Week 1–2: Building, Compilation, Makefiles, Linking

Simulation: Build System Visualizer
Students could use a drag-and-drop interface where they place source files, headers, and libraries into a “project box.” When they click *build*, the simulator animates how files are compiled into `.o` object files, combined into `.a` static libraries or `.so` shared libraries, and finally linked into an executable. Students can experiment with modifying a file and see which parts of the build are re-compiled, illustrating how makefiles and dependency rules save time.

Week 2–3: Accounts, Permissions, System Calls, Exceptions, Signals

Simulation: Permissions Sandbox
Students are given a virtual filesystem with files and directories. They can assign user IDs, groups, and POSIX permission bits. An avatar tries to read, write, or execute files, and the simulator highlights whether access is granted or denied. By experimenting with set-user-ID and sudo, they can see the difference between user permissions and kernel privileges, reinforcing the abstraction.

Simulation: Signal Playground
Students control a process that occasionally receives interrupts or exceptions (e.g., division by zero, Ctrl-C). They can attach or detach signal handlers and visually observe how the process responds—whether it terminates, ignores, or runs custom code. A timeline view shows delivery order and race conditions when signals arrive quickly.

Week 3–4: Processes and Threads

Simulation: Process Tree Explorer
A graphical “forest” shows processes spawning via `fork()` and replacing themselves with new executables using `exec()`. Students can run sample programs to watch children being created, then terminate them to see how the parent reaps orphans. Switching views reveals how process IDs and file descriptors are inherited, deepening understanding of the process model.

Week 5–6: Virtual Memory

Simulation: Virtual Memory Mapper
Students enter a virtual address and watch it be translated through page tables to a physical frame. They can trigger page faults, then choose between demand loading and copy-on-write to

see the effect. A memory view highlights pages in RAM vs. on disk, showing how multi-level page tables reduce memory usage.

Week 6–8: Caches

Simulation: Cache Hit/Miss Explorer
Students step through a small C loop, watching each memory access pass through a cache. The simulator animates whether the access is a hit or miss, highlights the tag store, and shows data being evicted under direct-mapped, set-associative, or fully associative policies. They can adjust cache size and associativity, then re-run the program to compare miss rates.

Week 10–12: Threads and Synchronization

Simulation: Thread Race Lab
Students run two or more threads that increment a shared counter. Without synchronization, the counter's final value is wrong due to race conditions. By adding mutexes, barriers, or semaphores through an interactive control panel, they can see correct results appear. The visualization includes a timeline showing when each thread acquires/release locks and how deadlock can occur.

Week 13–14: Networking and Secure Channels

Simulation: Network Layers Game
Students drag a "message" through stacked layers (application, transport, network, link). At each layer, the simulator wraps the message in headers (like an envelope). Errors or delays can be introduced (lost packets, incorrect addresses), and students must debug by checking headers. Extending this, they can toggle encryption or digital signatures to see how secure channels prevent tampering or replay.

Week 14–15: CPU Pipelining and Out-of-Order Execution

Simulation: Pipeline Builder
Students place instructions in a timeline and watch them flow through stages (fetch, decode, execute, memory, writeback). Hazards (data, structural, control) are visually flagged, and students can apply solutions like stalling, forwarding, or branch prediction. Later, they can turn on out-of-order execution and register renaming to see how instruction throughput increases while maintaining correctness.

Week 15: Side Channels

Simulation: Cache Side-Channel Demo
Students run two simulated programs: a "victim" that accesses secret data, and an "attacker" that measures cache access times. By toggling which memory blocks the victim touches, the attacker's timing chart reveals the secret indirectly. Students can then apply mitigations (constant-time operations, cache flushing) to see how defenses reduce leakage.

These are the example outputs for each weeks activity:

Week 1–2: Build System Visualizer

Example Input:
Project with files:

- `main.c` (depends on `util.h`)
- `util.c` (defines helper function)
- `math.c` (independent)

What happens:
The student clicks *build*. The tool shows `main.c → main.o`, `util.c → util.o`, `math.c → math.o`, then links them into `program.exe`. If the student edits `util.c` and rebuilds, only `util.o` is recompiled, saving time.

Week 2–3: Permissions Sandbox

Example Input:

- File `secret.txt` with permissions `rw-------` (owner-only).
- User: `bob` (not owner).

What happens:
When Bob tries to `cat secret.txt`, the simulator shows "Permission Denied" and highlights the mismatch. If the student toggles the permission to `rw-r--r--`, the access now succeeds.

Week 2–3: Signal Playground

Example Input:
Run program `count.c` that prints numbers forever.
Send `SIGINT` (Ctrl-C).

What happens:

- With no handler: the program terminates immediately.
- With a custom handler: the program prints `"Caught SIGINT!"` and keeps counting.
- With `SIG_IGN`: the Ctrl-C has no effect.

Week 3–4: Process Tree Explorer

Example Input:
Run code:

```
if (fork() == 0) {
   execlp("ls", "ls", NULL);
}
```

What happens:
The simulator shows one parent process (`bash`) spawning a child. The child replaces itself with `ls`, prints a directory listing, and then terminates. The parent reaps the exit status.

Week 5–6: Virtual Memory Mapper

Example Input:
Access virtual address `0x00403abc`.

What happens:
The simulator shows:

1. Split into page number + offset.
2. Page number looked up in the page table.
3. If not in memory, a page fault occurs → load from disk.
4. Physical frame address is constructed, and memory at that offset is read.

Week 6–8: Cache Hit/Miss Explorer

Example Input:
Loop in C:

```
for (int i=0; i<16; i++)
   sum += arr[i];
```

Cache: 4 blocks, direct-mapped, block size = 4 integers.

What happens:

- The first access to `arr[0]` is a miss (brings in block [0–3]).
- `arr[1]`, `arr[2]`, `arr[3]` are hits.
- `arr[4]` is a miss (brings in block [4–7]).
  The timeline shows alternating misses and hits as blocks are reused.

Week 10–12: Thread Race Lab

Example Input:
Two threads increment a counter 1000 times each, without locks.

What happens:
Expected: 2000.
Actual: ~1700–1900 depending on race conditions.
When the student enables a mutex around the increment, the result is always 2000.

Week 13–14: Network Layers Game

Example Input:
Message: `"Hi"` sent from Alice (IP 1.1.1.1) to Bob (IP 2.2.2.2).

What happens:

- Application layer: `"Hi"`.
- Transport layer: adds UDP header with source/dest ports.
- Network layer: adds IP header with source/dest addresses.
- Link layer: adds MAC addresses.
  If the student changes Bob's IP to `3.3.3.3`, the packet is dropped because no route is found.

Week 14–15: Pipeline Builder

Example Input:
Instructions:

```
1: LOAD R1, 0(R2)
2: ADD  R3, R1, R4
3: STORE R3, 0(R5)
```

What happens:
The timeline shows instruction 2 waiting because it depends on R1 from instruction 1 (data hazard). If the student enables *forwarding*, instruction 2 executes one cycle earlier, reducing the stall.

Week 15: Cache Side-Channel Demo

Example Input:
Victim accesses either array element `[0]` or `[64]` depending on a secret bit.
Attacker times access to `[0]`.

What happens:

- If the victim touched `[0]`, the attacker sees a fast access (hit).
- If the victim touched `[64]`, the attacker sees a slow access (miss).
  By repeating, the attacker infers the victim’s secret bit.