Building Computer Systems From Scratch Around *AbstractMachine*

Yanyan Jiang

<jyy@nju.edu.cn>

Zihao Yu

<yuzihao@ict.ac.cn>

SPAR Group

Inst. of Computer Software

Nanjing University

Inst. of Computing Technology

Chinese Academy of Sciences

July 6, 2018

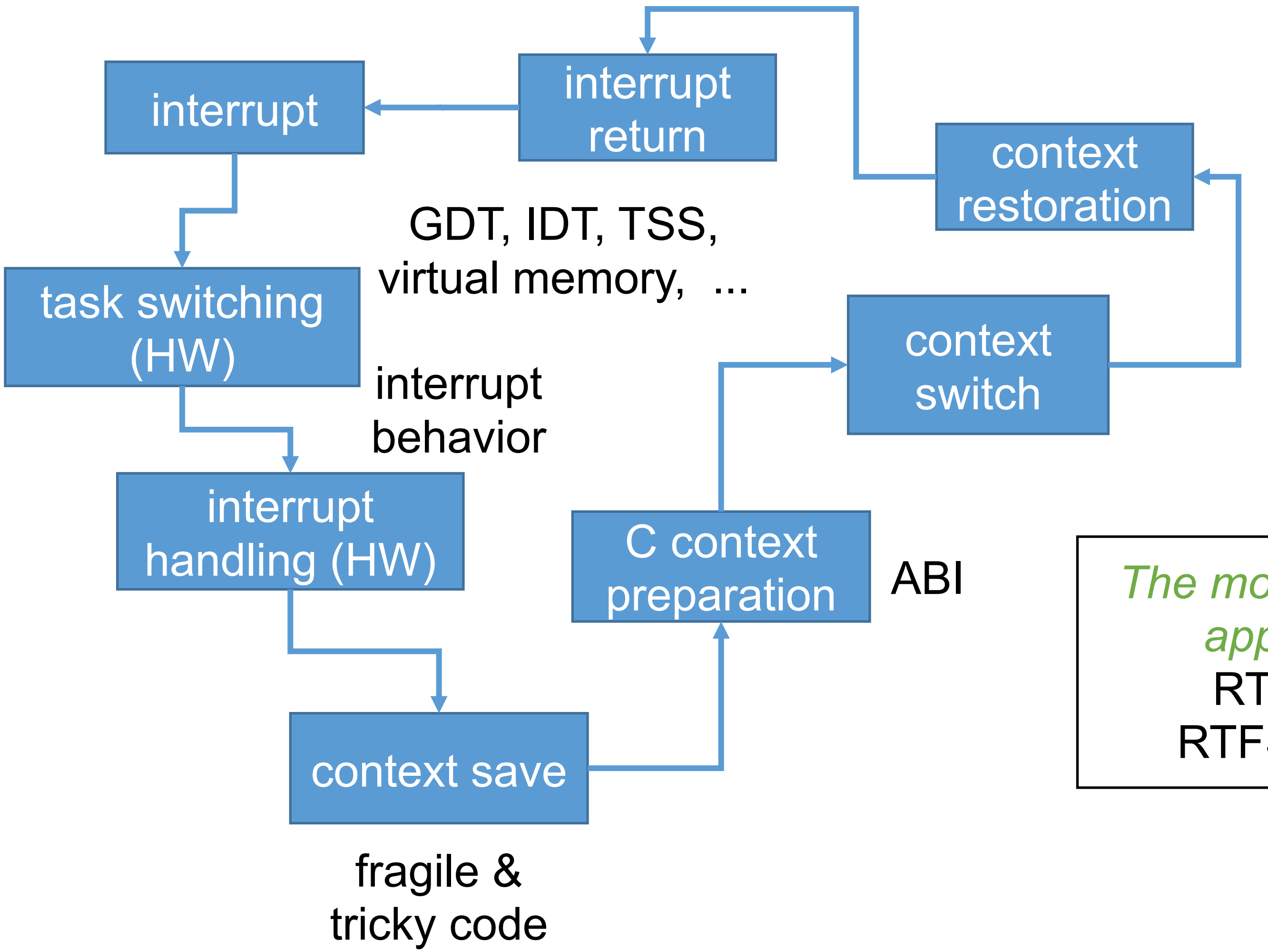
Background and Motivation

Students Build Systems at NJU

- The Project-N

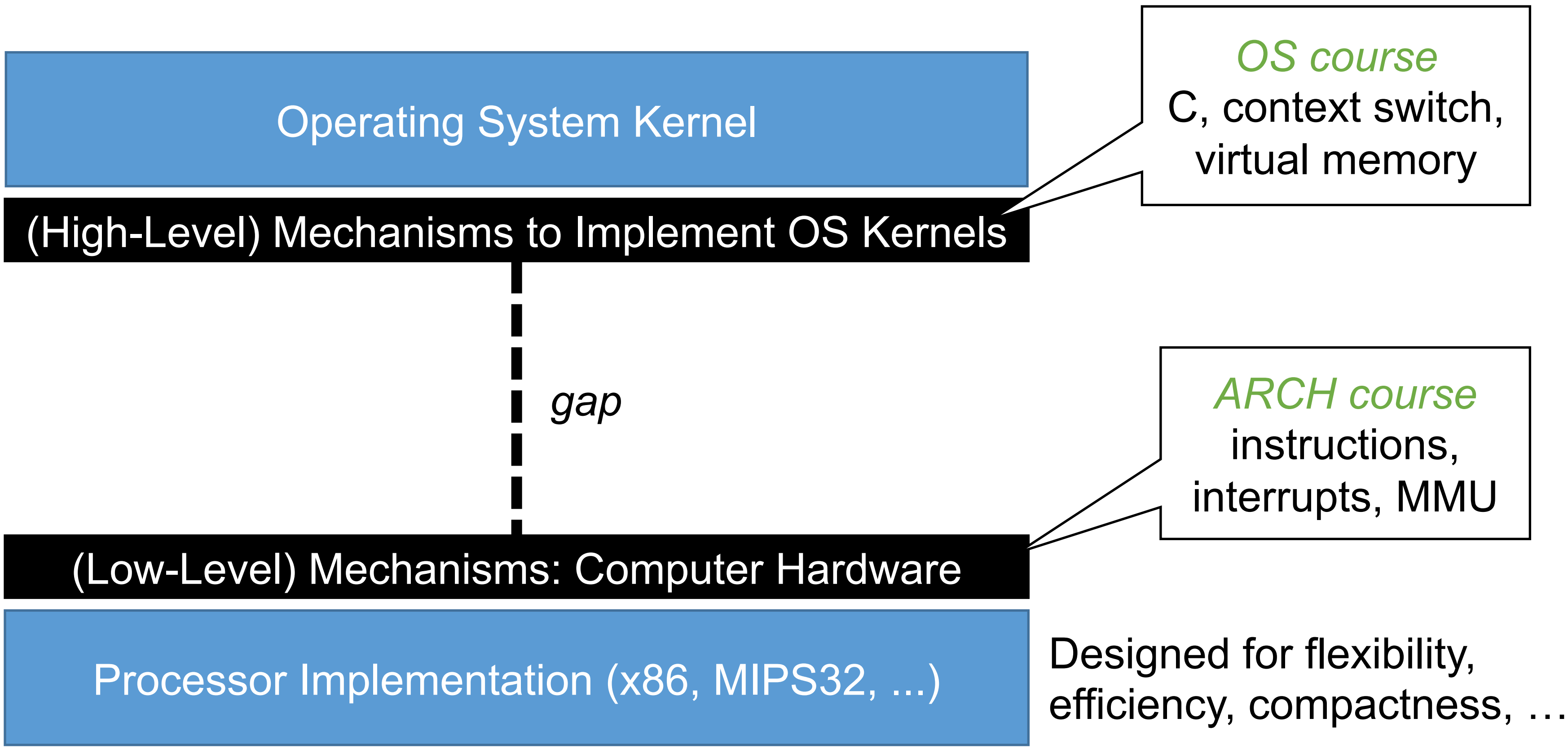
3rd semester	4th semester	5th semester	6th semester
NEMU (*NJU Emulator*) x86 full system emulator	Nanos (*NJU OS*) operating system kernel	NPC (*NJU Processor*) MIPS32 SoC	NCC (*NJU CCompiler*) MIPS32 Assembly
Linux native	x86 qemu	FPGA	SPIM emulator

But... Students Have Troubles Building (x86-Based) Systems!

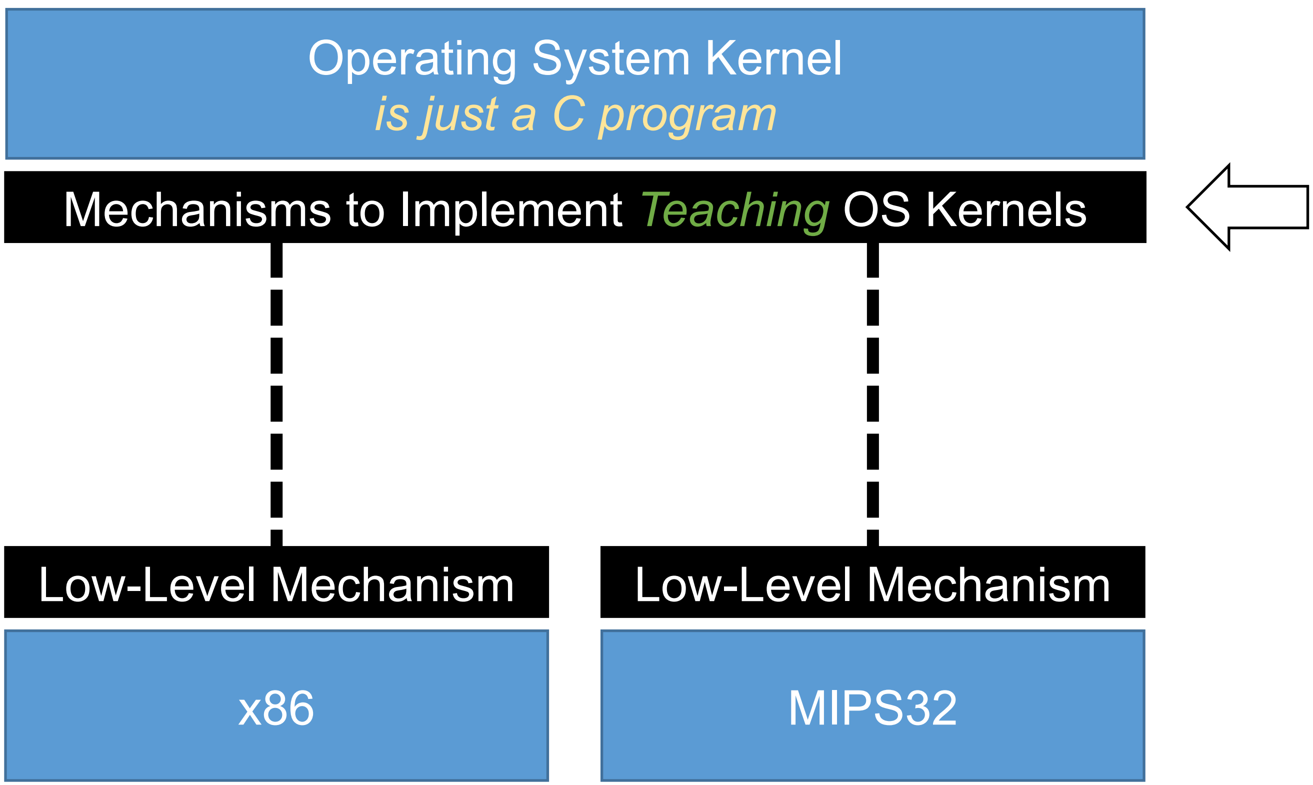


The most effective approach:
RTFM 😭
RTFSC 😭😭

Students' Trouble: There is Gap Between Low-Level and High-Level Mechanisms



Our Approach

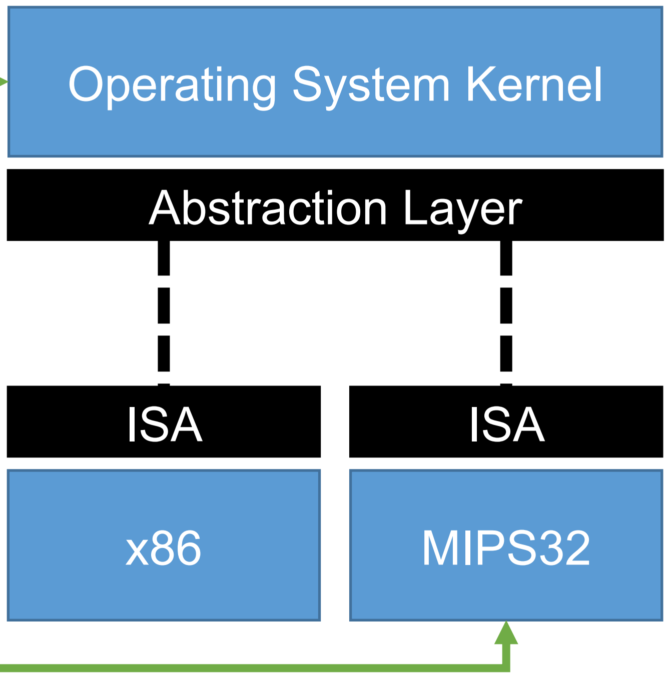


Build This Layer! 💡

Introducing the AbstractMachine

Design Goals

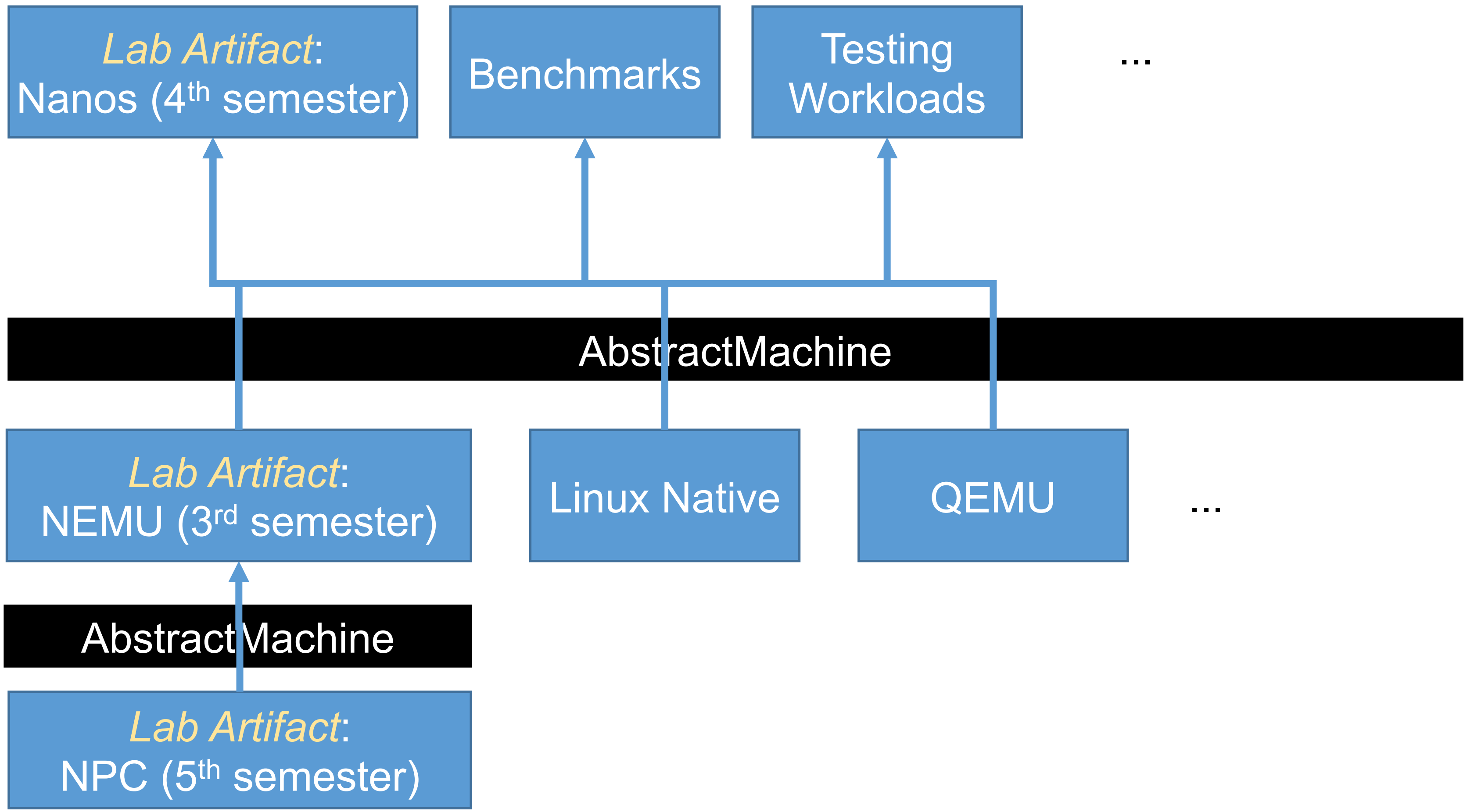
- An “abstract” architecture to
  - provide sufficient support for modern (teaching) system software
  - can be implemented on (perhaps overly) simplified processors
  - maybe at the cost of losing efficiency and/or flexibility
- Facilitate *portable* and *less painful* bare-metal software development



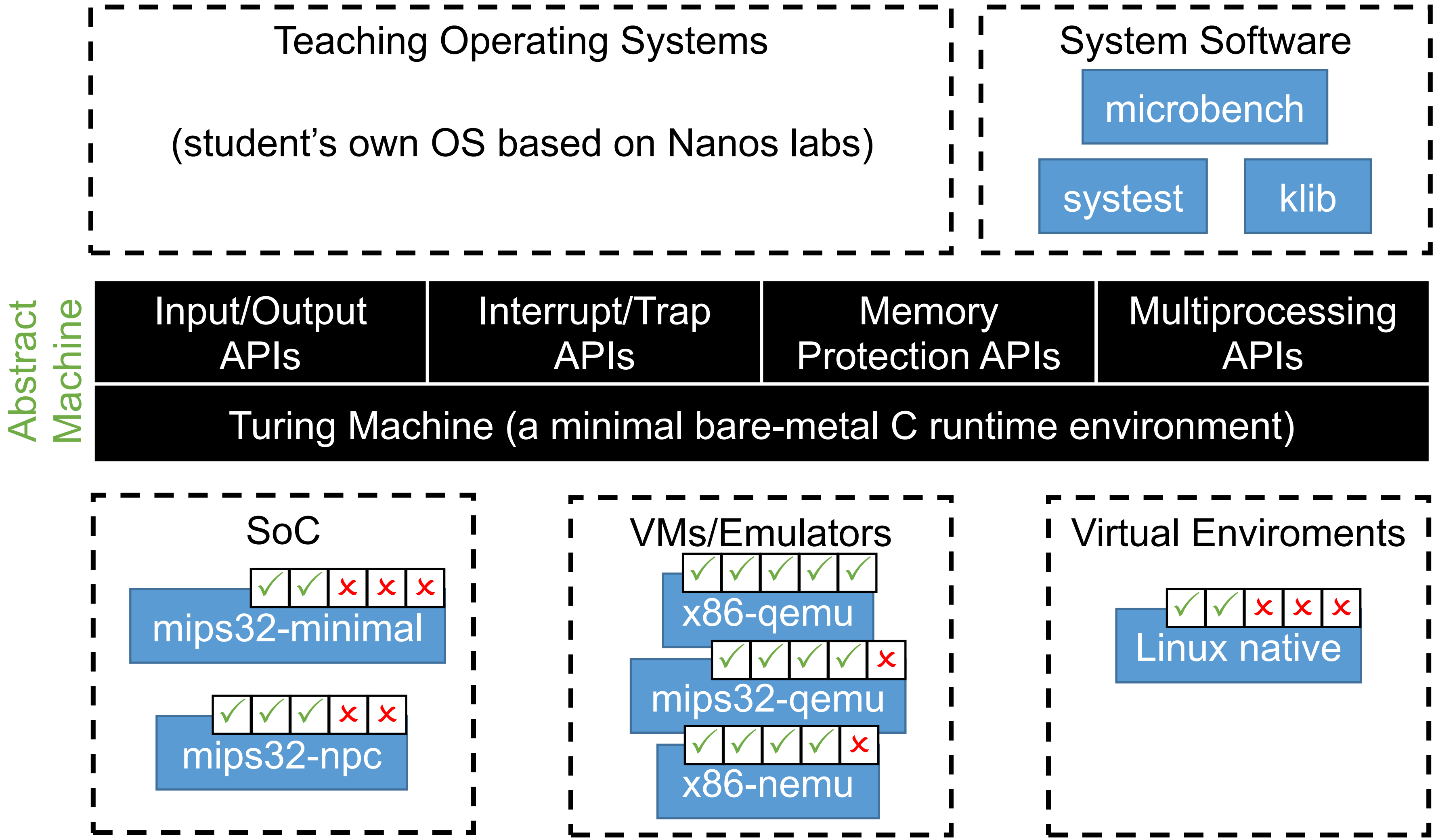
The AbstractMachine (Nexus-AM Project)

- To highlight the *high-level mechanisms* provided by computer hardware for implementing system software
  - a bare-metal C runtime (no libraries, statically linked)
    - bootstrap stack
    - a flat heap (usable physical memory)
    - a putc function for debugging
  - a series of optional C APIs
    - input and output
    - interrupt management and handling
    - virtual memory
    - multiprocessing

By Adding the Abstraction Layer...



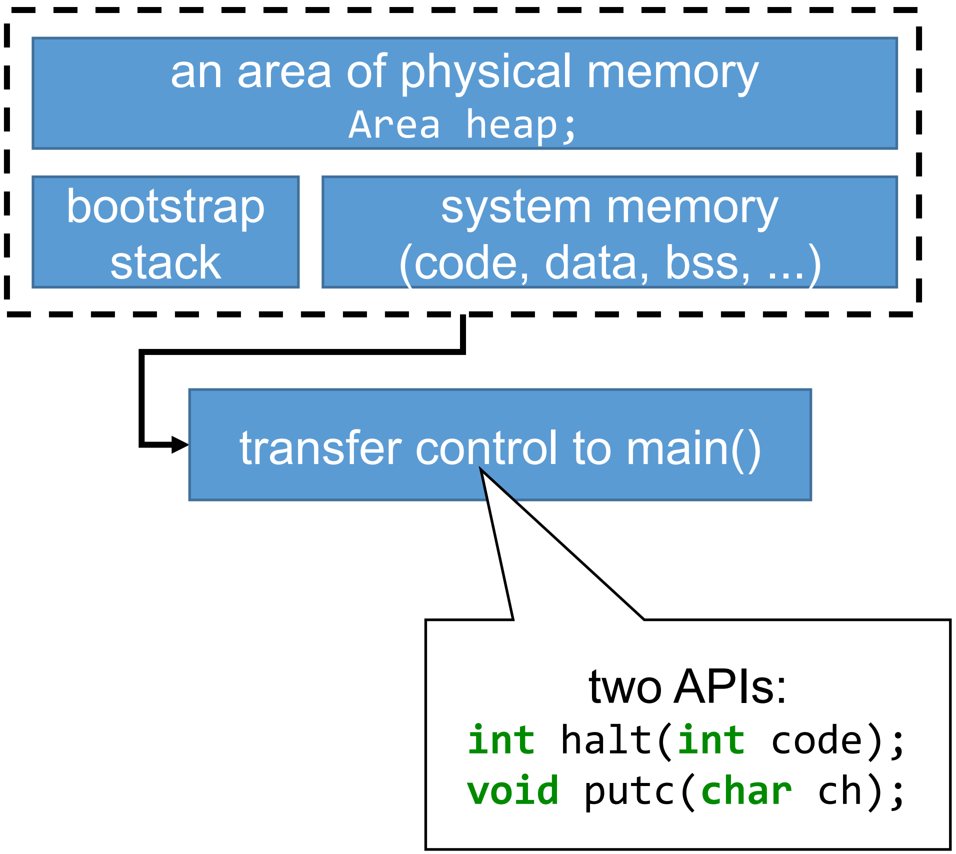
The AM Ecosystem: Overview

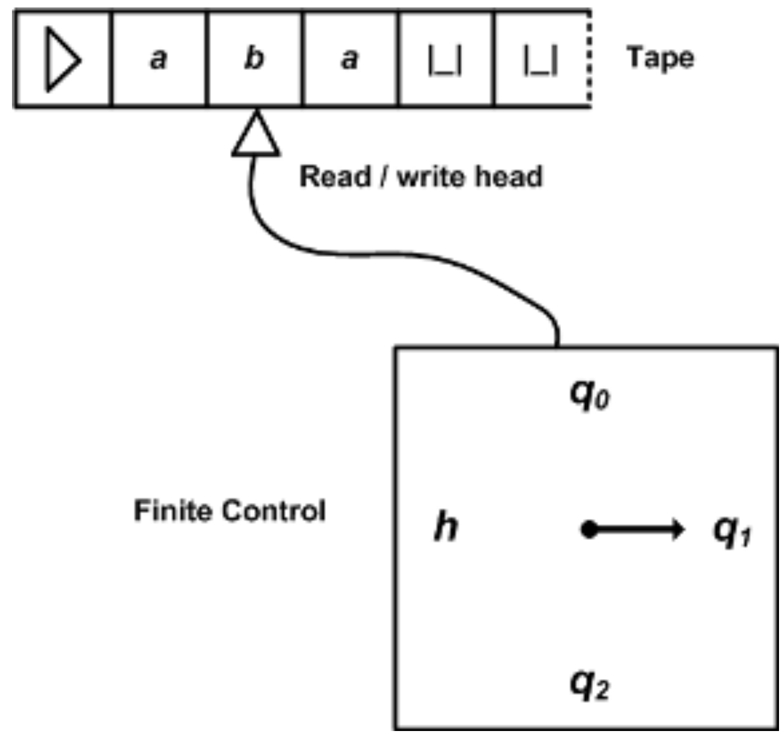


AbstractMachine Details

The Turing Machine (TRM)

- A minimal C bare-metal runtime environment

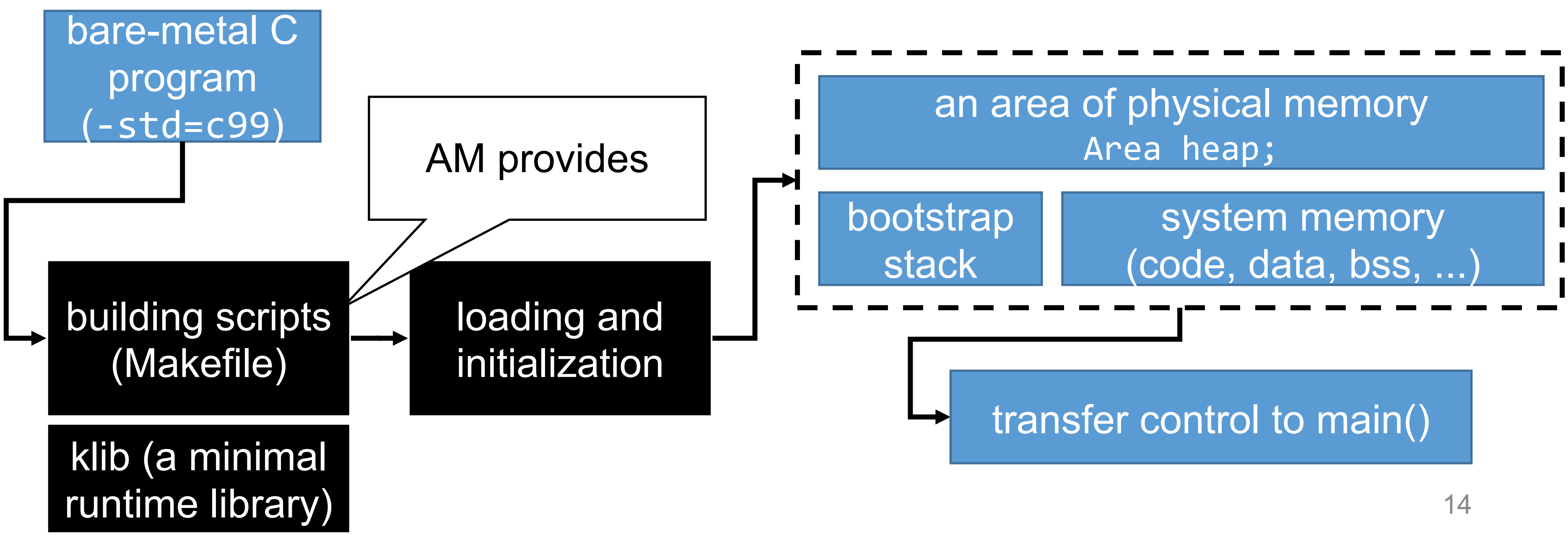




A Turing Machine
(image source: Wikipedia)

TRM Implementations

- TRM can be implemented on a *minimal*/*incomplete* system
  - qemu-system-i386 (full system emulation)
  - student's single-cycle processor (block RAM, no bus)
  - student's incomplete system emulator (basic instructions + a debug instruction)



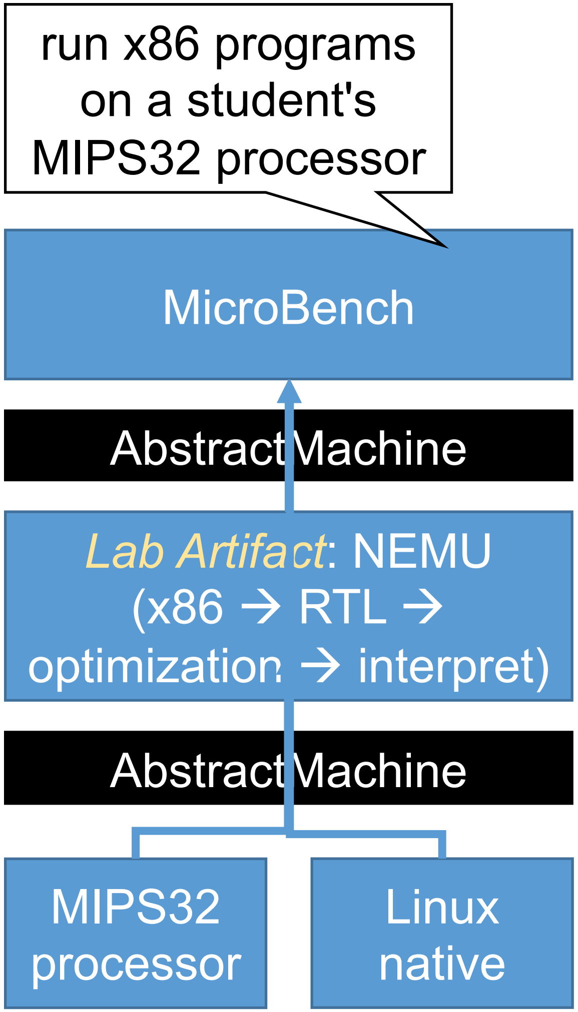
With TRM...

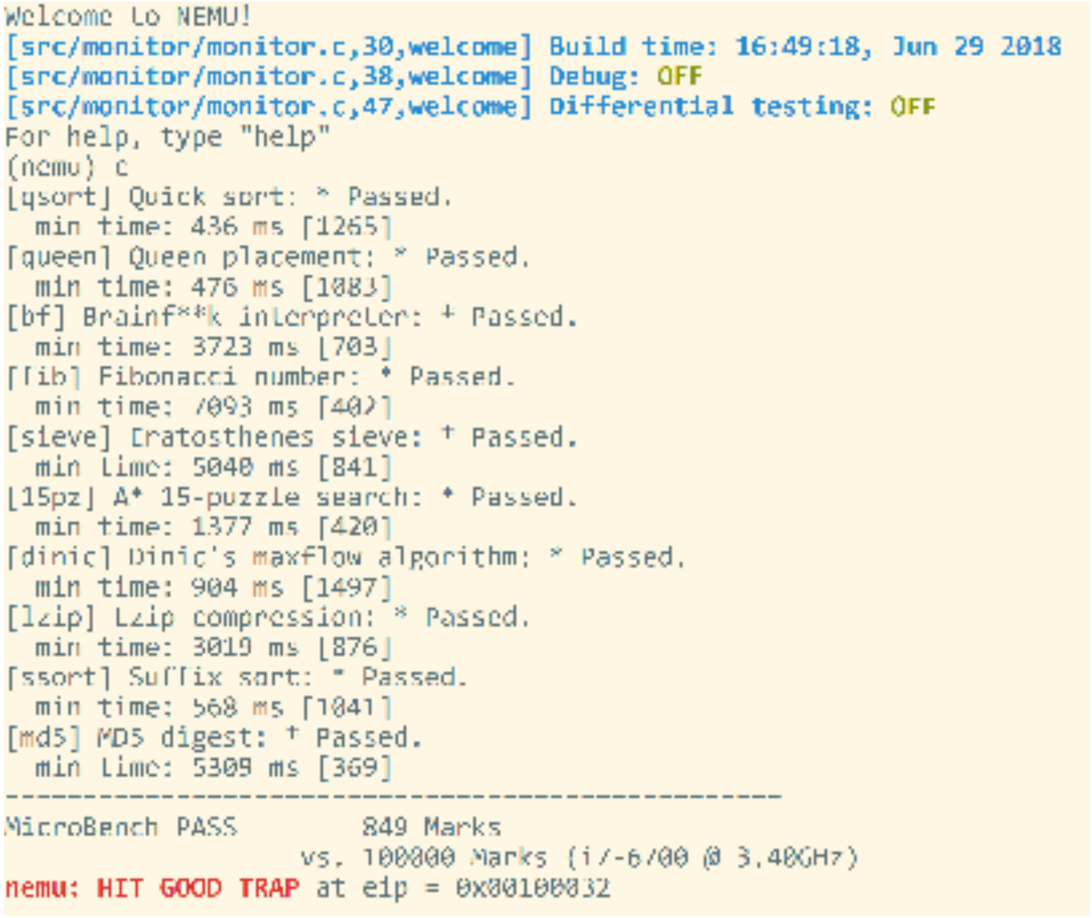
- We can run almost *any algorithm* for testing/benchmarking
  - coremark, dhyrstone
  - MicroBench (small footprint non-trivial programs)

makes labs more interesting (challenging)

```
#define BENCHMARK_LIST(def) \
  def(qsort, "qsort", QSORT_SM, QSORT_LG, "Quick sort") \
  def(queen, "queen", QUEEN_SM, QUEEN_LG, "Queen placement") \
  def(   bf,    "bf",    BF_SM,    BF_LG, "Brainf**k interpreter") \
  def(  fib,   "fib",   FIB_SM,   FIB_LG, "Fibonacci number") \
  def(sieve, "sieve", SIEVE_SM, SIEVE_LG, "Eratosthenes sieve") \
  def( 15pz,  "15pz",  PZ15_SM,  PZ15_LG, "A* 15-puzzle search") \
  def(dinic, "dinic", DINIC_SM, DINIC_LG, "Dinic's maxflow algorithm") \
  def( lzip,  "lzip",  LZIP_SM,  LZIP_LG, "Lzip compression") \
  def(ssort, "ssort", SSORT_SM, SSORT_LG, "Suffix sort") \
  def(  md5,   "md5",   MD5_SM,   MD5_LG, "MD5 digest")
```

Example: Student’s x86 Emulator



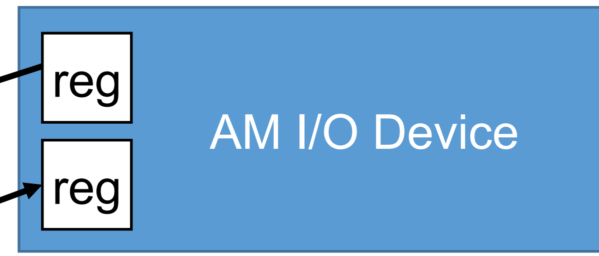


```
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 16:49:18, Jun 29 2018
[src/monitor/monitor.c,38,welcome] Debug: OFF
[src/monitor/monitor.c,47,welcome] Differential testing: OFF
For help, type "help"
(nemu) c
[qsort] Quick sort: * Passed.
  min time: 436 ms [1265]
[queen] Queen placement: * Passed.
  min time: 476 ms [1083]
[bf] Brainf**k interpreter: * Passed.
  min time: 3723 ms [703]
[fib] Fibonacci number: * Passed.
  min time: 7093 ms [402]
[sieve] Eratosthenes sieve: * Passed.
  min time: 5040 ms [841]
[15pz] A* 15-puzzle search: * Passed.
  min time: 1377 ms [420]
[dinic] Dinic's maxflow algorithm: * Passed.
  min time: 904 ms [1497]
[lzip] Lzip compression: * Passed.
  min time: 3019 ms [876]
[ssort] Suffix sort: * Passed.
  min time: 568 ms [1041]
[md5] MD5 digest: * Passed.
  min time: 5309 ms [369]
--------------------------------------------------
MicroBench PASS        849 Marks
                   vs. 100000 Marks (i7-6700 @ 3.40GHz)
nemu: HIT GOOD TRAP at eip = 0x00100032
```

I/O Extension (IOE) APIs

- An I/O device = read registers + write registers

```c
typedef struct Device {
  uint32_t id;
  const char *name;
  size_t (*read)(uintptr_t reg,
                 void *buf, size_t size);
  size_t (*write)(uintptr_t reg,
                  void *buf, size_t size);
} Device;
```



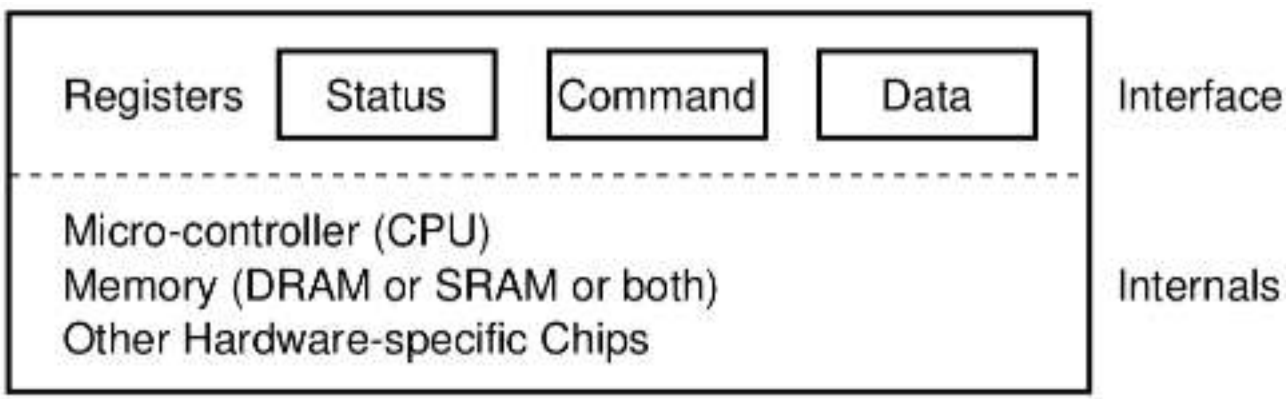


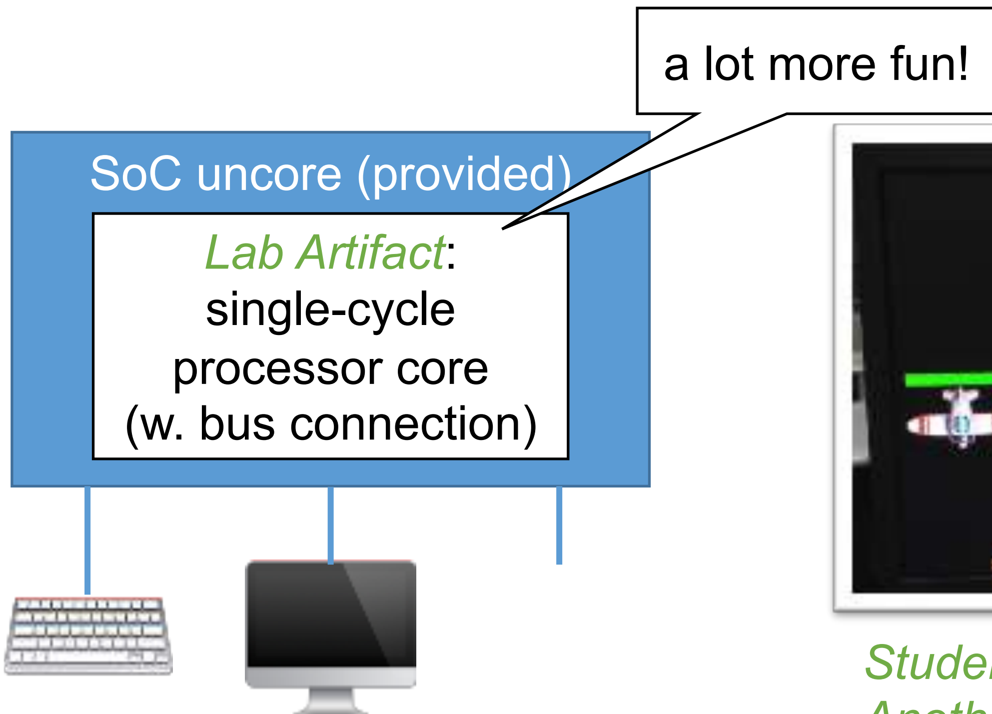
Image source: "Operating Systems: Three Easy Pieces"

I/O Device Examples

- ⌨️ AM keyboard
  - non-blocking read of key code
- 🖥️ AM frame buffer
  - draw a texture (an array of pixels) to a W*H rectangle
  - can be accelerated by DMA and/or a student's graphics card!
- PCI configuration space
  - memory-mapped I/O, x86 only

With IOE…

- We can run almost *any single-threaded kernel*, e.g., games



Student's OSLab0 (Game) on
Another student's MIPS32 SoC

Asynchronous Extension (ASYE) APIs

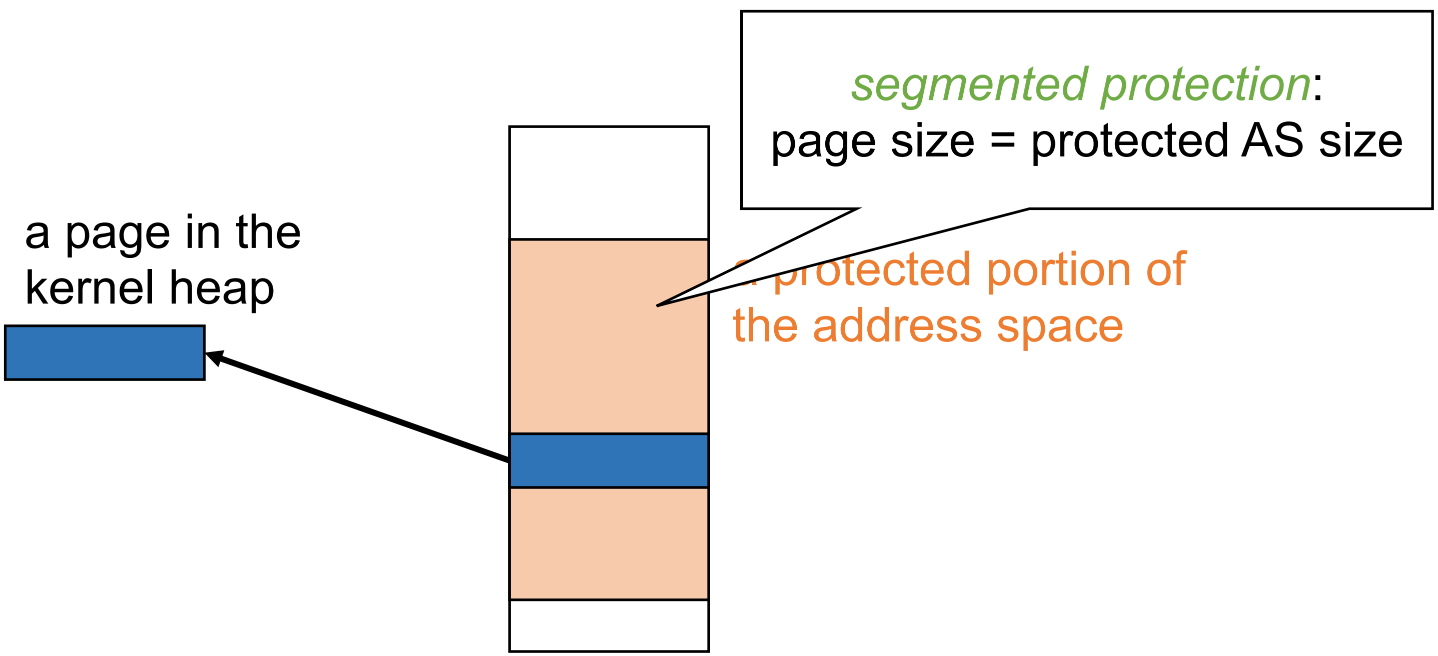
- Interrupt and processor context management
  - interrupt handling callback
  - interrupt enable/disable
  - self-trapping
  - context creation

Our experience: students have *no problem* understanding the semantics of these APIs

```
int asye_init(RegSet *(*handler)(Event ev, RegSet *regs));
int intr_read();
void intr_write(int enable);
void yield();
RegSet *make(Area kstack, void (*entry)(void *), void *arg);
```

Protection Extension (PTE) APIs

- (Very) high-level mechanisms for implementing virtual memory: *VM is a dict-like data structure*
  - creation/teardown of a protected address space
  - mapping/unmapping a page



With ASYE and PTE…

- We can run almost *anything*!
  - applications (and libc) on operating system

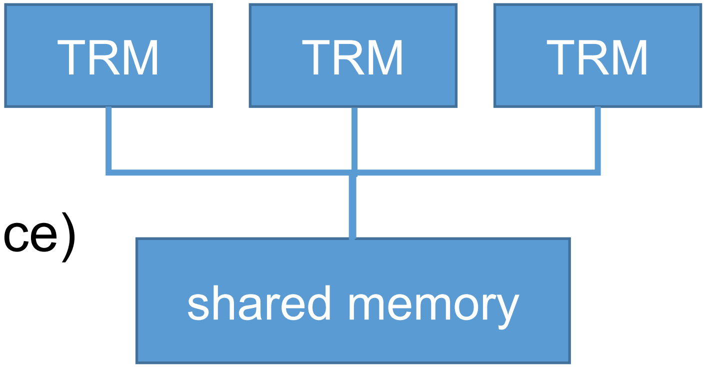
opens `/dev/fb` for graphics and `/dev/inputs` for timer/keyboard events

Playable PAL (仙剑奇侠传), newlibc,
Student’s Nanos and MIPS32 SoC (with AXI bus and DDR)

2nd place in 2017 Loongson Contest

Multiprocessing Extension (MPE) APIs

- Simple management of parallel bare-metal (shared-memory) C runtimes
  - boot multiprocessors
  - get multicore information
  - atomic operation (with cache coherence)




```c
int mpe_init(void (*entry)());
int ncpu();
int cpu();
intptr_t atomic_xchg(volatile intptr_t *addr,
                     intptr_t newval);
```

Case Study: AM in the Research Project

- In the development of Labeled RISC-V
  - lead by Zihao Yu at ICT-CAS

- Booting a full runtime environment is too costly in simulation
  - alternatively, implementing AM APIs on riscv64-rocket is *very easy*
  - we can run kernels (tests) without booting an OS

MemPerf on	Build Time	Run Time
FPGA	~30 minutes (synthesis)	~1 minute
Simulator (Linux)	~1 minute	~1 hour
Simulator (AM)	**~1 minute**	**~3 minutes**

AM on Labeled RISC-V: Case #2

- Testing low-level memory virtualization and cache coherence with stress multi-core parallel memory I/O workloads
  - only requires TRM, ASYE (for exceptions), and MPE (for multicore) ← *another configuration of AM*
  - 100X parallel (random) regression testing
    - 60,000 test runs in 10 hours, 1 minute/per test
    - with record & replay debugging

- Found and fixed an L1 DCache concurrency bug
  - the similar bug may exist in the upstream (confirming)

Discussions

AM: Some Obvious Benefits

- Less f**king manuals to read (and thus easier to teach)
  - many concepts no longer exist (GDT, LDT, TSS, ...)
  - a much more simplified interrupt/exception model
  - much easier to debug

- Motivates a student to maintain his/her own system
  - strive to make the entire system stack to work (compiler → application → operating system kernel → SoC on FPGA)
  - get hands dirty in debugging and full system

AM: Some Less Obvious Benefits

- AM threads the computer system labs
  - ARCH labs: TRM (a minimal processor) → IOE (buses and memory, SoC) → ASYE (interrupts and exceptions) → PTE (MMU)
  - OS labs: TRM + MPE (a base system to play with) → ASYE (kernel multithreading) → IOE (file system) → PTE (processes and system calls)
- Providing a layer of abstraction for systematic testing and/or verification
  - forced separation of machine-dependent (AM APIs) and machine-independent code

AM: Limitations

- Less fun hacking the systems
  - OS ninja students are really addicted to this 😜
  - this is the *trade-off* we take (let them implement AM APIs)

- Less low-level controls
  - cannot take full advantage of hardware supports (e.g., page directory COW, access/dirty bits, ...)
  - solution: make it *work* on AM, make it *better* on a particular architecture (e.g., x86-qemu)

Project-N Brewed Projects (Ongoing)

- *Navy* application framework
  - newlib (libc), libbmp, NWM (window manager), NTerm (terminal emulator), ... (with Zihao Yu and students)

- *Needle* plagiarism detector[1]
  - measures $d(P_1, P_2)$ between programs (with Prof. Chang Xu)

- *Nuts* random program generator
  - random kernels for fuzzing a student's processor with control- and data-flow diversity (with Xianfei Ou), now a research project to fuzz compilers (found previously unknown bugs in GCC)

[1] Yanyan Jiang and Chang Xu, "Needle: Detecting code plagiarism on student submissions", in *Proceedings of the ACM Turing Award China Conference* (SIGCSE China), 2018.

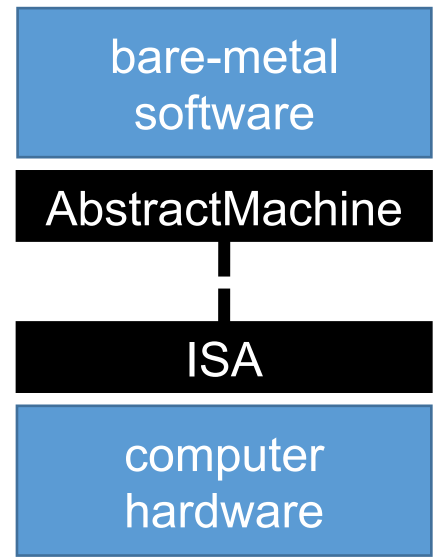
... And a Lot of Future Work!

- Enhancing existing infrastructures
  - testing/debugging/grading/... tools
  - libraries, applications (busybox), ...

- Porting more interesting stuffs into the AM ecosystem
  - xv6 (then it runs on MIPS!), ...

The AbstractMachine: Conclusion

- An abstraction layer of high-level instruction-set architecture mechanisms



- A bare-metal C runtime (and APIs)
- A trade-off for teaching computer systems
- The glue layer in Project-N system labs
  - currently adopted in ICS, OS, and ARCH labs at NJU

Yanyan Jiang <jyy@nju.edu.cn>
Zihao Yu <yuzihao@ict.ac.cn>