A Beginner's Guide to the xv6 Scheduler

Scheduling is one of the most important responsibilities of an operating system. In xv6, the scheduler is intentionally simple so we can understand the essential ideas without getting lost in unnecessary complexity.

This lecture explains:

- What kind of scheduler xv6 uses
- How the scheduling loop works
- How xv6 switches between processes
- How timer interrupts cause preemption
- Why locking is necessary
- How everything fits together in practice

By the end of this lecture, you should be able to

- Understand the basic functionality of a scheduler
- Describe how xv6 chooses which process to run
- Understand the meaning of process states (RUNNABLE, RUNNING, etc.)
- Understand why xv6 needs locks inside the scheduler
- Describe what `mycpu()` returns and why it matters
- Walk through how `swtch()` moves the CPU from the scheduler into a process
- Understand how a timer interrupt causes preemption and calls `yield()`

1. What Kind of Scheduler Does xv6 Use?

xv6 uses a **round-robin scheduler**. This means:

- Every RUNNABLE (ready) process gets a chance to run.
- The OS walks through the process table in order.
- The first RUNNABLE process it finds is chosen.
- After a short time slice (a **time quantum**), the process is preempted.
- The scheduler continues scanning and eventually runs the next RUNNABLE process.

2. The Process Table (`proc[]`)

xv6 keeps information about every process in a global array:

struct proc proc[NPROC]; // found in proc.c

Each entry contains both metadata and runtime state about the process, including:

- process ID (pid)
- page table
- registers
- file descriptors
- process name
- and most importantly: the **state**

The possible states are:

`UNUSED, USED, SLEEPING, RUNNABLE, RUNNING, ZOMBIE`

The scheduler only concerns itself with processes whose state is **RUNNABLE**.

3. Locks in xv6

xv6 must protect shared kernel data on multiprocessor systems.

Every process has its own lock:

```
struct spinlock lock;
```

Why do we need this?

Because the scheduler and other kernel code (e.g., `sleep()`, `wakeup()`, `exit()`) may want to modify the same process fields at the same time. To avoid corrupting process state, xv6 requires:

- **Acquire** a process's lock before changing or reading critical fields
- **Release** it after you are done

Without these locks, even basic operations like changing a process's state could race with interrupts or actions from other CPUs.

4. Where the Scheduler Runs

After xv6 initializes all hardware and kernel subsystems, it calls the scheduler:

```
void
main()
{
    // initialize hardware, start first user process...
    scheduler();   // never returns
}
```

This means the CPU core in xv6 permanently sits inside `scheduler()`.

5. Understanding `mycpu()`

Each CPU has its own `struct cpu` object that stores information specific to that core:

```
struct cpu *c = mycpu();
```

Inside this structure is:

```
c->proc
```

which always points to the process currently running on that CPU **(or 0 if the CPU is in the scheduler)**.

This helps xv6 keep track of:

- which process is active
- where to save registers
- how to resume execution

6. The Scheduler Loop Explained

Here is the core of the scheduler (simplified):

```
for(;;){
  for(p = proc; p < &proc[NPROC]; p++){
    acquire(&p->lock);
    if(p->state == RUNNABLE){
      p->state = RUNNING;
      c->proc = p;
      swtch(&c->context, &p->context);
      c->proc = 0;   // when process stops running
    }
    release(&p->lock);
  }
}
```

Step 1 — Forever Loop

The outer `for(;;)` runs forever. The scheduler never exits. Its job is to always find work for the CPU to do.

Step 2 — Scan All Processes

The inner loop examines every entry in `proc[]`:

```
for(p = proc; p < &proc[NPROC]; p++)
```

It looks for any process whose state is **RUNNABLE**.

Step 3 — Lock the Process

Before examining or modifying a process:

```
acquire(&p->lock);
```

This prevents other parts of the kernel from changing its state while the scheduler is working with it.

Step 4 — Run the Process

If a process is RUNNABLE:

```
p->state = RUNNING;

c->proc = p;

swtch(&c->context, &p->context);
```

What happens here?

1. `p->state` is set to `RUNNING`.
2. `c->proc` marks this as the CPU's active process.
3. `swtch(...)` saves the scheduler's registers and restores the process's saved registers.

After `swtch` completes, execution **continues inside the process**, not inside the scheduler. The process resumes exactly where it left off (e.g., after a system call or trap). If a process hasn't started yet, it executes its first line of code and proceeds.

Step 5 — Returning to Scheduler

When the process eventually stops running (because of a timer interrupt, or it voluntarily yields, sleeps, or exits), control returns back into the scheduler **right after the swtch call**.

At that point:

```
c->proc = 0;
```

The CPU is now idle (in the scheduler) and ready to run another process.

7. How a Time Quantum Causes a Context Switch

Every time quantum:

- The hardware timer fires an interrupt.
- xv6 jumps into its trap handler.
- The trap handler notices that this is a timer interrupt.
- It calls:

```
yield();
```

`yield()` does:

- acquire the process lock
- mark the process RUNNABLE again
- call `sched()` → which performs `swtch(&p->context, &c->context)`

The process saves its registers and returns back to the scheduler. Now the CPU is ready to pick the next RUNNABLE process. This cycle repeats forever.