The Hitchhiker's Guide to Operating Systems

Yanyan Jiang, *Nanjing University*

https://www.usenix.org/conference/atc23/presentation/jiang-yanyan

This paper is included in the Proceedings of the
2023 USENIX Annual Technical Conference.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-35-9

Open access to the Proceedings of the
2023 USENIX Annual Technical Conference
is sponsored by

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

The Hitchhiker's Guide to Operating Systems

Yanyan Jiang
Nanjing University

Abstract

This paper presents a principled approach to operating system teaching that complements the existing practices. Our methodology takes state transition systems as first-class citizens in operating systems teaching and demonstrates how to effectively convey non-trivial research systems to junior OS learners within this framework. This paper also presents the design and implementation of a minimal operating system model with nine system calls covering process-based isolation, thread-based concurrency, and crash consistency, with a model checker and interactive state space explorer for exhaustively examining all possible system behaviors.

1 Introduction

> "Everything should be made as simple as possible, but no simpler." —*Albert Einstein*

The teaching foundation of operating system design and implementation has been well-established for decades. From Tanenbaum's "*Operating Systems: Design and Implementation* (1987)" [45] to Arpaci-Dusseau's "*Operating Systems: Three Easy Pieces* (2018)" [3], students approach operating systems by studying the layered design of abstractions over processors, memory, and storage systems.

In parallel, researchers have observed the emergence of fast, scalable, reliable, and secure systems over the past few decades. This progress has been driven by the development of innovative system technologies, such as hardware/software co-design [24, 43], cross-stack integration [20, 23], program analysis [11, 48], and formal methods [30, 31], among others.

This paper attempts to share these exciting ideas with junior operating system learners under a unified theme by "adding a layer of indirection." Our key insight is to view all components of a computer system–including hardware, applications, and the operating systems that connect them–as state transition systems. By analyzing these components as informal yet mathematically rigorous objects, we aim to bridge the gap between theoretical concepts and practical system implementations.

This model-driven approach is grounded in several innovative philosophies on operating systems education, which are outlined below:

Everything is a state machine (Section 2). The key idea of this paper is to consider state transition systems as the foremost concept in teaching operating systems. The state-machine abstraction is fundamental: the state of a modern multi-processor system is essentially determined by register/memory bit values, driven by the non-deterministic selection of a single CPU executing a single-step instruction[1]. The same abstraction is also applicable to any multi-threaded program.

Consequently, we argue that it is beneficial to *view the operating system as both a state machine and a manager of state machines*. An operating system essentially leverages application-invisible data structures (e.g., a page table) to multiplex CPUs across processes and threads. This approach provides a rigorous explanation of process management APIs: `fork`/`execve`/`exit` functions simply clone, reset, and destroy live state machines. This abstraction also encourages in-depth discussions about `fork` [4] and the initial process state after `execve`.

By adopting this state-machine-centric perspective, we can explain research systems with a clear and rigorous foundation. For instance, every debugger [12], trace [8], and profiler [9] essentially "observes" runtime state snapshots, facilitating discussions on interactive query debuggers [39], deterministic full-system replay [16], time-travel failure reproduction [11], snapshot-based fault tolerance [38], and state space exploration [7] in an introductory-level operating system course.

Emulate state machines with executable models (Section 3). Since state machine is a mathematically rigorous concept, we could always *emulate* the execution of real state machines. Although emulation has been widely adopted in operating sys-

[1] Under the assumption of race-freedom that instructions are serializable.

System Call	Description
fork()	Create current thread's heap and context clone
spawn(f,xs)	Spawn a heap-sharing thread executing $f(xs)$
sched()	Switch to a non-deterministic thread
choose(xs)	Return a non-deterministic choice among xs
write(xs)	Write strings xs to standard output
bread(k)	Return the value of block k
bwrite(k,v)	Write block k with value v to a buffer
sync()	Persist all outstanding block writes to storage
crash()	Simulate a non-deterministic system crash

Table 1: System calls in the operating system model.

tem teaching[2], this paper takes one step further by emulating a "fully functional" operating system model with processes, threads, a debug console, and block storage. The system calls are listed in Table 1. The executable model approach has the following advantages:

First, *executable model is a foundation for exploring operating system concepts*. Synchronization primitives like Peterson's algorithm [34], condition variable, and semaphore can be implemented over shared memory. The non-trivial state copy behavior of `fork()` [4] can be reproduced under this model. A file system checker can be carried out upon a simulated `crash()`.

Second, *executable model is a behavioral specification of real operating systems*; it is the golden standard on the application-observable behaviors. A model facilitates discussions on the abstractions–the concrete implementation of the `fork()` function may employ copy-on-write, but this should remain transparent to a process. Such a model also motivates the key idea behind formally verified systems like seL4 [25] and Hyperkernel [31].

Enumeration demystifies operating systems (Section 4). We design our emulator to handle all sources of non-determinism in a coherent way: every system call (not merely `choose`) returns a set of possible choices as callbacks. Consequently, we can exhaustively enumerate all possible system behaviors with little implementation effort.

Such a design finally leads to our MOSAIC (Modeled Operating System And Interactive Checker) *operating system model and checker*. MOSAIC adds lightweight formal methods [21,47] to operating systems teaching. MOSAIC is capable of checking fork-based process parallelism, thread-based shared memory concurrency, and crash consistency [36]. The model checker's output can be piped to an interactive state space explorer that can be embedded in a Jupyter notebook (Figure 3); thus, all non-trivial corner cases of the operating system model can be rigorously explained.

In summary, this paper makes the following contributions:

1. We propose a new "state-machine first" approach in the breakdown of operating system teaching: (1) model systems as state machines, (2) realize models by emulation, and (3) explore models by enumeration. This approach enabled us to introduce non-trivial research systems to junior operating system learners.

2. We design and implement MOSAIC, a minimal (500 lines of code, including comments) executable operating system model and checker, which strikes a balance between understandability and functionality. MOSAIC can rigorously explain non-trivial textbook cases concerning concurrency, virtualization, and persistence. MOSAIC is available via

   `https://github.com/jiangyy/mosaic`.

3. We incorporated these ideas in a first undergraduate operating system course (Section 5). This course became one of the most popular operating system courses in China and has attracted over 2,000,000 video views since its initial release in 2020.

2 State Machines: First-class Citizens of Operating Systems

Philosophy 1: Everything is a state machine.

This paper's key contribution is the "state-machine first" approach to operating systems. By regarding both user-level applications and kernels as state machines (Section 2.1), it became obvious that operating systems are state machine managers (Section 2.2). This section also discusses modern computer systems and tools under the state machine perspective (Section 2.3).

2.1 Introducing State Machines in the Operating System Class

Program as a state machine. Every program run essentially boils down to the execution of binary instructions, whose behavior is rigorously defined by a state machine in which states are register/memory values and transitions are the execution of one instruction at the program counter. We *implement* this idea on Linux (Figure 1) to provide a definition of system calls: system call is a state transition (e.g., via a trap instruction or any process-kernel communication mechanism [44]) for accessing the "exterior" of the state machine, e.g., writing data to the operating system or changing the state machine's memory address space (via `mmap` or `mprotect`) and existence (via `exit`). Without system calls, the program (state machine) is a "closed world" that can only perform arithmetic and logical operations over memory and register values.

[2]We loved the emulated process scheduler, virtual memory, and file systems in the "Three Easy Pieces" [3].

```
1  #include "sys/syscall.h"
2  mov $SYS_write, %rax  // write(
3  mov $1,         %rdi  //   fd=1,
4  mov $hello,     %rsi  //   buf=hello,
5  mov $16,        %rdx  //   count=16
6  syscall               // );
7  // "ret" here yields SIGSEGV
8  mov $SYS_exit,  %rax  // exit(
9  mov $1,         %rdi  //   status=1
10 syscall               // );
11 hello: ; .ascii "Hello, OS World\n"
```

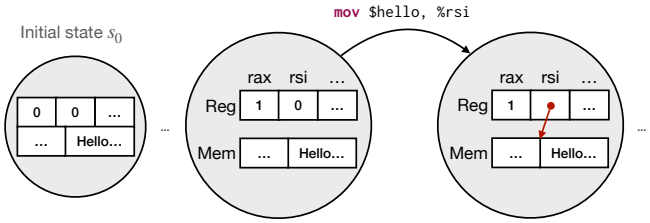

Figure 1: A minimal "Hello World" program and its corresponding state machine.

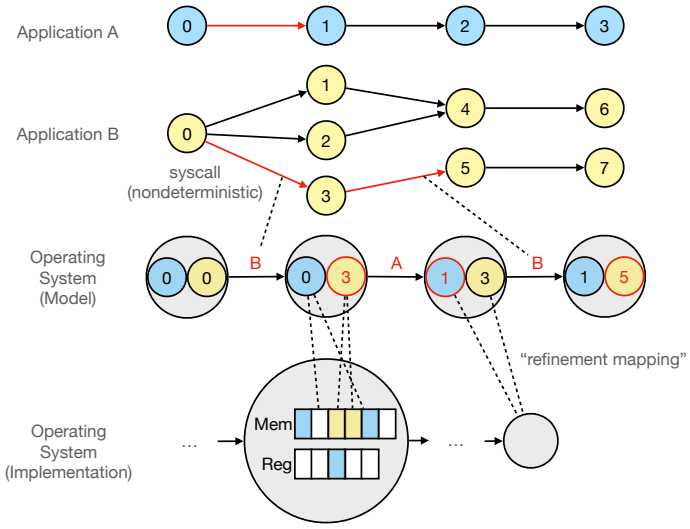

Figure 2: Operating system as a state machine manager. In this example, the operating system "executes" state transitions $0 \to 1$ and $0 \to 3 \to 5$ for applications A and B, respectively.

Bare-metal as a state machine. The bare-metal hardware shares a similar model with binaries: a CPU essentially operates as an infinite loop of instruction execution, which is also the case for a full-system emulator [5]. In contrast to user-level programs that can perform system calls, bare-metal kernels (including operating systems) access the "external world" via port or memory-mapped I/O and can be interrupted as if a trap instruction is non-deterministically injected.

Discussions. The advantage of introducing the state machine model early in an operating system course is that it fosters a tendency of *rigorous thinking*–state transition systems are well-defined mathematical objects. Specifically, we motivate the students to think of what is the mathematically precise definition of the process initial state. We explain that any process's initial state is well-defined by its binary executable and the Application Binary Interface. We also demonstrate how to inspect the initial state of the code in Figure 1 using `stepi` in GDB and memory mapping files in procfs. We further encourage students to consider more involved details of process states, such as the reasons behind the inability to perform a function return (using a `ret` instruction) and the necessity of wrapping C `main` functions with a `__libc_start_main`.

2.2 Operating System as a State Machine Manager

Computer system stack on state machines. Virtualization is the most fundamental mechanism of modern operating systems. Each application in an operating system can be regarded as a state machine whose initial memory layout and state transitions are specified by its binary executable. The operating system should give the application the illusion that the state transition system runs continuously following its specification, even though instruction execution could be non-deterministically interrupted at any time.

The state-machine approach provides a natural "implementation" of virtualization: by making state snapshots of all processes available and scheduling a process through "moving" a state machine to the CPU. The trap/interrupt handler plays such a role: it stores the state machine's registers in the operating system's private memory space, ensuring the system-wide invariant that all application states can be reconstructed. Subsequently, the operating system can continue processing interrupts, executing system calls, and resuming any process based on a predefined scheduling policy.

These arguments conclude our claim that "everything is a state machine" and gives us a new picture of understanding operating systems, as shown in Figure 2:

1. Application code is the developer's specification of a state machine.

2. Operating system code is the designer's specification of a state machine manager, a "superset" state machine container of all application state machines.

3. The operating system provides system calls as services and leverages application-invisible states (e.g., page table) to give processes the illusion of continuous state machine execution.

Process APIs on state machines. Following the idea that running applications are state machines, the need for process APIs became obvious: an operating system must provide

mechanisms for manipulating the set of live state machines. We found that the state machine language[3] precisely and concisely explains UNIX process APIs:

1. `fork()` makes a "full copy" of the currently running state machine. Registers and the address space should appear to be deeply copied. References to operating system objects (e.g., file descriptors and signal handlers) should also be copied, but with caution [4].

2. `posix_spawn(...)` creates a new state machine (always resets to the initial state of an application) with controllable state sharing with the parent.

3. `execve(path, argv, envp)` resets a running state machine to the initial state specified by the binary file `path`, with arguments `argv` and environment list `envp` placed in memory following the Application Binary Interface.

4. `exit(status)` removes the currently running state machine from the operating system, reclaims used resources, and notifies any waiting process with the exit `status`.

2.3 State Machines Meet Operating Systems

We discovered that the state-machine approach is not only beneficial for clarifying operating system concepts, but it can also serve as a fundamental basis for explaining non-trivial research systems to students:

Understanding system execution. Theoretically, executing a state transition system (be it an application or an operating system kernel) results in an execution trace composed of state snapshots connected by state transitions

$$tr = s_0 \to s_i \to \ldots \to s_{i+1} \to \ldots,$$

as if we single-instruction debug the program and save a core dump after each instruction execution. Such a trace contains *all* information needed for understanding this specific program execution.

However, such a massive trace (billions of instructions executed per second and megabytes of snapshots) is impractical and unnecessary to keep for any engineering practice. Debuggers provide the break/watchpoint mechanism to efficiently stop at interested program points (sometimes with hardware assistance like debug registers) and let the developer examine the program states interactively.

Understanding a program's execution usually only requires a tiny fraction of information in the full trace *tr*. The trade-off space of "what parts of *tr* to observe" leads to many important mechanisms incorporated in the engineering of modern operating systems, which are explained below.

[3]For brevity, we removed less critical mechanisms including signals, process groups, and access control in this discussion. However, all of them can be explained under the state machine perspective whenever needed.

Playing with snapshots. `fork` provides a verbatim copy of a program's state s_i with reasonably low cost. Holding such program state snapshots yields interesting applications. One is the Zygote process of Android [14], which copies initialized Java virtual machine state to avoid repetitive and time-consuming bootstrap-time class loading. Another example is that one can take periodical clean-state snapshots (e.g., in the idle state of an event loop) and fall back to a snapshot when an unexpected error occurs [38].

Time-travel debugging. Developers use a debugger to interactively examine *tr*, which can be enhanced by a query language [39]. Debuggers can also enable time-travel debugging by recording the differences between consecutive states, essentially creating an undo log. Time-travel debugging is already implemented in GDB [12]. Observing that non-deterministic transitions are only a tiny fraction of *tr*, one can also keep track of their locations and choices to enable a deterministic replay [16, 33].

Trace and profiler. One can insert probes exclusively at state transitions relevant to the application logic (e.g., function calls and returns) and gather diagnostic data (e.g., call stack traces). Trace utilities such as ftrace and Kprobe in Linux [8] are widely used for debugging production failures.

One can place probes only at application logic relevant state transitions (e.g., function calls and returns) to collect diagnostic information (e.g., call stack trace). Such trace tools like ftrace and Kprobe in Linux [8] are widely used in debugging production failures.

The overhead associated with tracing can be further reduced through sampling, which involves periodically activating probes within a specified time interval. Such profilers generate summaries of the sampled program states and are extremely useful in diagnosing performance issues.

Runtime checkers. Runtime checkers can also be considered as functions that accept *tr* as input and check it against specific bug patterns. A broad spectrum of checkers operate in this manner: AddressSanitizer [40] asserts the absence of out-of-bounds and use-after-free memory accesses. ThreadSanitizer [41] confirms that there are no conflicting shared memory accesses unordered by happens-before relations. Lockdep [29] checks whether all observed lock acquisition orderings do not form a cycle.

Symbolic execution and program verification. It is obvious that system calls can exhibit non-deterministic behavior. However, it is less emphasized that such non-determinism can be rigorously quantified; for instance, a read system call returns only a finite number of possibilities. Thus, we can enumerate all possible state transitions to capture all potential program behaviors; however, this approach is only feasible in a theoretical context. Even reading a 32-bit integer results in 2^{32} distinct states.

Using a compact representation of a vast number of states (e.g., using a symbolic value x to represent an "arbitrary"

value of variable `x`) and imposing constraints on symbolic values across branches results in a symbolic program verifier [7].

3 An Executable Operating System Model

> **Philosophy 2**: Emulate state machines with executable models.

As state machines are mathematically rigorous constructs, their usefulness is not limited to merely clarifying operating system concepts. It is also feasible to develop *executable* state machines that accurately emulate the behavior of processes and operating systems.

Specifically, we leverage modern programming language mechanisms like coroutines for lightweight in-process context switches to implement a lightweight executable operating system model with emulated threads, processes, and devices (Section 3.1). This section also discusses how instructors could use a model to simplify non-trivial textbook cases (Section 3.2) and use models as behavioral specifications of real systems (Section 3.3).

3.1 Emulating an Operating System

State machines (processes) and system calls. We implement our operating system model in Python, a popular programming language among students. A process is emulated by a generator (stackless coroutine) object where process memory is its local variables. System calls (Table 1) are emulated by `yield` in which the generator saves its local state (local variables and program counter) in a closure and transfers control to its caller[4]:

```
def main(msg):  # an emulated application process
  i = 0
  while (i := i + 1):
    yield 'SYS_write', msg, i  # write(msg, i)
    yield 'SYS_sched',  # sched()
```

Our operating system model, as a state machine manager, maintains a set of processes (continuable generators) and is an infinite loop of `yield` trap handler, just like any real operating system:

```
class OperatingSystem:
  def __init__(self, procs):  # OS initialization
    self._procs = procs
    self._current = procs[0]

  def run(self):  # the OS main loop
    while True:
      syscall, *args = self._current.__next__()
      match syscall:
        case 'SYS_write':  # write to debug console
          print(*args)
        case 'SYS_sched':  # switch to a random process
          self._current = random.choice(self._procs)

OperatingSystem([main('ping'), main('pong')]).run()
```

[4] In the MOSAIC implementation, the process code is stored in a standalone Python file. Applications invoke system calls in Table 1 as ordinary function calls like `x = sys_choose(['Head', 'Tail'])`, and MOSAIC rewrites the AST by replacing all system call nodes to `yield`.

Process APIs. Because deep-copying a generator object is not allowed in Python, we implement `fork()` by creating a new `OperatingSystem` object and replaying all executed system calls to obtain a deep copy of the process. This requires `OperatingSystem` to keep track of the non-deterministic choices of all previously executed system calls. Processes have increasing IDs starting from 1,000, and the child process ID is returned on `fork()`. There is no `exit()` because returned generators are never scheduled and are considered exited. There is also no `execve()` because its functionality largely overlaps with `spawn()` and `fork()`.

Threads and shared memory. The shared memory among threads is emulated by the global `heap` variable, whose value is updated before switching to a process/thread by

```
globals()['heap'] = self._current.heap,
```

and readers may notice that this `heap` models a "page table base register" which is changed on context switches. `spawn(f, *xs)` creates a new generator calling `f` with arguments `xs` and a shared `heap`. The replay-based `fork()` obtains a deep copy of the heap in the freshly allocated `OperatingSystem` object.

Devices. Writing to the debug console appends the message to a buffer. Reading from the debug console can be implemented by `choose()` from possible inputs. The emulated block device is a key-value mapping, which maps each block's ID (any string like `inode` or even emojis) to its contents (any serializable data structure including strings and lists). All block device writes are first appended to a queue to simulate real disks with a volatile buffer [36]. Write-back happens only when `sync()` is called.

3.2 Modeling Operating System Concepts

Such a surprisingly simple model can simplify textbook cases that require non-trivial interactions across system layers and are thus challenging to debug or even reproduce–we can selectively model the essential elements of the system to minimize the complexity:

A `fork()` in the road [4]. Fork is no longer simple, considering it conducts a full state copy of libraries and references (handles) to operating system objects. Below is such a non-trivial case related to the buffer mechanism in the standard C libraries:

```
for (int i = 0; i < 2; i++) {
  int pid = fork();
```

```
  printf("%d\n", pid);
}
```

```
(unix) $ ./a.out
1000
1001
0
0
1002
0
```

```
(unix) $ ./a.out | wc -l
8  # ???
```

Debugging the internal implementation of libc (even with a much simpler implementation like musl [1]) to understand this case requires substantial engineering efforts. Alternatively, we first model this case by removing all low-level details of process creation and focusing on the behavior of a fork-cloned buffer:

```
def main():
  heap.buf = ''
  for _ in range(2):
    pid = sys_fork()  # heap.buf is deeply copied
    sys_sched()  # non-deterministic context switch
    heap.buf += f'{pid}\n'  # or sys_write()
  sys_write(heap.buf)  # flush buffer at exit
```

The executable model always gives a process schedule to explain its outputs[5]. After fully understanding the model, students can examine the system call traces and debug the libc source code with less pain.

Understanding synchronization. Synchronization primitives (mutexes, condition variables, semaphores, etc.) are usually informally introduced in a textbook or an operating system course. Implementing them upon our operating system model gives them a rigorous semantics specification[6]. Below displays a model of the buggy producer-consumer implementation from Chapter 30 of "The Three Easy Pieces" [3], in which a consumer may erroneously wake up another consumer (instead of a producer), resulting in a deadlock:

```
def Tworker(name, delta):
  for _ in range(N):
    while heap.mutex == '🔒':  # mutex_lock()
      sys_sched()              # |- spin wait
    heap.mutex = '🔒'          # |

    while not (0 <= heap.count + delta <= BUFSIZE):
      sys_sched()
      heap.mutex = '🔓'          # cond_wait()
      heap.cond.append(name)     # |
      while name in heap.cond:   # |- spin wait
        sys_sched()              # |
      while heap.mutex == '🔒':  # |- reacquire lock
        sys_sched()              # |
      heap.mutex = '🔒'          # |

    if heap.cond:                # cond_signal()
      t = sys_choose(heap.cond)  # |
      heap.cond.remove(t)        # |- wake up anyone
    sys_sched()

    heap.count += delta  # produce or consume

    heap.mutex = '🔓'  # mutex_unlock()
    sys_sched()

def main():
  heap.mutex = '🔓'  # 🔒 or 🔓
  heap.count = 0  # filled buffer
  heap.cond = []  # condition variable's wait list
  sys_spawn(Tworker, 'Tp', 1)  # delta=1, producer
  sys_spawn(Tworker, 'Tc1', -1)  # delta=-1, consumer
  sys_spawn(Tworker, 'Tc2', -1)  # delta=-1, consumer
```

At first glance, this model seems to diverge from the textbook example, as all synchronization primitives are denoted by spin-wait constructs (Lines 3-4, 11-12, and 13-14). However, this is intentional: spin wait reflects the specification that the thread could not make any progress unless the synchronization condition is satisfied (e.g., a mutex is in the unlocked state or a condition variable has been signaled). Blocking wait is merely one possible implementation. Such a model also captures a detail often overlooked by students: a condition variable contains an implicit re-acquisition of its associated mutex (Lines 13–15) after being signaled. An executable model facilitates the development of rigorous concepts in operating systems.

The incorrect use of condition variable is also non-trivial: manifesting the bug requires at least three threads (a producer and two consumers) and $N \geq 2$. Such a fact can be easily verified by the model checker (Section 4.1). Running this model under a uniform-random scheduler, there is only approximately an 8% chance of triggering the deadlock in which all three threads T_p, T_{c_1}, and T_{c_2} are spinning on Line 11.

Finally, we found that emojis in the code can improve the readability of program states: "🔒" intuitively indicates that a thread holds this mutex. Other cases include using 🙋 in Peterson's algorithm [34] (instead of `flag[2]` and integer values `0` or `1`) to indicate a thread "raising hand" to enter the critical section and 💥👍👮 to denote success or failure.

File system consistency and journaling. The emulated block device enabled us to implement ideas in file systems without tedious low-level device details. Recall that the block device is conceptually a `dict`. Thus, we can assign blocks with intuitive names like `'bitmap1'` to indicate a bitmap block in the persistent storage. We can also use this `dict` as a file system by mapping file names (e.g., `'/tmp/a.txt'`) to their metadata and contents (e.g., `('symlink', '/etc/passwd')`) when the actual storage layout is not relevant. Below is a simplified model of xv6 [10] log commit:

```
def main():
  # 1. log the write to block #B
```

[5]The model checker (Section 4.1) can be used to exhaustively examine all process schedules and understand the possible outputs.

[6]Our model assumes that the execution of statements between consecutive `sched()` appears to be atomic and uninterruptible.

```
  head = sys_bread(0)  # blocks #1, #2, ... are the log
  free = max(log.values(), default=0) + 1  # allocate log
  sys_bwrite(free, f'contents for #{B}')
  sys_sync()

  # 2. write updated log head
  head = head | {B: free}
  sys_bwrite(0, head)
  sys_sync()

  # 3. install transactions
  for k, v in head.items():
    content = sys_bread(v)
    sys_bwrite(k, content)
  sys_sync()

  # 4. clear log head
  sys_bwrite(0, {})
  sys_sync()
```

With the model checking feature (Section 4.1), all possible crash behaviors and potential file system inconsistencies can be exhaustively explored.

3.3 Application: Specification of Systems

An operating system model can be useful beyond explaining textbook cases. A model also provides a *behavioral specification* for real operating systems, like a high-level reference implementation. For example, it could be proved that the mutex model in Section 3.2 has the following two properties:

1. Safety: as long as a thread holds a mutex, any other thread's lock acquisition never returns.

2. Liveness: a thread eventually acquires a mutex if threads with acquired locks eventually release them under a fair (random) scheduler.

Because everything is a state machine (and thus a well-defined mathematical object), it could be theoretically possible to *prove* that a real system's implementation is consistent with a model by constructing a refinement mapping[7]. This is exactly the idea behind formally verified systems like seL4 [25] (with a Haskell executable model) and Hyperkernel [31] (with a Python executable model), which all modeled operating systems as a state machine. Even though the technical details of the research work may be too involved for first-time operating system learners, state machines still facilitate grasping the fundamental concepts underlying them–one could always perform a "brute-force prove" by enumerating all reachable vertices on the state transition graph for finite systems.

Models are also useful as a behavioral reference for real system implementations. A more practical "refinement mapping" is to feed the same workload to both a model and a real system. Cross-checking the model and system traces validates the implementation's correctness. For example, executing the same

[7] One fundamental result of program verification is that refinement mappings between high-level and low-level specifications always exist [2].

```
1 Q ← {[]};                          // the queue of traces pending checking
2 S ← ∅;                             // the set of checked states
3 while ¬Q.empty() do
4     tr ← Q.pop() ;
5     ⟨s, choices⟩ ← replay(tr);
6     if s ∉ S then
7         S ← S ∪ {s};               // add the unexplored state to S
8         for c ∈ choices do
9             Q.push(tr :: c);       // extend tr with c and append to Q
```

Algorithm 1: The MOSAIC model checker

`fork()` sequence (assuming that all forks succeed) should yield identical process trees for both the model and a student's operating system kernel. Such an approach is also known as the lightweight formal method [22] and has been widely adopted in validating practical systems [6].

4 One Model Checker to Rule Them All

> **Philosophy 3**: Enumeration demystifies operating systems.

The executable model's behavior can be exhaustively explored by enumerating all possible non-deterministic choices. This section presents such a model checker (Section 4.1) and its application to operating system teaching (Section 4.2), followed by short quantitative experiments in Section 4.3.

4.1 MOSAIC Model Checker Design and Implementation

Instead of executing a system call immediately, all MOSAIC systems calls return a `dict` mapping possible choices (which can be regarded as labeled transitions in the state machine) to lambda callbacks for actually performing the system call, even if there is only one unique choice:

```
def sys_sched(self):
  return {  # all possible choices
    f't{i+1}': (lambda i=i: self._switch_to(i)) # callback
      for i, th in enumerate(self._threads)
        if is_runnable(th.context)
  }

def sys_fork(self, *args):
  return {  # only one choice
    'fork': (lambda: self._do_fork())
  }
```

Such a design yields a simple replay-based state space explorer as shown in Algorithm 1. The algorithm is a straightforward breadth-first search that memorizes traversed states in S. A trace is a chronological list of each system call's selected choice. Replaying a trace will always reach the next system

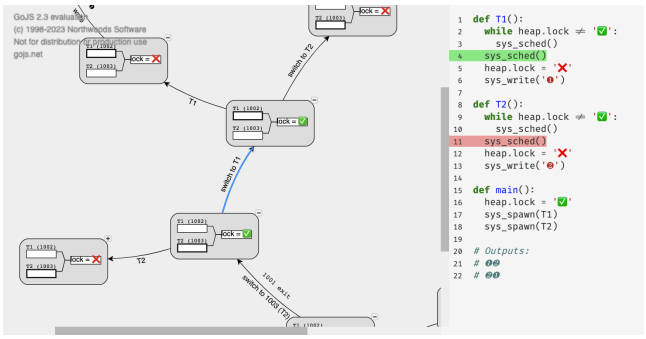

Figure 3: The interactive thread interleaving space explorer on MOSAIC's results of checking a spin lock implementation. Process and thread states are plotted as vertices. Thread program counters are highlighted on the source code like a debugger. Clicking a vertex expands its children.

call's non-deterministic choices (Line 5), or there is no choice ($choices = \varnothing$) when all processes and threads are terminated.

For finite-state models, the algorithm always terminates and produces a state transition graph whose vertices are traces in S and edges are labeled with c in Line 9. MOSAIC serializes the state transition graph as a JSON file. Both states (generator states, heaps, debug console output, and storage state) and transitions (labeled edges) are serialized. We encourage the students to follow the UNIX philosophy and pipe the text output to different backends:

1. Simply `grep stdout | sort | uniq -c` for a quick (and dirty, perhaps unsound) check for all possible debug console outputs.

2. Any JSON query or viewer like `jq` [15] to extract fields of interest (e.g., variable values or block device contents).

3. Our interactive state explorer (Figure 3) in which one can selectively expand nodes in the state transition graph. This interactive explorer is particularly handy for class demonstration.

4.2 Model Checking for Fun and Profits

The ability to exhaustively explore the state space makes a model checker suitable for rigorously explaining non-trivial cases in operating systems. A few such cases are shown below.

Processes and TOCTTOU attack. Both UNIX and our operating system model lack a mechanism (e.g., transactions [13, 37]) to enforce the atomicity across system calls and may be subject to time-of-check to time-of-use attacks. We demonstrate such a case of process-level race from [46]:

```
1 def main():
2   sys_bwrite('/etc/passwd', ('plain', 'secret...'))
3   sys_bwrite('file', ('plain', 'data...'))
4   pid = sys_fork()
5   sys_sched()
6   if pid == 0:  # attacker: symlink file -> /etc/passwd
7     sys_bwrite('file', ('symlink', '/etc/passwd'))
8   else:  # sendmail (root): write to plain file
9     filetype, contents = sys_bread('file')  # for check
10    if filetype == 'plain':
11      sys_sched()  # TOCTTOU interval
12      filetype, contents = sys_bread('file')  # for use
13      match filetype:
14        case 'symlink': filename = contents
15        case 'plain': filename = 'file'
16      sys_bwrite(filename, 'mail')
17      sys_write(f'{filename} written')
18    else:
19      sys_write('rejected')
```

MOSAIC reveals that "`/etc/passwd written`" is possible and gives such a process schedule. The exhaustive search can also reveal that the two `sys_sched` in Lines 6 and 10 are essential to produce such a result.

Hardness of shared-memory concurrency. Understanding thread interleaving can be difficult. Restoring the global ordering of shared memory accesses on thread-local read/write sequences is NP-Complete [17]. One interesting case is the possible outcomes of concurrent `tot++`, assuming that loads and stores are atomic (i.e., a sequentially consistent memory model) and the compiler does not merge multiple `tot++`:

```
def Tsum():
  for _ in range(N):
    tmp = heap.tot  # load(tot)
    sys_sched()
    heap.tot = tmp + 1  # store(tot)
    sys_sched()

def main():
  heap.tot = 0
  for _ in range(T):
    sys_spawn(Tsum)
```

MOSAIC reveals that `tot` can be 2 regardless of N and T (for $N, T \geq 2$) and gives such a thread schedule in which one thread "holds" a value of 2 in the last iteration of the loop and does not write it back until all other threads are terminated. We used the $N = 3, T = 2$ case as an exam problem, and approximately half of the students got wrong.

Persistence and crash consistency. Upon `crash()`, MOSAIC automatically explores all 2^n possible crash disks, assuming that any of the n buffered block I/O requests could be lost [36]. By modeling a file operation that involves multiple block updates (inode, bitmap, and data), an instructor can clearly and rigorously illustrate potential inconsistencies in a file system upon a system crash. Below is a textbook case in Chapter 42 of "The Three Easy Pieces" [3]:

```
def main():
  # intially, file has a single block #1
  sys_bwrite('file.inode', 'i [#1]')
  sys_bwrite('used', '#1')
  sys_bwrite('#1', '#1 (old)')
```

Subject	Parameters	# State	Memory	Time
fork-buf (7 LOC)	$n=1$ ($p=2$)	15	17.0 MB	<0.1s
	$n=2$ ($p=4$)	557	19.8 MB	3.3s (171 st/s)
	$n=3$ ($p=8$)			Timeout (>60s)
cond-var (34 LOC)	$n=1;t_p=1;t_c=1$	33	17.3 MB	<0.1s
	$n=1;t_p=1;t_c=2$	306	19.7 MB	0.1s (2 912 st/s)
	$n=2;t_p=1;t_c=2$	2 799	26.0 MB	0.8s (3 343 st/s)
	$n=2;t_p=2;t_c=1$	4 666	30.5 MB	1.4s (3 247 st/s)
xv6-log (27 LOC)	$n=2$	55	17.3 MB	<0.1s
	$n=4$	265	19.2 MB	<0.1s
	$n=8$	6 157	40.2 MB	1.3s (4 810 st/s)
	$n=10$	28 687	93.9 MB	20.7s (1 385 st/s)
tocttou (24 LOC)	$p=2$	33	17.4 MB	<0.1s
	$p=3$	97	17.8 MB	0.2s (413 st/s)
	$p=4$	367	19.4 MB	2.7s (135 st/s)
	$p=5$	1 402	23.5 MB	30.2s (46 st/s)
parallel-inc (11 LOC)	$n=1;t_s=2$	40	17.2 MB	<0.1s
	$n=2;t_s=2$	164	18.0 MB	<0.1s
	$n=2;t_s=3$	6 635	37.4 MB	1.4s (4 580 st/s)
	$n=3;t_s=3$	52 685	139.5 MB	14.1s (3 725 st/s)
fs-crash (25 LOC)	$n=2$	90	17.5 MB	<0.1s
	$n=4$	332	19.4 MB	<0.1s
	$n=8$	5 136	36.2 MB	2.6s (1 944 st/s)
	$n=10$			Timeout (>60s)

Table 2: Evaluation subjects and results. p, t, n denote the number of processes, threads, and loop iterations, respectively. All experiments were performed on an i7-6700 Linux PC with 4 GB RAM running Python 3.11. Each configuration is repeated for 10 times, and the average number is reported.

```
6   sys_sync()
7
8   # append a block #2 to the file
9   sys_bwrite('file.inode', 'i [#1 #2]')  # inode
10  sys_bwrite('used', '#1 #2')  # bitmap
11  sys_bwrite('#1', '#1 (new)')  # data block 1
12  sys_bwrite('#2', '#2 (new)')  # data block 2
13  sys_crash()  # system crash
14
15  # display file system state at crash recovery
16  inode = sys_bread('file.inode')
17  used = sys_bread('used')
18  sys_write(f'{inode:10}; used: {used:5} | ')
19  for i in [1, 2]:
20    if f'#{i}' in inode:
21      b = sys_bread(f'#{i}')
22      sys_write(f'{b} ')
```

MOSAIC's self-explanatory outputs verified that the one-page informal arguments in the textbook are indeed exhaustive and correctly covered all possible cases. MOSAIC can also check the journal implementation in Section 3.2 by adding `crash()` to the code and reveal that removing the `sync()` in Line 6 may result in file system inconsistency.

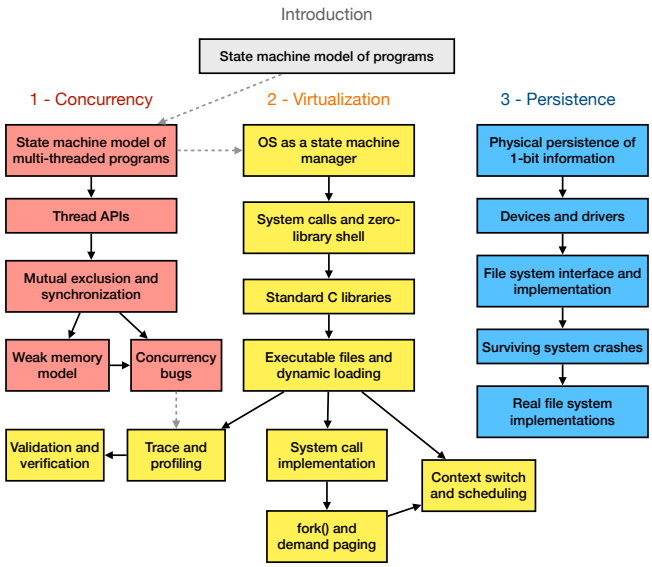

Figure 4: Major modules and their dependencies in our operating system course. The concept of state-machine is a good foundation for thread-based concurrency, and thus we introduce concurrency first in the course.

4.3 Experiments

We evaluate the performance of MOSAIC by checking the six representative models in Sections 3.2 and 4.2. Both experimental subjects and results are listed in Table 2. As expected, MOSAIC cannot address the state space explosion problem and has no comparable performance with a state-of-the-art software model checker with dedicated optimizations. Furthermore, programs that extensively fork is significantly slower (benchmarks fork-buf and tocttou) because our `fork()` is implemented by a full-system replay. Nevertheless, checking thousands of nodes per minute could be considered sufficiently useful for instructional purposes, and our design choice is to make a functional model checker minimal and elegant.

5 A New Operating System Course

We design a new operating system course from scratch based on "The Three Easy Pieces" [3] and our teaching philosophies: everything is a state machine, emulate state machines with executable models, and enumeration demystifies operating systems. The course syllabus is shown in Figure 4. This section presents the impacts of the state machine perspective (Section 5.1) and model checker (Section 5.2) in the course design, followed by discussions in Section 5.3.

5.1 State Machines and Operating Systems

In addition to introducing the key concepts in operating systems using state machines, the state machine perspective also

brings the following advantages in establishing a high-level understanding of important concepts regarding computer systems in a natural and coherent way.

Don't panic in hacking real systems! All students had a hard time in debugging real (even minimal) systems, including but not limited to operating system kernel, even if we provided skeletal code, tool chain, and state visualization scripts.

The state-machine perspective provides a natural reflex on how to deal with bugs or unexpected behavior in real systems: All bugs in computer systems are essentially some anomaly in the state-machine's execution trace. Given an unlimited amount of time, one just seeks the *first* abnormal state, and the root cause is right there. We teach students this (impractical) debugging principle and motivate students to consider clever tricks to make this procedure fast, robust, and easy.

For example, the essence of printf-debugging is to provide a high-level digest of the state-machine trace, which helps in narrowing down the scope of the initial anomalous state. One can also employ defensive programming by inserting assertions to the validity of states. These lessons are usually less taught in an operating system class but are essential for surviving hacking or implementing a large-scale system.

One classroom story is using a profiler (i.e., "frequent" state sampler) in diagnosing an unexpected 100% CPU usage on an idle workload in a production system in on a specific machine. The `perf` tool [9] attributes the hot spot to an `xhci`-related function, which leads us to a short-circuited USB port.

Concurrency meets state machines. The model checking community has long represented concurrent programs as state transition systems, and model checking is widely recognized as a computationally intensive technique that frequently encounters state explosion issues. Nevertheless, employing exhaustive enumeration is not the sole efficient approach to harness the capabilities of state machines.

The concept of data race, an important topic in operating system courses, refers to the simultaneous access of a shared memory location by two threads or processors (with at least one performing a write). Data races are considered harmful in systems programming.

When one checks a state machine trace against data races, it is essential to examine all types of state transitions that could lead to memory access [41]. However, two sources of memory access may be overlooked by students: (1) fetching an instruction from the program counter and (2) stack operations, including function and interrupt returns.

We let the students experience a subtle data race in an operating system kernel lab that requires students to migrate a process from one processor to another. The destination processor could not immediately schedule the process. Otherwise, there will be a data race on the kernel's interrupt stack.

Demystifying compilers. It is not obvious to students that C programs can also be represented by state transition systems. We use the example in Figure 5 (a non-recursive "Tower of Hanoi" implementation) to illustrate that the "runtime state" of C programs consists of static variables, heap memory, and a list of stack frames. State transitions are small-step expression evaluations at the top-most stack frame's program counter.

```c
void hanoi(int n, int from, int to, int via) {
  if (n == 1) {
    printf("%d -> %d\n", from, to);
  } else {
    hanoi(n - 1, from, via, to);
    hanoi(1, from, to, via);
    hanoi(n - 1, via, to, from);
  }
}
```

```c
typedef struct { int pc, n, from, to, via; } Frame;
#define call(...) ({*(++top) = (Frame) {0, __VA_ARGS__};})
#define ret()     ({top--;})
#define jmp(loc)  ({f->pc = (loc) - 1;})

void hanoi_nr(int n, int from, int to, int via) {
  Frame stk[64], *top = stk - 1, *f;
  call(n, from, to, via);
  while ((f = top) >= stk) {
    switch (f->pc) {
      case 0: if (f->n == 1) {
        printf("%d -> %d\n", f->from, f->to); jmp(4);
      } break;
      case 1: call(f->n - 1, f->from, f->via, f->to); break;
      case 2: call(1, f->from, f->to, f->via); break;
      case 3: call(f->n - 1, f->via, f->to, f->from); break;
      case 4: ret(); break;
      default: assert(0);
    }
    f->pc++;
  }
}
```

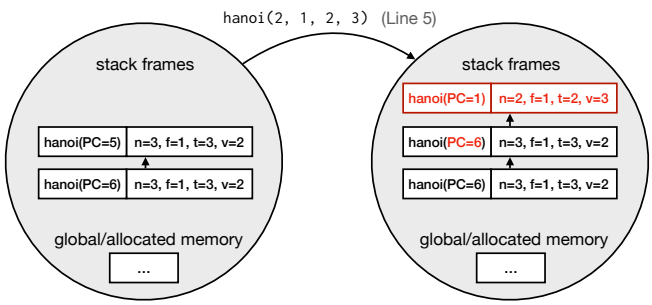

Figure 5: State machine perspective of C programs. `hanoi_nr` is also an "executable model" emulating recursions for rigorously understanding the semantics of C programs.

Compilers should always generate *equivalent* assembly (low-level state machine specification) from source code (high-level state machine specification). Therefore, a fundamental question is what kinds of translation are allowed for an optimized compiler. Notably, such deliberations are frequently neglected throughout the undergraduate curriculum, including in courses specifically addressing compilers.

With the conceptual model of state machines, the correctness of translation is essentially the equivalence between two state machines. This naturally leads to the definition of external observable equivalence between systems: given that system calls are the only way of influencing the remaining parts of the system, two programs are considered equivalent if they generate identical system call traces for the same inputs, and one program terminates if and only if the other program terminates. This principle serves as the core concept behind a verified compiler such as CompCert [27].

5.2 Modeling and Model Checking in Action

Models and emulation are everywhere. Models in an operating system course may not be limited to Python-implement system calls. We advocate using minimal but functionally "working" models, even they are implemented using a lower-level programming language.

One particular example is that we long had difficulties in explaining the ELF dynamic linker and loader to the students due to the unnecessarily excessive complexity of the ELF format. We identified that the problem stems from the fact that the ELF design is intended to be read exclusively by machines, rather than humans.

Therefore, we design a simplified binary format implemented using GNU C preprocessor and assembly. The binary file contains merely a magic number, a `NULL`-terminating symbol table whose entries are macros like `IMPORT(printf)` or `EXPORT(main)`, followed by assembly instructions. By reusing GCC and binary utilities, we implement the full toolchain of linker, loader, and an `objdump` equivalent in 200 lines of C code. Student and social media feedback indicate that such a model significantly flattens the learning curve of dynamic loading.

Formal method meets operating systems. We motivate the need for a model checker by making substantial (boring) efforts to draw a state transition graph to prove the safety and liveness of Peterson's mutex algorithm [34]. It is then obvious that a program like MOSAIC can replace human labor by emulation. We received positive feedback from students on their first contact with the model-checking approach, particularly the interactive visualizer (Figure 3), which is embedded in a Jupyter notebook for in-class demonstrations. The machine-generated state transition graph is also generally more reliable than the informal arguments in popular textbooks [42].

Another advantage of a model checker over existing teaching methods is the immediate feedback when answering "what if" questions related to changes in assumptions, implementations, and other factors. We encourage students to extensively experiment with the model, e.g., to see if the system breaks with added `sched()` or removed `sync()`.

The gap between models and real systems. We also teach students that models do not fully reflect the real world. Models are good at making *all* assumptions explicit, e.g., MOSAIC assumes the atomicity of statements between consecutive `sched()` calls and a sequentially consistent memory model.

The discrepancies between a model and an actual system are explained by careful examination of these assumptions. Peterson's algorithm is correct only under proper assumptions–specifically, a sequentially consistent memory model as if context switches only happen on instruction boundaries. For Peterson's algorithm, we provide an equivalent C implementation to illustrate how compiler and memory barriers may impact the program's behavior.

5.3 Student Acceptance and Discussions

Student Feedback. After publicizing the course lecture notes, demonstrations, and videos on the Internet, we received an excessive amount of positive feedback. Comments included statements like, "It is remarkable that such a comprehensive explanation of operating system principles can be provided in an undergraduate-level course." Students conveyed that they "gained valuable insights on overcoming the panic in hacking large-scale systems in this course."

There are also controversial arguments on the appropriateness of incorporating state machines as a key concept in operating system courses. However, we have also received feedback from the industry professionals supporting our approach by indicating that state machines are one of the most fundamental abstractions for controlling complexity in building production systems.

Since the first public release of the course in 2020, the video has received more than 2,000,000 plays on the Internet. Moreover, this course has been conferred the "Test-of-Time Teaching Award of the Department," as chosen by alumni who evaluated all courses in their curriculum.

Usefulness of models. Modeling is a versatile technique for establishing concepts and understanding. Modeling can also control the complexity by selectively hiding low-level implementation details. Another major advantage of executable models is making operating system concepts *rigorous*. Concepts (e.g., mutex, condition variable, and crash consistency) can be defined by "all possible behaviors on a model."

One may argue that any model behavior can be manifested by real workloads, and thus students should have first-hand experiences on real systems. We consider understanding the model (and thus the concepts) a critical step before students can hack real systems. Otherwise, the excessive and irrelevant implementation details can be a significant source of distraction.

Limitations. The "state-machine perspective" motivates the key insights and high-level designs of operating systems well. However, such over-simplification may yield students overlooking the challenges of implementing real systems. Therefore, we still consider the "hands-on approach," [26] in which students implement their own operating system kernel on

emulated bare metal, an indispensable part of an operating system course[8].

MOSAIC only models a small fraction of an operating system. More are missing, and one has to model them explicitly: file descriptors, signals, futexes, RAID, network stack, etc. Theoretically, it could be possible to model them in MOSAIC; however, we preferred simplicity in our model design and leaving these mechanisms to user-level applications like we did in Sections 3.2 and 4.2.

The implementation of MOSAIC also has limitations: `main` must be a Python generator (rather than a stackful coroutine). Thus, system calls are not allowed in functions being called by `main`. MOSAIC also assumes that the program being checked is deterministic. Non-determinism beyond system calls (e.g., random numbers) results in unsound model-checking results. Considering that MOSAIC is a pedagogical model checker and an instructor can easily bypass these limitations; thus, they are not a significant obstacle to adopting MOSAIC in practice.

6 Related Work

Emerging from the logic and programming language community, formal methods (mainly model checking and formal verification) has been widely adopted in the validation and verification of computer systems [6, 25, 27, 31, 47]. The key idea of formal methods is to treat specifications, models, and implementations as unambiguously-defined mathematical objects and *prove* properties by exhaustive search or axiomatic reasoning.

Despite a growing trend of formal method applications for computer systems, the teaching practice of "classical" operating systems remains classical on the layered abstractions of computer systems [3, 42] and the "hands-on" approach [26] in which students hack teaching operating system kernels [10, 19, 35] over emulators like QEMU [5] to fully understand all low-level implementation techniques.

There are attempts to incorporate model checking in teaching computer systems. Hamberg and Vaandrager [18] modeled textbook concurrency control algorithms using the Uppaal modeling language and checker. Michael et al. [28] target real Java programs on a message-passing model and check against all possibilities of message reorderings, drops, and duplications. Both concurrent programs and distributed systems are classical application scenarios of a model checker. To the best of our knowledge, we are the first to apply formal methods throughout an entire operating system course.

MOSAIC models a fully functional operating system by the unified treatment of non-determinism in system calls (Section 4.1). MOSAIC can check the interactions between processes, threads, and devices. Such a design resembles the EXPLODE system [47] for model checking real storage systems, in which all non-determinism and fault injection are implemented upon `choose()`.

As a pedagogical model checker, MOSAIC's primary use is to explain real operating system behaviors by mapping the model's execution traces (e.g., examples in Sections 3.2 and 4.2) to real systems. Such an approach belongs to the paradigm of lightweight formal methods [21, 22], which strongly emphasizes practicability rather than the full soundness of a proof. Lightweight formal methods have been proven effective against validating excessively complex real systems [6]. Like other pedagogical model checkers [28], we intentionally trade off the performance with understandability. Compared with fully verified systems [31], MOSAIC is functional but with magnitudes less code.

Emulation is also a widely-adopted approach in operating system teaching, which facilitates students establishing a correct and rigorous understanding of concepts. The exercises of "Three Easy Pieces" [3] are based on a substantial amount of independent emulators. MOSAIC as a unified model, on the other hand, can model (and check) the interplay between different levels of system mechanisms, e.g., how file system operations and process-level race result in a TOCTTOU attack in Section 4.2.

Finally, (replicated) state machines also play a fundamental role in distributed systems [32]. Formal methods became increasingly necessary in handling the counter-intuitive corner cases often overlooked by informal arguments. We believe that getting familiar early with such a paradigm on rigorous modeling and reasoning in a first operating system course can inspire the future generation of system researchers.

7 Conclusion

This paper presents a state-machine-first and model-based approach to teaching operating systems. By leveraging modeling and model checking, we can define operating system concepts rigorously, explore system behaviors exhaustively, and motivate non-trivial research systems intuitively under a unified framework in a first operating system course. We believe that this paper's teaching philosophies have the potential to lead a paradigm shift in the teaching of operating systems.

Acknowledgments

We would like to thank Haibo Chen, Yubin Xia, the anonymous reviewers, and our shepherd, David Cock, for their valuable and constructive feedback on this work. This work is supported in part by National Natural Science Foundation of China (Grant #61932021, #62025202, #62272218), Fundamental Research Funds for the Central Universities (#2022300287, #020214380102), State Key Laboratory for Novel Software Technology, and the Xiaomi Foundation.

[8] Students all had a hard time debugging a bare-metal kernel. Such experiences further motivate the need for debugging aids and dynamic analysis in Section 2.3.

References

[1] the musl libc. https://musl.libc.org/.

[2] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[3] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 1.00 edition, August 2018.

[4] Andrew Baumann, Jonathan Appavoo, Orran Krieger, and Timothy Roscoe. A fork() in the road. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS 19, pages 14–22, New York, NY, USA, 2019. Association for Computing Machinery.

[5] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *2005 USENIX Annual Technical Conference*, USENIX ATC 05, Anaheim, CA, April 2005. USENIX Association.

[6] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. Using lightweight formal methods to validate a key-value storage node in Amazon S3. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP 21, pages 836–850, New York, NY, USA, 2021. Association for Computing Machinery.

[7] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI 08, pages 209–224, USA, 2008. USENIX Association.

[8] The Kernel Development Community. Linux tracing technologies. https://www.kernel.org/doc/html/latest/trace/index.html.

[9] The Kernel Development Community. perf: Linux profiling with performance counters. https://perf.wiki.kernel.org.

[10] Russ Cox, Frans Kaashoek, and Robert Morris. xv6: a simple, Unix-like teaching operating system. https://pdos.csail.mit.edu/6.1810/2022/xv6/book-riscv-rev3.pdf.

[11] Weidong Cui, Xinyang Ge, Baris Kasikci, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. REPT: Reverse debugging of failures in deployed software. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI 2018, Carlsbad, CA, October 2018.

[12] The GDB Developers. GDB: The gnu project debugger. https://sourceware.org/gdb/.

[13] The Microsoft Windows Developers. The Windows Kernel Transaction Manager (KTM). https://learn.microsoft.com/en-us/windows/win32/ktm/kernel-transaction-manager-portal.

[14] Android Developer Documentation. Overview of memory management. https://developer.android.com/topic/performance/memory-overview.

[15] Stephen Dolan. jq: sed for JSON data. https://stedolan.github.io/jq/.

[16] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, OSDI 02, pages 211–224, USA, 2002. USENIX Association.

[17] Phillip B. Gibbons and Ephraim Korach. Testing shared memories. *SIAM Journal on Computing*, 26(4):1208–1244, 1997.

[18] Roelof Hamberg and Frits Vaandrager. Using model checkers in an introductory course on operating systems. *SIGOPS Oper. Syst. Rev.*, 42(6):101–111, October 2008.

[19] David A. Holland, Ada T. Lim, and Margo I. Seltzer. A new instructional operating system. In *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education*, SIGCSE 02, pages 111–115, New York, NY, USA, 2002. Association for Computing Machinery.

[20] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. GhOSt: Fast & flexible user-space delegation of Linux scheduling. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP 21, pages 588–604, New York, NY, USA, 2021. Association for Computing Machinery.

[21] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, revised edition, 2016.

[22] Daniel Jackson and Jeannette Wing. Lightweight formal methods. *Computer*, 29(4):20–22, April 1996.

[23] Kostis Kaffes, Jack Tigar Humphries, David Mazières, and Christos Kozyrakis. Syrup: User-defined scheduling across the stack. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP 21, pages 605–620, New York, NY, USA, 2021. Association for Computing Machinery.

[24] Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostić, Youngjin Kwon, Simon Peter, and Emmett Witchel. LineFS: Efficient SmartNIC offload of a distributed file system with pipeline parallelism. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP 21, pages 756–771, New York, NY, USA, 2021. Association for Computing Machinery.

[25] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. SeL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP 09, pages 207–220, New York, NY, USA, 2009. Association for Computing Machinery.

[26] Malcolm G. Lane. Teaching operating systems and machine architecture—more on the hands-on laboratory approach. In *Proceedings of the Twelfth SIGCSE Technical Symposium on Computer Science Education*, SIGCSE 81, pages 28–36, New York, NY, USA, 1981. Association for Computing Machinery.

[27] Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[28] Ellis Michael, Doug Woos, Thomas Anderson, Michael D. Ernst, and Zachary Tatlock. Teaching rigorous distributed systems with efficient model checking. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys 19, New York, NY, USA, 2019. Association for Computing Machinery.

[29] Ingo Molnar and Arjan van de Ven. Runtime locking correctness validator. https://www.kernel.org/doc/html/latest/locking/lockdep-design.html.

[30] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. Scaling symbolic evaluation for automated verification of systems code with Serval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP 19, pages 225–242, New York, NY, USA, 2019. Association for Computing Machinery.

[31] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. Hyperkernel: Push-button verification of an OS kernel. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP 17, pages 252–269, New York, NY, USA, 2017. Association for Computing Machinery.

[32] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC 14, pages 305–320, USA, 2014. USENIX Association.

[33] Robert O'Callahan and Kyle Huey. rr: Lightweight recording and deterministic debugging. https://rr-project.org/.

[34] Gary L. Peterson. Myths about the mutual exclusion problem. *Inf. Process. Lett.*, 12(3):115–116, 1981.

[35] Ben Pfaff, Anthony Romano, and Godmar Back. The Pintos instructional operating system kernel. In *Proceedings of the 40th ACM Technical Symposium on Computer Science Education*, SIGCSE 09, pages 453–457, New York, NY, USA, 2009. Association for Computing Machinery.

[36] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI 14, pages 433–448, USA, 2014. USENIX Association.

[37] Donald E. Porter, Owen S. Hofmann, Christopher J. Rossbach, Alexander Benn, and Emmett Witchel. Operating system transactions. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP 09, pages 161–176, New York, NY, USA, 2009. Association for Computing Machinery.

[38] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: Treating bugs as allergies–a safe method to survive software failures. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP 05, pages 235–248, New York, NY, USA, 2005. Association for Computing Machinery.

[39] Andrew Quinn, Jason Flinn, Michael Cafarella, and Baris Kasikci. Debugging the OmniTable way. In *16th USENIX Symposium on Operating Systems Design and Implementation*, OSDI 22, pages 357–373, Carlsbad, CA, July 2022. USENIX Association.

[40] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC 12, pages 28–37, USA, 2012. USENIX Association.

[41] Konstantin Serebryany and Timur Iskhodzhanov. Threadsanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation*

and Applications, WBIA 09, pages 62–71, New York, NY, USA, 2009. Association for Computing Machinery.

[42] Abraham Silberschatz, Greg Gagne, and Peter B. Galvin. *Operating System Concepts*. Wiley, 10th edition, 2018.

[43] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. GPUfs: Integrating a file system with GPUs. *ACM Transcation on Computer Systems*, 32(1), February 2014.

[44] Livio Soares and Michael Stumm. Flexsc: Flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI 10, pages 33–46, USA, 2010. USENIX Association.

[45] Andrew S. Tanenbaum. *Operating Systems: Design and Implementation*. Prentice-Hall, first edition, 1987.

[46] Jinpeng Wei and Calton Pu. TOCTTOU vulnerabilities in UNIX-style file systems: An anatomical study. In *Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies - Volume 4*, FAST 05, page 12, USA, 2005. USENIX Association.

[47] Junfeng Yang, Can Sar, and Dawson Engler. EXPLODE: A lightweight, general system for finding serious storage system errors. In *7th USENIX Symposium on Operating Systems Design and Implementation*, OSDI 06, Seattle, WA, November 2006. USENIX Association.

[48] Cristian Zamfir and George Candea. Execution synthesis: A technique for automated software debugging. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys 10, pages 321–334, New York, NY, USA, 2010. Association for Computing Machinery.