitpid 4
return value of wait -1
child reaped
$ █
```

Figure 1: Execution of `t_waitpid`

Expected output:

```
return value of wrong waitpid -1
return value of correct waitpid 4
return value of wait -1
child reaped
```

3 Part B: Kernel Barrier Implementation

Goal

Implement two syscalls:

```
int barrier_init(int n);
int barrier_check(void);
```

A barrier must:

- block the first $N - 1$ processes,
- release all processes when the N -th process arrives.

Files Modified

- **syscall.h**: Added SYS_barrier_init, SYS_barrier_check
- **syscall.c**: Added two syscall entries
- **sysproc.c**: Implemented wrappers
- **defs.h**: Added prototypes
- **barrier.c**: Entire barrier logic implemented here
- **Makefile**: Added **barrier.o**

Implementation Summary

The barrier is implemented with:

- a spinlock,
- a global arrival counter,
- sleep/wakeup primitives.

`barrier_check()` sleeps on the address of the barrier object, using xv6's channel mechanism.

Test Output

```
iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1EFCAF60+1EF0AF60 CA00

Booting from Hard Disk...xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ t_barrier
Parent at barrier
Child 1 at barrier
Child 2 at barrier
Child 2 cleared barrier
Parent cleared barrier
Child 1 cleared barrier
$ █
```

Figure 2: Execution of `t_barrier`

4 Part C: Threads in xv6

Goal

Implement the following syscalls:

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

Threads must:

- share the parent's address space,
- have **independent** user stacks,
- have their own **struct proc** entries,
- track their main thread.

This was by far the most intricate part.

Files Modified

- **proc.h**: Added `mainthread`, `isthread`, `ustack`.
- **proc.c**: Implemented:
 - `thread_create`
 - `thread_exit`
 - `thread_join`
 - Modified `fork` to set `mainthread`
 - Modified `exit` to redirect for threads
 - Modified `wait` to reap threads with `mainthread`
- **syscall.h**: Added syscall numbers
- **syscall.c**: Added entries and externs
- **sysproc.c**: Added wrappers
- **user.h**: Added user-facing syscall definitions

Challenges and Issues Faced

This part consumed most debugging time. Major issues included:

1. Double Free Bug I initially freed stack pages in both `thread_exit` and `thread_join`. This instantly corrupted page tables. The fix: **Only `thread_join` frees the stack.**

2. Stack Pointer Setup Threads were crashing as soon as they started because:

- I misaligned the user stack
- forgot that xv6 grows stacks downward
- added an unnecessary fake return address

3. Shared Memory Pitfalls Since threads share `pgdir`, freeing memory in one thread affected all. The rule became:

Only free thread-private stack pages during `join`. Never touch global memory in `thread_exit`.

4. Kernel Panic: “panic: remap” During `t_threads` During development, I repeatedly hit a very specific and frustrating crash: the system booted normally, printed the initial “Hello World” from the thread test program, and then xv6 immediately panicked with:

```
lapicid 0: panic: remap
```

This panic indicates that the kernel attempted to remap or free a virtual address mapping that was either already unmapped or had become inconsistent due to overlapping mappings in the page table. The screenshot below shows one of these crashes:

```
Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ t_threads
Hello World
lapicid 0: panic: remap
80108046 801084c9 80104fb0 80106a28 80105a7c 80106d38 80106b2c 0 0 0
```

Figure 3: “panic: remap” encountered while running `t_threads`

After several hours of debugging, I traced the root cause to incorrect memory management in my thread implementation. In particular:

- I had mistakenly freed the same user stack page twice — once in `thread_exit` and again in `thread_join`. xv6’s `deallocuvm` is not resilient to double frees and corrupts the page table silently before eventually panicking.
- My initial trapframe copying logic inherited the parent thread’s stack pointer, causing the new thread to run with an ESP pointing into the wrong memory region. As soon as the thread touched its stack, it triggered remap inconsistencies.
- I also briefly allocated stack pages using `allocuvm` without carefully ensuring the new region did not overlap with existing memory. This created partially mapped regions, which xv6 later tried to “fix” and crashed.

Fixing these required three corrections: ensuring that **only** `thread_join` frees the per-thread stack; resetting ESP and EIP manually in the trapframe instead of copying them blindly; and confirming that each new stack page lands at a clean, unused virtual address boundary. Once these changes were made, the remap panic disappeared entirely and the thread tests ran reliably.

Test Output

```
Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ t threads
Hello World
thread_create: Created thread 4 (mainthread=3, func=0, arg=2fcc, stack=3000)
thread_create: Created thread 5 (mainthread=3, func=36, arg=2fcc, stack=4000)
thread_join: Mainthread 3 waiting for thread 5
thread_exit: Thread 4 exiting (mainthread=3)
thread_exit: Thread 5 exiting (mainthread=3)
thread_join: Thread 5 reaped by mainthread 3
thread_join: Mainthread 3 waiting for thread 4
thread_join: Thread 4 reaped by mainthread 3
Value of x = 11
$
```

Figure 4: Execution of `t_threads`

Expected:

```
Hello World
Thread 1 created 10
Thread 2 created 11
Value of x = 11
```

Although it seems different, I have my own version of explaining the following. More than just normal thread creation indication, I believed in displaying the input and exit of the threads as their chain of command was over and finally the value of x.

5 Part D: Userspace Spinlocks

Goal

Implement user-level spinlocks using atomic x86 `xchg` instruction.
Spinlocks must work across multiple threads created in Part C.

Files Modified

- `user.h`: Changed `lockvar` type to `uint`.
- `ulib.c`: Implemented:
 - `initiateLock`

- `acquireLock`
- `releaseLock`

Implementation Summary

The essential idea was:

- `initiateLock`: set `lockvar = 0`
- `acquireLock`: busy-wait using:

```
while(xchg(&l->lockvar, 1) != 0);
```
- `releaseLock`: simply set `lockvar = 0`

Once my thread implementation stabilized, this part was straightforward.

Test Output

```
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ test_lock

==== Testing Userspace Spinlocks ====

Test 1: Incrementing with locks (2 threads, 100000 iterations each)
thread create: Created thread 4 (mainthread=3, func=0, arg=3f9c, stack=4000)
thread create: Created thread 5 (mainthread=3, func=0, arg=3f9c, stack=5000)
thread join: Mainthread 3 waiting for thread 4
thread exit: Thread 5 exiting (mainthread=3)
thread exit: Thread 4 exiting (mainthread=3)
thread join: Thread 4 reaped by mainthread 3
thread join: Mainthread 3 waiting for thread 5
thread join: Thread 5 reaped by mainthread 3
Expected value: 200000
Actual value: 200000
✓ PASS: Locks working correctly!

Test 2: Incrementing WITHOUT locks (2 threads, 100000 iterations each)
thread create: Created thread 6 (mainthread=3, func=5c, arg=3f9c, stack=6000)
thread create: Created thread 7 (mainthread=3, func=5c, arg=3f9c, stack=7000)
thread join: Mainthread 3 waiting for thread 6
thread exit: Thread 6 exiting (mainthread=3)
thread exit: Thread 7 exiting (mainthread=3)
thread join: Thread 6 reaped by mainthread 3
thread join: Mainthread 3 waiting for thread 7
thread join: Thread 7 reaped by mainthread 3
Expected value: 200000
Actual value: 200000
? Note: Got correct value by chance (race conditions are non-deterministic)

Test 3: Multiple shared variables with same lock
thread create: Created thread 8 (mainthread=3, func=3fb8, arg=0, stack=8000)
thread create: Created thread 9 (mainthread=3, func=3fb8, arg=0, stack=9000)
thread join: Mainthread 3 waiting for thread 8
thread exit: Thread 9 exiting (mainthread=3)
thread exit: Thread 8 exiting (mainthread=3)
thread join: Thread 8 reaped by mainthread 3
thread join: Mainthread 3 waiting for thread 9
thread join: Thread 9 reaped by mainthread 3
var1 (expected 100000): 100000
var2 (expected 200000): 200000
✓ PASS: Multiple variables protected correctly!

==== All tests completed ====
$ █
```

Figure 5: Execution of t_lock

Expected:

Final value of x: 2000000

6 Conclusion

Implementing these four parts helped me understand the subtle but crucial differences between processes, threads, and synchronization primitives in xv6. `waitpid` taught me to navigate ptable safely; the barrier illustrated xv6's

sleep/wakeup mechanism; threading forced me to debug real concurrency issues at the OS level; and spinlocks tied everything together.

The project was a deep dive into xv6's internals, and I now feel significantly more confident working close to the kernel.