6.S081: Using Virtual Memory

Adam Belay

abelay@mit.edu

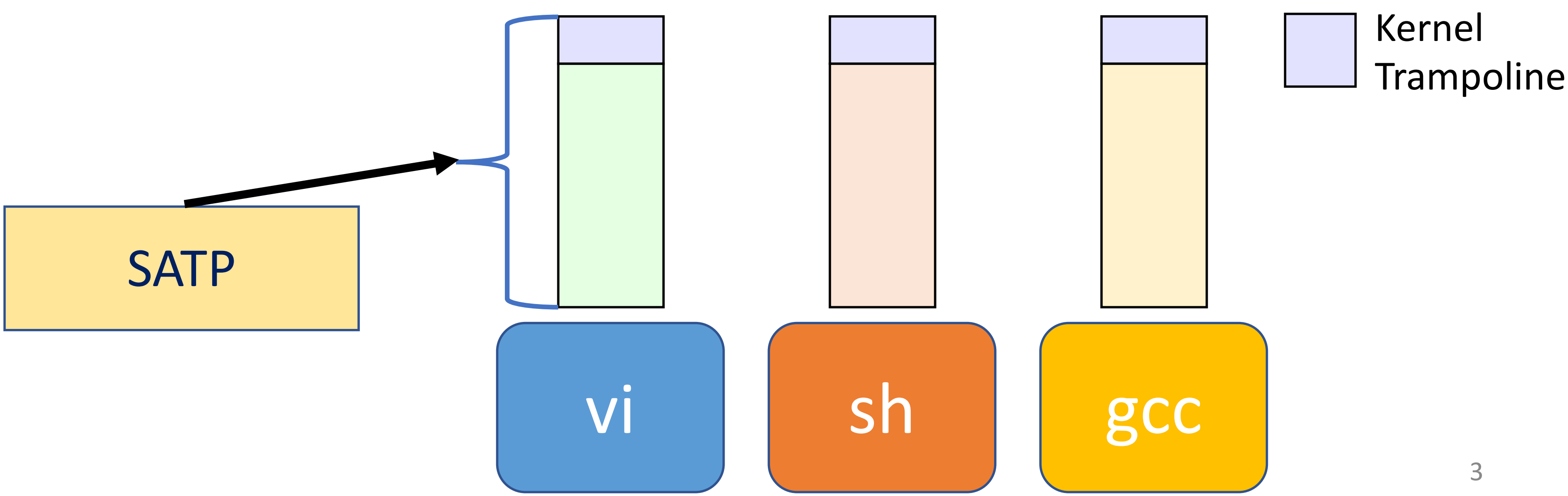
Outline

Cool things you can do with virtual memory:

- Virtual memory recap
- Lazy page allocation
- Better performance/efficiency
  - E.g. One zero-filled page
  - E.g. Copy-on-write w/ fork()
- New features
  - E.g. Memory-mapped files

Recap: Virtual memory

- Primary goal: Isolation – each process has its own address space
- But… virtual memory provides a level of indirection that allows the kernel to do cool stuff



Page table entries (PTE)

63–54	53–28	27–19	18–10	9–8	7	6	5	4	3	2	1	0
Reserved	PPN[2]	PPN[1]	PPN[0]	RSW	D	A	G	U	X	W	R	V
10	26	9	9	2	1	1	1	1	1	1	1	1

Figure 4.18: Sv39 page table entry.

Some important bits:

- **Physical page number (PPN)**: Identifies 44-bit physical page location; MMU replaces virtual bits with these physical bits
- **U**: If set, userspace can access this virtual address
- **W**: writeable, **R**: readable, **X**: executable
- **V**: If set, an entry for this virtual address exists
- **RSW**: Ignored by MMU

RISC-V page faults

- RISC-V supports 16 exceptions
  - Three related to paging
- Exceptions are controlled transfers into the kernel
  - Seen in previous and future lectures
- Information we might need to handle a page fault:
  1. The VA that caused the fault
  2. The type of violation that caused the fault
  3. The instruction where the fault occurred

SCAUSE register

Intr	Exception Code	Description
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	*Reserved*
0	5	Load access fault
0	6	AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call
0	9-11	*Reserved*
0	12	**Instruction page fault**
0	13	**Load page fault**
0	14	*Reserved*
0	15	**Store/AMO page fault**
0	>16	*Reserved*

STVAL register

- Contains exception-specific information
- Some exceptions don’t use it (set to zero)
- Page faults set it to the faulting address!
- Use r_stval() in xv6 to access

Gathering info to handle a pgfault

1. The VA that caused the fault?
   - STVAL, or r_stval() in xv6
2. The type of violation that caused the fault?
   - Encoded in SCAUSE, or r_scause() in xv6
   - **12**: page fault caused by an **instruction** fetch
   - **13**: page fault caused by a **read**
   - **15**: page fault cause by a **write**
3. The IP and privilege mode where fault occurred?
   - **User IP**: tf->epc
   - **U/K**: SSTATUS, or r_sstatus() & SSTATUS_SPP in xv6

xv6 user memory layout

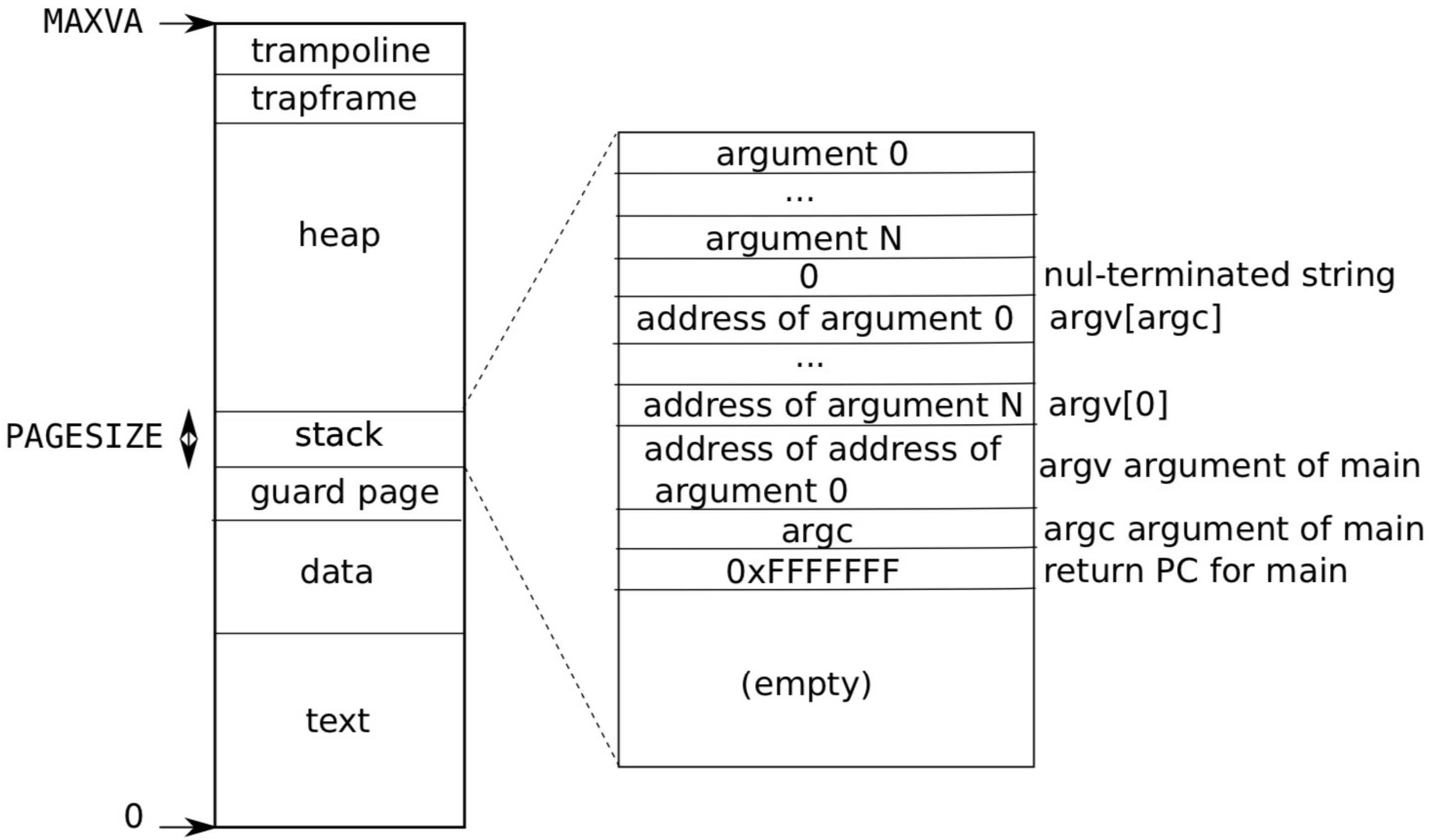
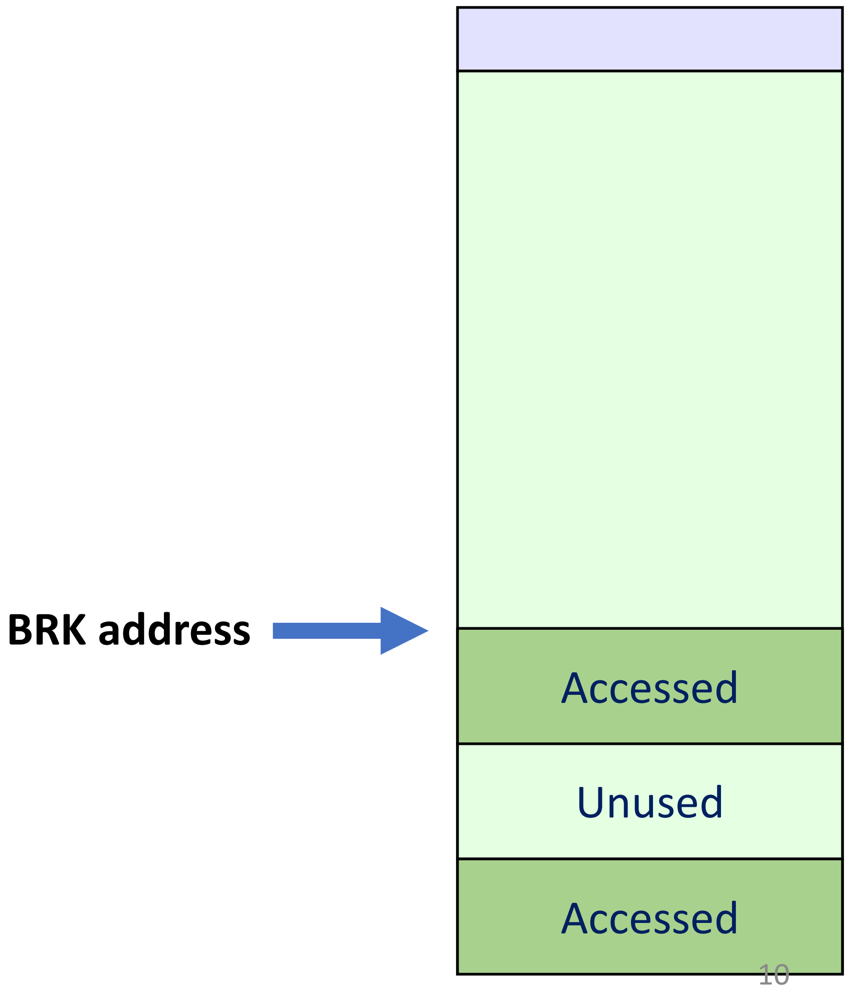


Figure 3.4: Memory layout of a user process with its initial stack.

Idea: On-demand page allocation

- Problem: sbrk() is old-fashioned
  - Allocates memory that may never be used
- Modern OSes allocate memory lazily
  - Insert physical pages when they're accessed instead of in advance



On-demand page allocation demo

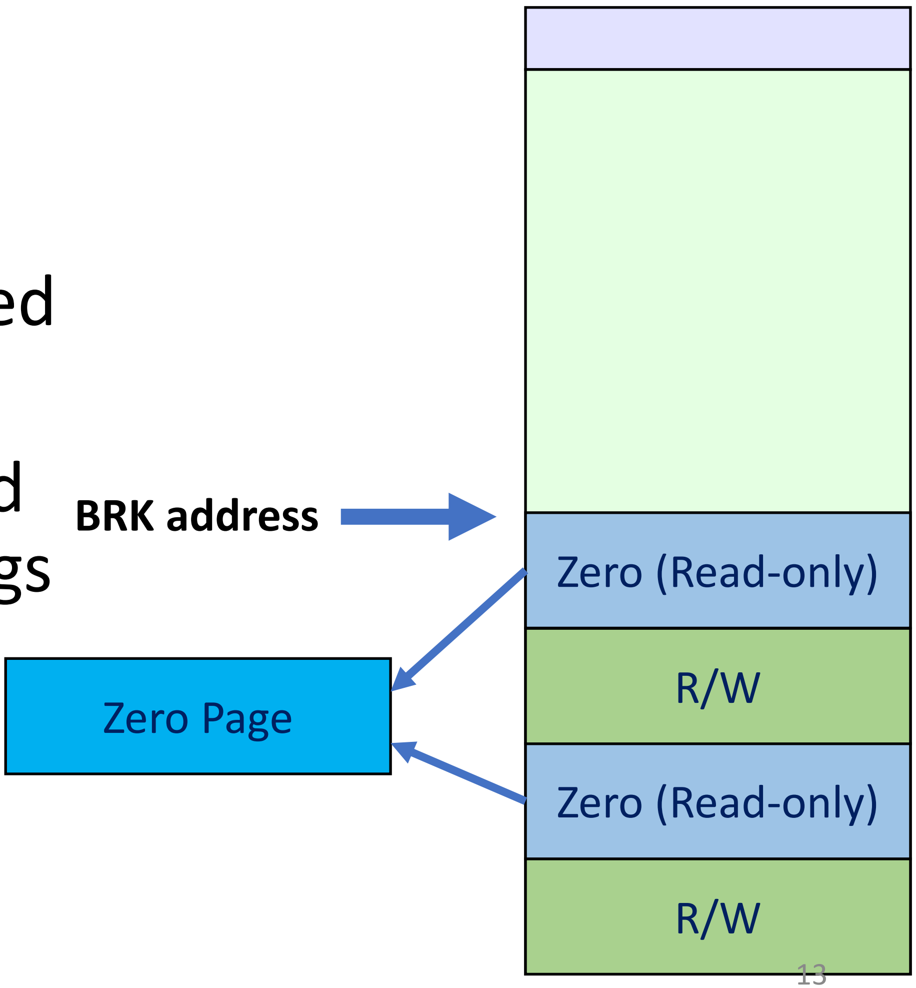
Caveats

- Page faults below user stack are invalid
- Page faults too high could overwrite the kernel
- Many more caveats…

- Real kernels are difficult to build, every detail matters

Optimization: Zero pages

- Observation: In practice, some memory is never written to
- All memory gets initialized to zero
- Idea: Use just **one** zeroed page for all zero mappings
- Copy the zero page on write



Feature: Stack guard pages

- Observation: Stack has a finite size
- Push too much data and it could overflow into adjacent memory
- Idea: Install an empty mapping (PTE_V cleared) at the bottom of the stack
- Could automatically increase stack size in page fault handler

Optimization: Copy-on-write fork()

- Observation: Fork() copies all pages in new process
- But often, exec() is called immediately after fork()
  - Wasted copies
- Idea: modify fork() to mark pages copy-on-write
  - All pages in both processes become read-only
  - On page fault, copy page and mark R/W
  - Extra PTE bits (RSV) useful for indicating COW mappings

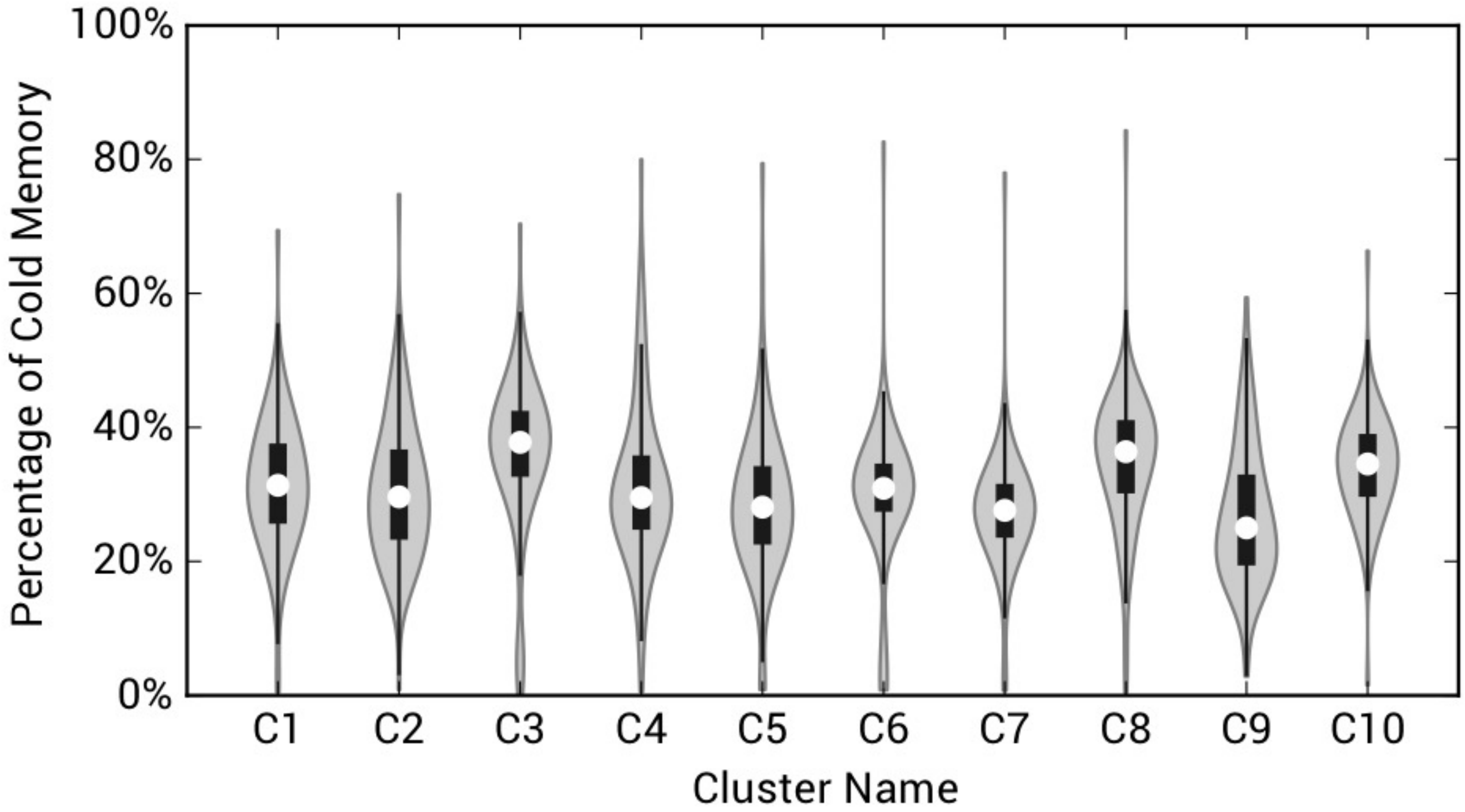
Optimization: Demand paging

- Observation: exec() loads entire object file into memory
  - Expensive, requires slow disk block access
  - Maybe not all of the file will be used
- Idea: Mark mapping as demand paged
  - On page fault, read disk block and install PTE
- Challenge: What if file is larger than physical memory?

Feature: Support more virtual memory than physical RAM

- Observation: More disk capacity than RAM
- Idea: “Page in” and out data between disk and RAM
  - Use page table entries to detect when disk access is needed
  - Use page table to find least recently used disk blocks to write back
- Works well when working set fits in RAM

Opportunity is large



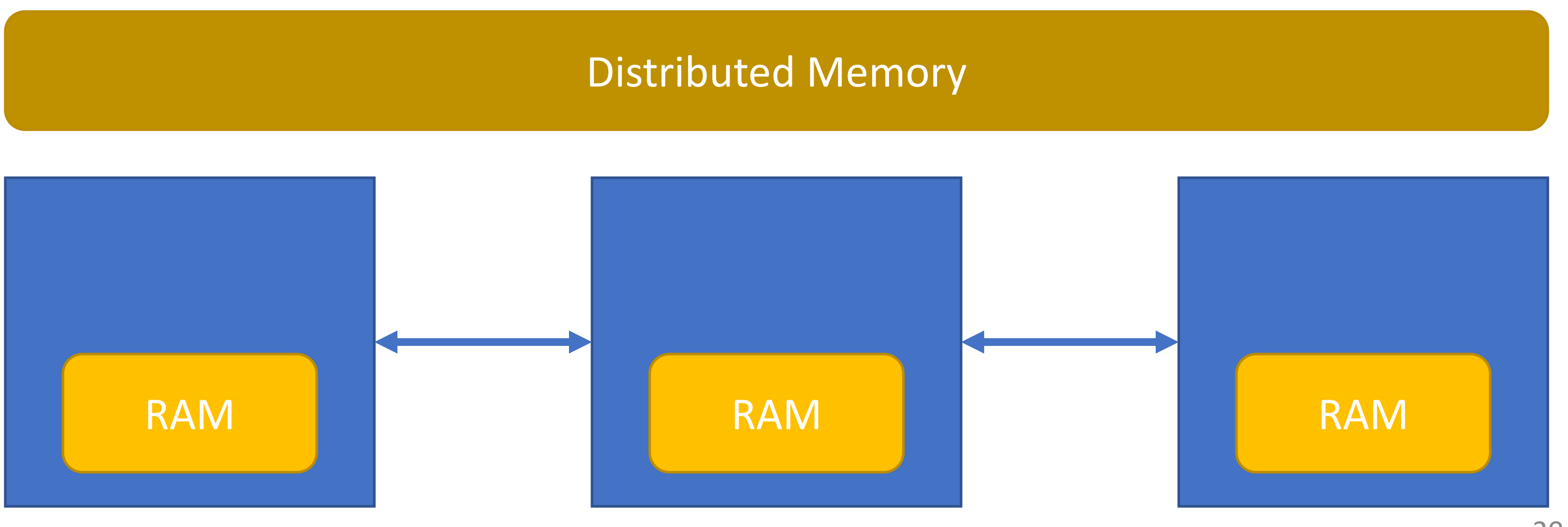
Software-Defined Far Memory in Warehouse-Scale Computers
Lagar-Cavilla et. Al. ASPLOS’19.

Feature: Memory-mapped files

- Normally files accessed through read(), write(), and lseek()
- Idea: Use load and store to access file instead
  - New system call mmap() can place file at location in memory
  - Use memory offset to select block rather than seeking
- Any holes in file mappings require zeroed pages!

Feature: Distributed shared memory

- Idea: Use virtual memory to pretend that physical memory is shared between several machines on the network



Optimization: TLB management

- CPUs cache paging translations for speed
- xv6 flushes entire TLB during user/kernel transitions
  - Why?
- RISC-V TLB is sophisticated in reality
  - **PTE_G**: global TLB bits
  - **SATP**: takes ASID number
  - **sfence.vma**: ASID number, addr
  - **Large pages**: 2MB and 1GB support

Virtual memory is still evolving

Recent Linux Kernel Changes:

- Support for 5-level page tables
  - 57 address bits!
- Support for ASIDs
  - TLB can cache multiple page tables at a time

Conclusion

- There’s no one way to design an OS
  - Many OSes use virtual memory
  - Enables powerful features and optimizations
- xv6 presents one example of OS design
  - They lack many features of real OSes
  - But still quite complex!
- Our goal: Teach you ideas so you can extrapolate