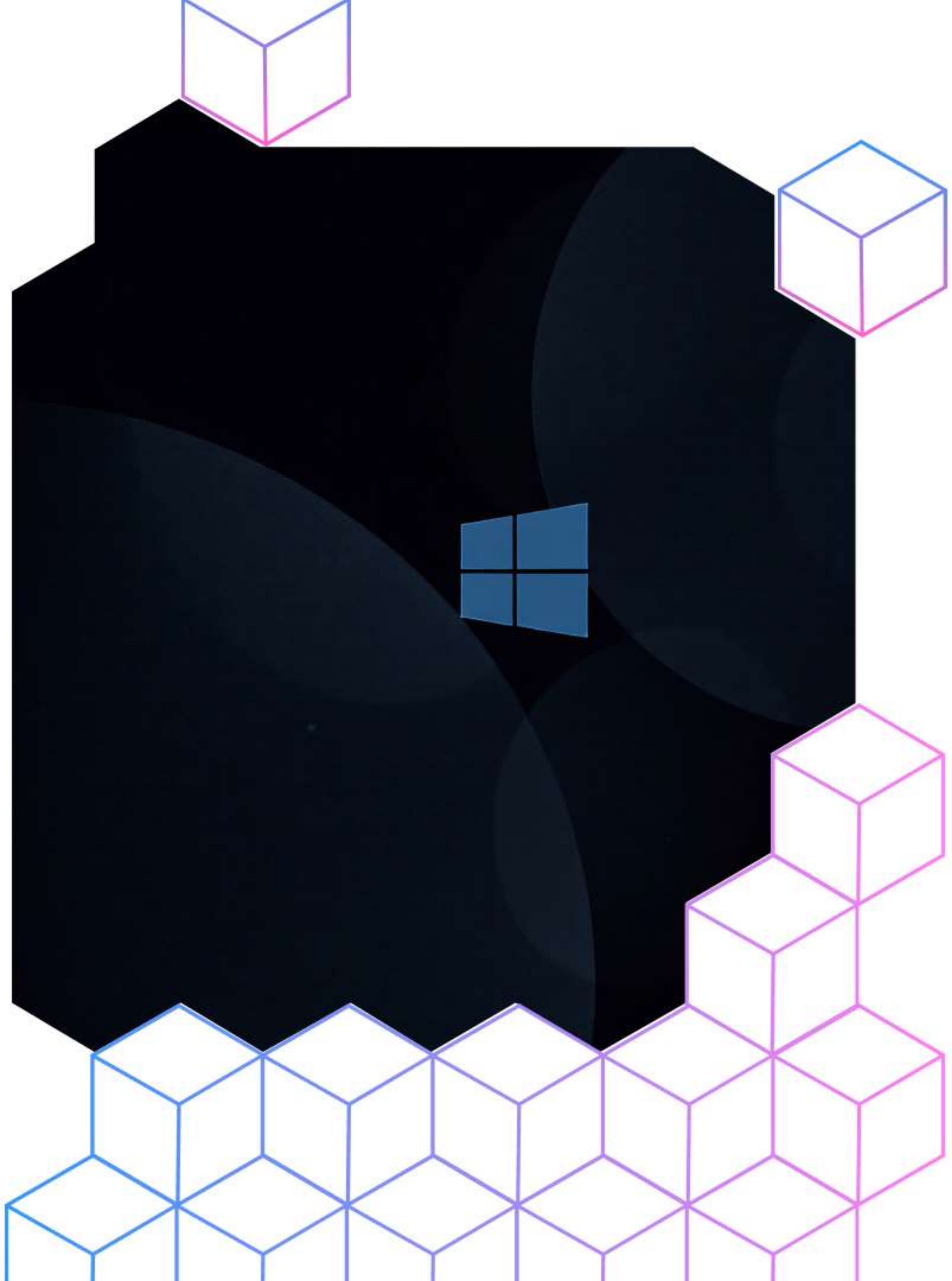


Building a **Minimal OS** using Lottery Scheduling

Compact OS for preemptive multitasking
Design and implement a small OS emphasizing interrupts, scheduler, and randomness.

Build a Minimal OS -- LAN-iX

Hands-on kernel basics: scheduling, interrupts, system calls

1. **Modern kernels are complex**
   Paging, VFS, SMP and many subsystems obscure fundamentals
2. **Project focus areas**
   Scheduling, interrupts, randomness, timer preemption
3. **Educational approach**
   Build a minimal but functional OS from scratch to observe core mechanics
4. **Key learning outcomes**
   Trap and interrupt flow, protected-mode constraints, interrupt latency
5. **Scope trade-offs**
   Omit advanced features like paging to simplify learning and debugging

Project Scope — Included & Excluded

1 Included features

- Used GRUB bootloader for multiboot support, easier implementation.l
- Basic memory model without paging (flat physical memory model for simplicity)
- Simple PCB and context switching to manage task state transitions reliably

2 Included runtime and services

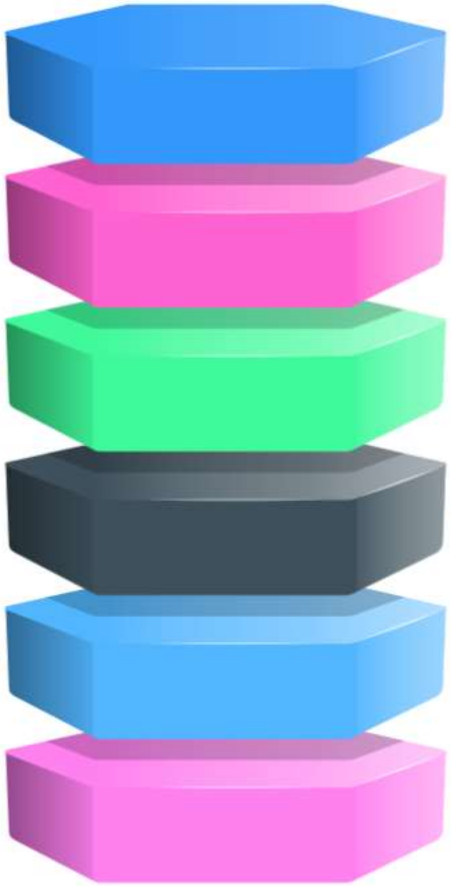
- Lottery Scheduling as the CPU scheduler to explore probabilistic fairness
- Timer interrupts with PIC remapping to enable preemption and proper IRQ routing
- PRNG using CPU timing and interrupt entropy for lightweight randomness support

3 Intentionally excluded features

- Paging and virtual memory to avoid the complexity of memory virtualization
- User mode support to keep the kernel/privilege model minimal for the course
- Filesystem implementation to narrow scope away from persistent storage challenges

System Architecture Overview

Bootloader to scheduler: control and verification checkpoints



1. **Boot Stage**
   GRUB loads kernel.elf at 1MB; verifies boot correctness.

2. **Interrupts and IDT**
   IDT setup for hardware and exception handling; verify vectors and handlers.

3. **PIC Controller**
   Programmable Interrupt Controller routes IRQs to CPU; confirm mask and remap.

4. **Timer Driver**
   Generates periodic ticks for scheduling; ensure reliable timer frequency.

5. **Scheduler and Process System**
   Kernel creates function-based processes with stack and context; scheduler switches on timer ticks.

6. **Randomness Subsystem**
   Provides entropy to kernel components; verify seeding and availability.

Boot and Entry Code: deterministic early startup

Tiny boot.S sets video memory, switches to protected mode, and jumps to kernel_main

GRUB loads kernel.elf

Verify multiboot header presence, alignment, and entry address at 1MB for GRUB compatibility

boot.S initializes video memory

Confirm video memory writes for early debug output are reliable for diagnosing boot issues

Switch to protected mode

Ensure mode transition sequence is correct and CPU is in protected mode before continuing

Jump to kernel_main

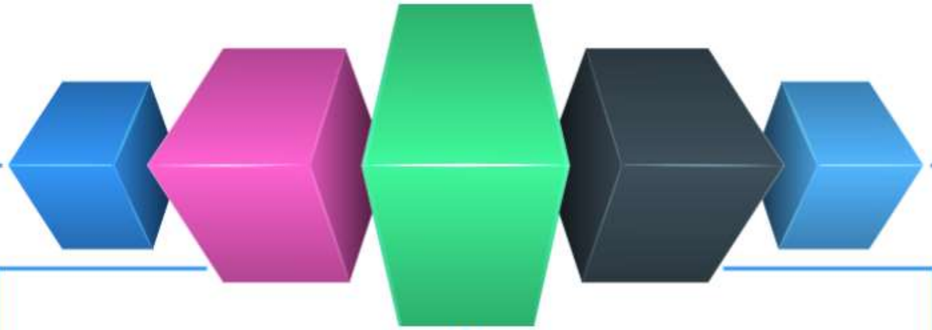
Validate the entry address and that registers/stack meet kernel_main ABI expectations

Initialize IDT

Load IDT early to allow later interrupt based scheduling; verify IDT base and limit

Interrupts and the IDT: Timer, ISR, and Scheduler

How IRQ0 maps to vector 32 and drives preemptive scheduling



1

IDT Setup

IDT configured in idt.c; load_idt() installs the table used by all interrupts.

2

IRQ0 to Vector 32

Timer interrupt (IRQ0) is mapped to vector 32 and invokes the timer ISR.

3

ISR Entry and Exit

Validate handler entry and exit sequences and correct IRQ vector usage.

4

Interrupts Handle

interrupts_handle mixes timing entropy and forwards events to the scheduler.

5

Scheduler Interaction

Interrupts are fully functional and essential for preemptive scheduling; state changes must be atomic relative to scheduler decisions.

Timer Subsystem: **50 Hz** PIT Tick

PIT tick flow, scheduler impact, and ISR guidance

- PIT generates tick
  - Programmable Interval Timer set to 50 Hz issues a hardware tick
- ISR entry
  - Interrupt service routine runs on tick; keep work minimal to preserve latency
- Mix TSC entropy
  - Add TSC-based entropy into PRNG pool during ISR
- Call lottery_schedule_tick
  - Notify scheduler to consider preemption and update runtime accounting
- Send EOI to PIC
  - Acknowledge interrupt to the PIC so new interrupts can be accepted
- Scheduler effect
  - Tick makes OS preemptive; frequency affects responsiveness and overhead
- Practical guidance
  - Balance tick rate for responsiveness versus CPU cost; document fairness and entropy impact

Process Model: fixed-array **PCBs**

MAX_PROCESSES = 10; each process is a function with its own stack

PID

Unique process identifier

State: UNUSED to RUNNABLE to RUNNING

Scheduler and interrupts drive transitions

Tickets

Used by lottery or weighted scheduler

CPU context (ESP saved/restored)

Registers saved on context switch

Stack pointer and stack mapping

Each PCB points to its own stack of 1024 words

Fixed array allocation

Array size MAX_PROCESSES = 10 simplifies allocation

Process = function with independent stack

Function executes using its 1024-word stack

Testing and correctness notes

Consider stack sizing, overflow risks, and interrupt-driven transitions

Diagram: PCB maps to stack

Visual: PCB entry pointing to its 1024-word stack

Context Switching: saving CPU state for multitasking

Preserve registers, stack, and flags to resume another process

- Save registers with pusha
  - Push all general purpose registers to stack to preserve caller and callee state
- Save old ESP
  - Store current stack pointer so the current process can resume with correct stack
- Load new ESP
  - Set ESP to the next process stack so its context is active
- Restore registers with popa
  - Pop registers from the new stack to restore the next process CPU state
- Resume execution
  - Continue execution on the next process stack with preserved EFLAGS and registers
- Preserve critical state bits
  - Ensure callee-saved registers and EFLAGS are preserved; validate stack pointers
- Minimize ISR work
  - Keep interrupt service routines short to avoid long blocking during context switches

Why use **Lottery Scheduling**

Probabilistic fairness with minimal code and flexible weights

VS

Lottery Scheduling

- Fairness via ticket-weighted chance
- Processes request more or less CPU by changing tickets
- Small code footprint, no complex priority queues
- Good for experiments and dynamic ticket assignment
- Illustrates probabilistic scheduling concepts for labs

Round Robin and Priority Queues

- Round robin: simple and predictable time slices
- Priority queues: deterministic prioritization by numeric priority
- Round robin can be unfair for differing needs
- Priority queues require complex data structures and tuning
- Less flexible for runtime, fine-grained fairness experiments

Deterministic Lottery Scheduling with Atomic Ticket Updates

Fair, preemptive selection using a global ticket pool

- Compute total tickets
  - Read global total_tickets; ensure atomic read while processes change ticket counts
- Generate winning ticket
  - winning = rng_get_range(total_tickets) + 1; RNG supplies nondeterministic entropy
- Traverse circular PCB list
  - Iterate PCBs summing tickets until cumulative reaches winning ticket
- Select winner and prepare switch
  - Mark chosen PCB; ensure ticket totals remain consistent during handoff
- Context switch
  - Perform context switch; scheduler runs preemptively on every timer tick
- Atomic updates and edge cases
  - Update ticket totals atomically; handle zero-ticket processes with skip or baseline tickets
- Fairness validation
  - Use test harnesses to run many trials and validate distribution qualitatively in class

Kernel Randomness: PRNG Inputs and Mixing

Design, initialization, and quality considerations

Primary entropy inputs
rdtsc Time Stamp Counter and interrupt timing jitter

Ongoing entropy injection
Inject fresh entropy on every interrupt to avoid staleness

Mixer: SplitMix64
Use SplitMix64 as a fast, well-studied mixer for internal state

Quality considerations
Combine high-resolution timers with jitter; avoid sole reliance on short-lived sources

Initialization requirements
Ensure strong initial seed and well-initialized mixer before use in scheduling

Adversarial/pathological cases
Limited interrupt activity or synchronized timers can reduce entropy; plan fallback or conservative use

Result
High-quality pseudo-random generator inside the kernel for lottery scheduling

Future Work: Extend OS with incremental features

Prioritize quick wins, plan for larger isolation efforts

System calls for processes

Add simple kernel call interface to expose services to processes

Dynamic ticket assignment via kernel commands

Allow runtime control of scheduling tickets for flexibility

Cooperative yield system call

Enable voluntary context switches to improve responsiveness

Improved debugging and kernel prints

More sophisticated logs to speed diagnostics and testing

Paging for memory isolation

Larger effort to provide isolation and realistic testing

User mode and privilege levels

Major work to enable protected user space and security