Based on slides by Harsha V. Madhyastha

EECS 482 Introduction to Operating Systems

Spring/Summer 2020

Lecture 13: Kernel vs. user address spaces

Nicole Hamilton

https://web.eecs.umich.edu/~nham/

nham@umich.edu

Agenda

1. Midterm.
2. Review of paging and eviction.
3. Kernel vs. user.

Agenda

1. Midterm.
2. Review of paging and eviction.
3. Kernel vs. user.

Midterm exam

Online using *Crabster.org* Wed Jun 24 3:00 to 5:00 pm EDT.

We will post a link via Canvas and Piazza once the exam once it's live.

We will monitor Piazza for questions.

Material for midterm:

1. All the lecture topics from start until end of lecture 9 on deadlock.
2. All the labs on these topics.
3. Projects 1 and 2.

Next week will be a break between spring and summer semesters and for Independence Day.

We will meet again on July 6.

Midterm policy

The exam will be *“open everything”* except collaboration.

You can use any existing resource, including lecture notes, the book, your P1 and P2 solutions, you can use Google, and your IDE.

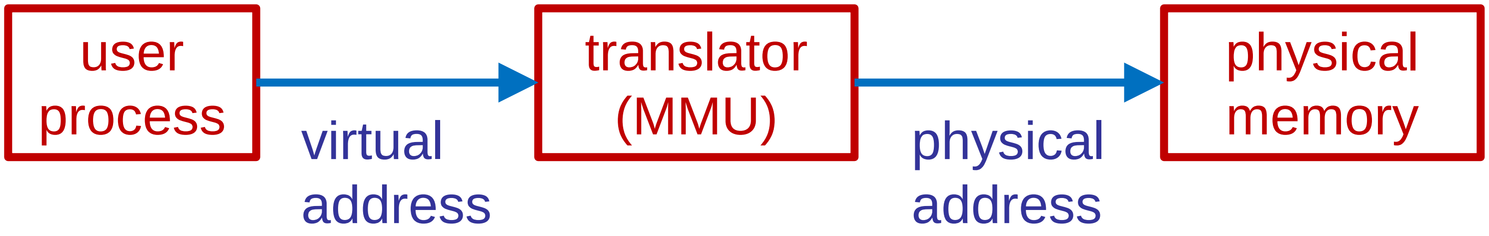
The only thing you can’t do is collaborate with others, including using social media to solicit help. If you can find an existing answer on stackexchange that’s helpful, that’s fair game. But you can’t post a question.

Also, parts of the exam ask for short answers, which must be *in your own words*. Cutting and pasting word-for-word from an existing source and “close copying” will be treated as plagiarism and reported to the Honor Council.

Agenda

1. Midterm.
2. Review of paging and eviction.
3. Kernel vs. user.

Dynamic address translation



Break the requirement that the process space be contiguous.

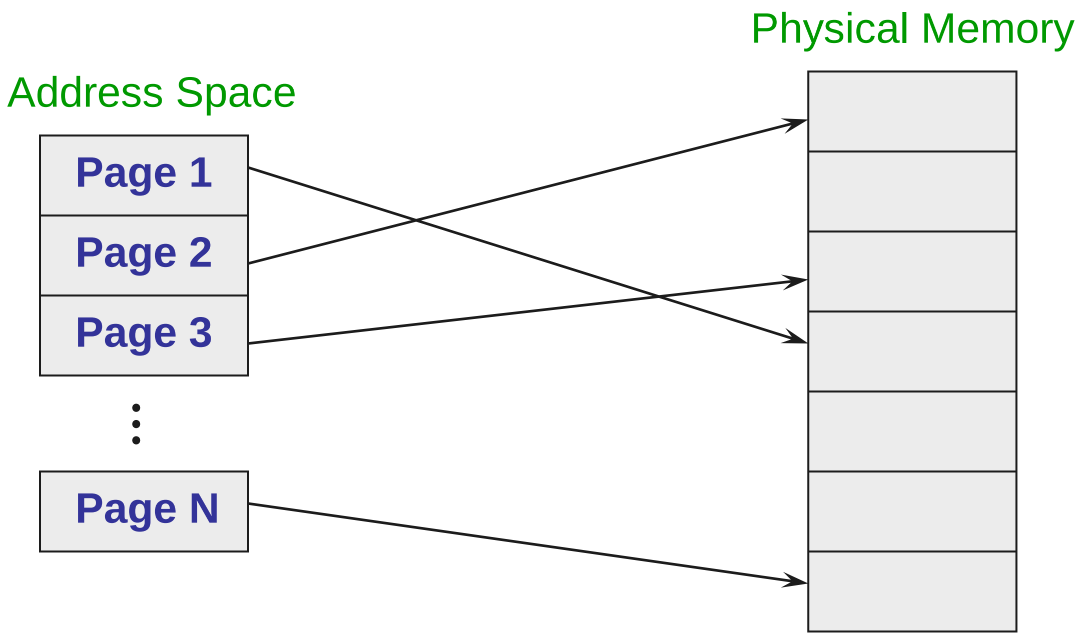
MMU strategies we'll discuss:

1. Base and bounds.
2. Segmentation.
3. Paging.

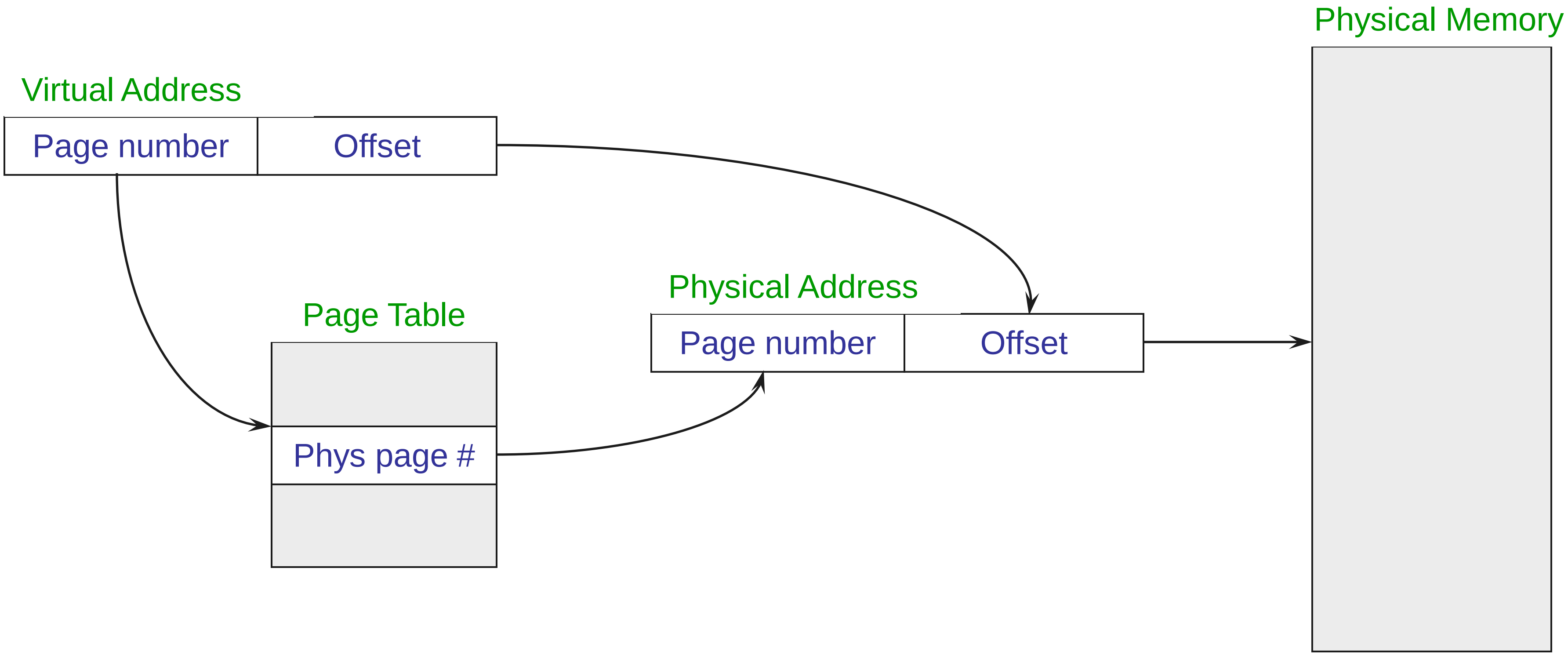
Paging

Allocate phys. memory in fixed-size units (pages).

Any free physical page can store any virtual page.



Page Lookups



Paging

Virtual page #	Physical page #	Resident	Protection
0	105	0	RX
1	15	1	R
2	283	1	RW
3	invalid		
...	invalid		
1048575	invalid		

Valid → virtual page is legal for process to access.

Resident → virtual page is valid and in physical memory.

Error to access invalid page, but not to access non-resident page.

Paging

Virtual page #	Physical page #	Resident	Protection
0	105	0	RX
1	15	1	R
2	283	1	RW
3	invalid		
...	invalid		
1048575	invalid		

Causes of a page fault:

1. Invalid page or disallowed by protection bits, often a user bug.
2. Not resident and must be brought in by the OS.

Paging

Each virtual page can be in physical memory or “paged out” to disk.

Pages can also have different protections (e.g., read, write, execute).

```
MMU_translation( )
   {
   if ( virtual page is
         invalid or non-resident or protected )
      trap to OS fault handler;
   else
      {
      physical page # =
         pageTable[ virtual page # ].physPageNum;
      physical address =
         concat( Physical page #, offset );
      }
   }
```

Page table size

Page size is typically 4 KB or 8 KB.

Some architectures support multiple page sizes.

Each process with a 32-bit address space with 4-byte page table entries requires:

$$\left(\frac{2^{32}}{4096}\right) * 4 = 4\ MB$$

A 64-bit address space with 8-byte entries requires:

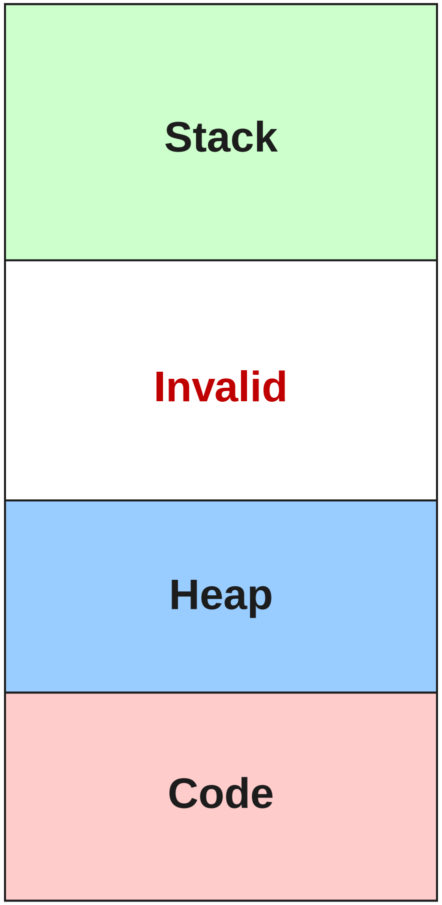
$$\left(\frac{2^{64}}{4096}\right) * 8 = 3.6 * 10^{16} = 36\ PB$$

Sparse address space

Most processes use only a tiny fraction of their 32 or 64-bit address space.

They usually have a huge hole in the middle.

So we only need to represent that part of the page table that isn’t marked invalid.



Stack

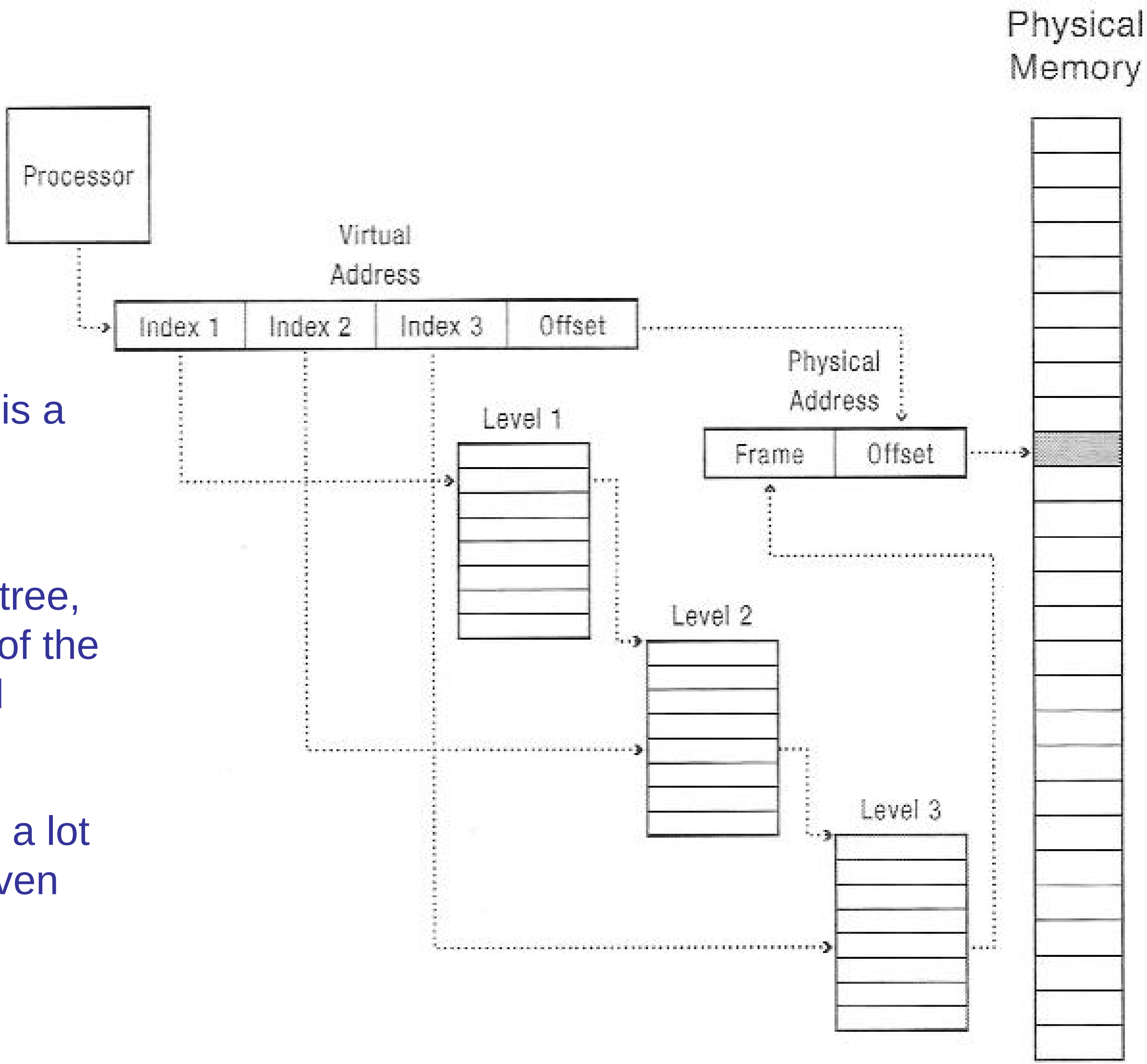
Invalid

Heap

Code

Virtual page #	Physical page #
0	105
1	15
2	283
3	invalid
...	invalid
1048572	invalid
1048573	1078
1048574	48136
1048575	60

Multi-level paging



A standard page table is a simple array.

Multi-level paging generalizes this into a tree, filling in only the parts of the tree that aren’t marked invalid.

With multilevel paging, a lot of the entries in any given page table will be null.

Image source: Anderson & Dahlin, *Operating Systems: Principles and Practice*, p. 398.

Multilevel paging

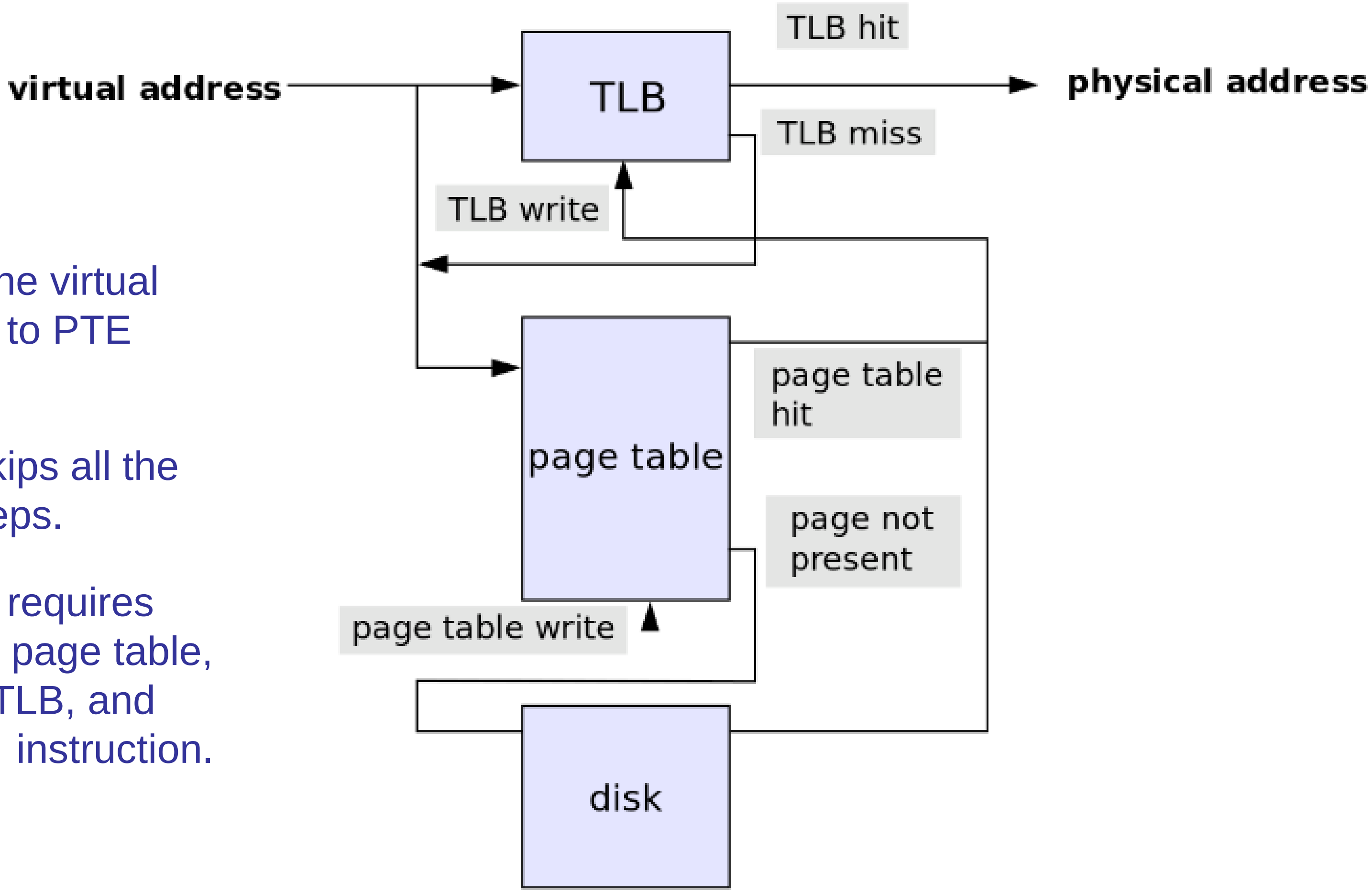
Pros

1. Simple memory allocation.
2. Flexible sharing.
3. Easy to grow address space.
4. Space-efficient representation of the page table.

Cons

1. Two or more extra lookups per memory reference.

The solution is to cache parts of the page table in hardware.

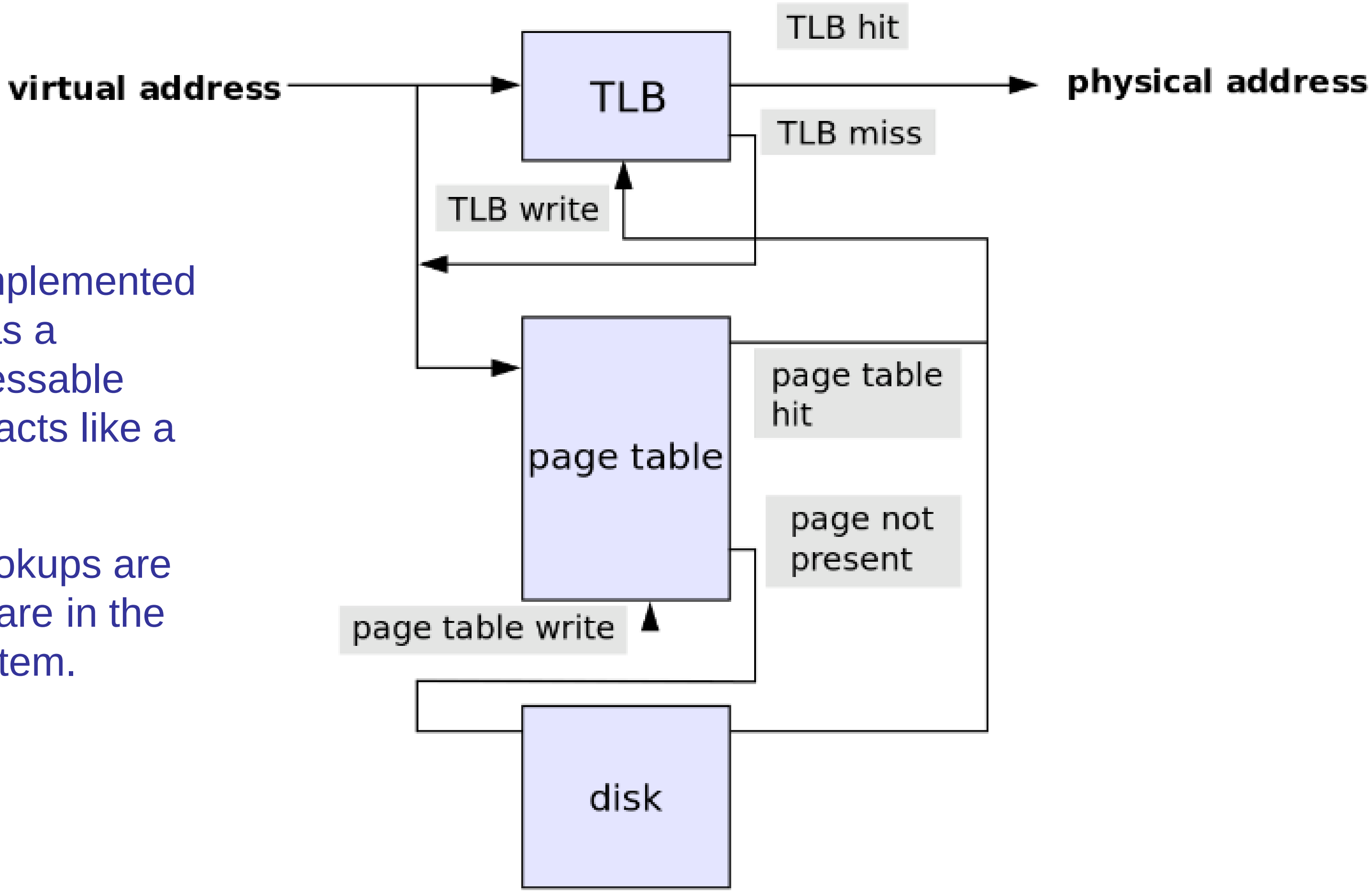


TLB caches the virtual page number to PTE mapping.

A *cache hit* skips all the translation steps.

A *cache miss* requires searching the page table, updating the TLB, and restarting the instruction.

Image source: Wikipedia, *“Page table”.*



The TLB is implemented in hardware as a content-addressable memory that acts like a map in C++.

Page table lookups are done in software in the operating system.

Image source: Wikipedia, *“Page table”.*

Page replacement

Not at all valid pages can be in physical memory.

Must decide how to handle loads/stores to non-resident pages.

Sometimes, we will need to *evict* a page to make room for another.

Page Replacement

Not all valid pages may fit in physical memory

Some pages must be paged out (written) to disk.

Disk is the “backing store”, physical mem acts as cache.

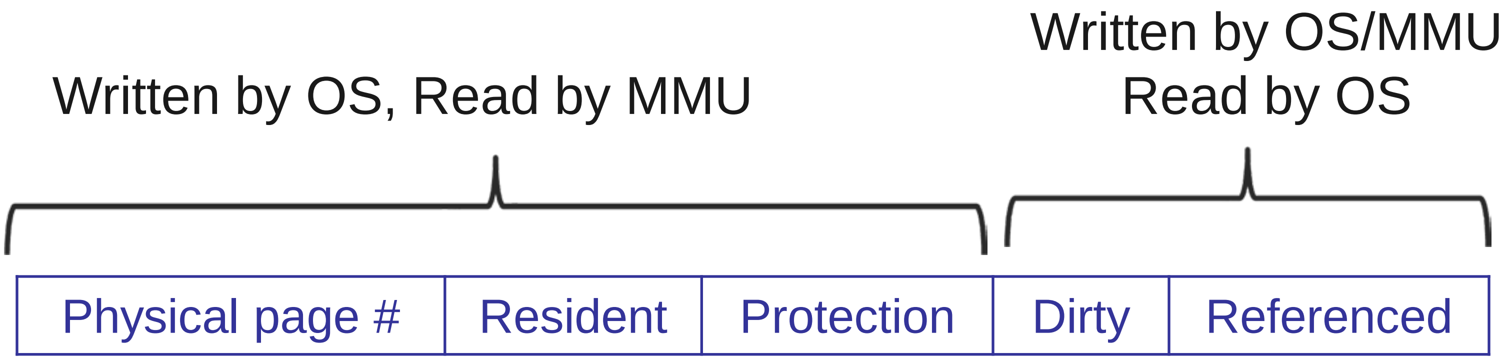
To read in a page from disk, some resident page may need to be paged out, *“evicted”,* first.

Need an algorithm to decide which page to evict to make space.

Goal: minimize page faults.

LRU hard to beat but sometimes fails.

Page table entry



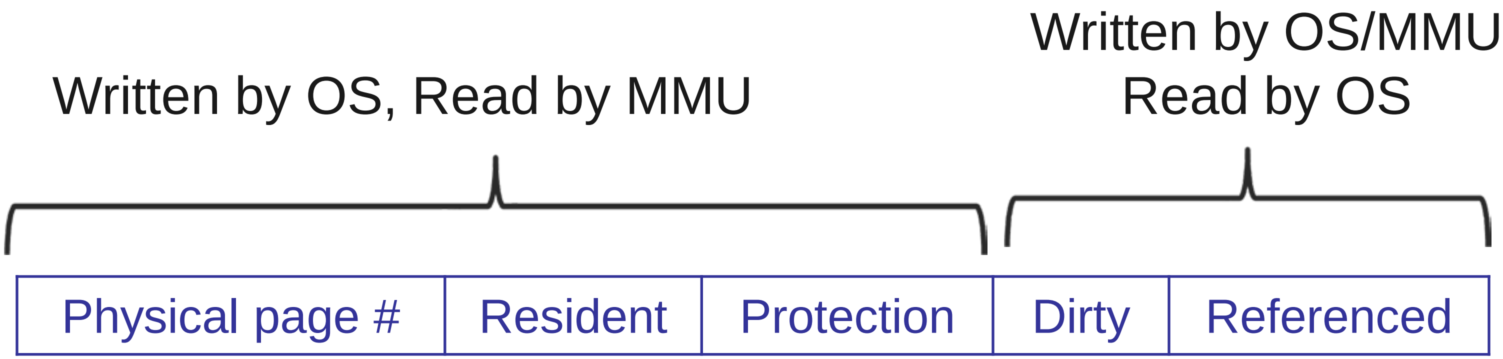
Written by OS, Read by MMU

Written by OS/MMU
Read by OS

Physical page #	Resident	Protection	Dirty	Referenced

```
MMU_translation( )
   {
   if ( virtual page is invalid or non-resident or protected )
      trap to OS fault handler;
   else
      {
      physical page # = pageTable[ virtual page # ].physPageNum;
      pageTable[ virtual page # ].referenced = true;
      if ( access is write )
         pageTable[ virtual page # ].dirty = true;
      physical address = concat( Physical page #, offset );
      }
   }
```

Page table entry



Written by OS, Read by MMU

Written by OS/MMU
Read by OS

Physical page #	Resident	Protection	Dirty	Referenced

Why no valid bit in PTE?

All invalid virtual pages are non-resident.

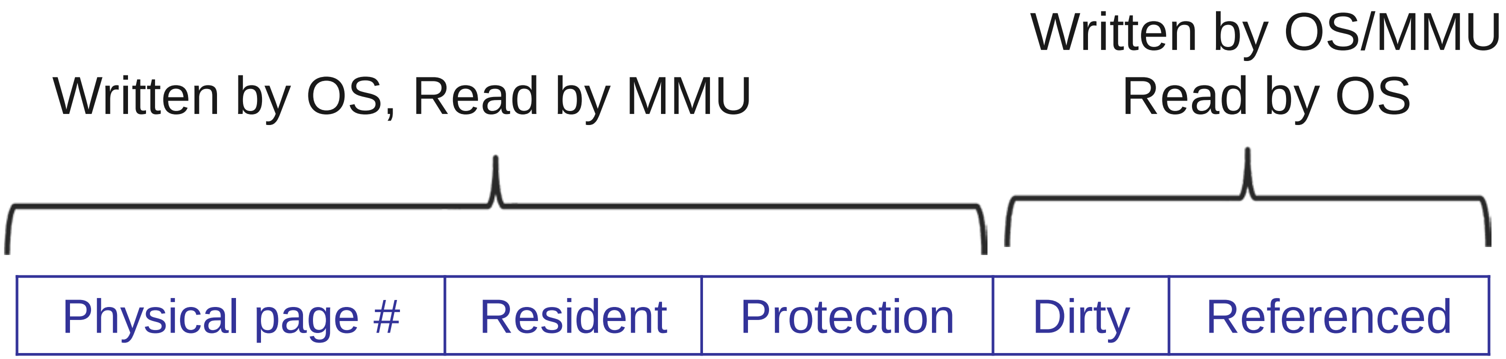
Valid non-resident pages: where’s the disk block?

OS must maintain this, MMU simply traps to OS.

How can we make do without resident bit?

Clear protection bits when non-resident to cause hardware fault.

Page table entry



Written by OS, Read by MMU

Written by OS/MMU
Read by OS

Physical page #	Resident	Protection	Dirty	Referenced

Can we make do without the dirty bit?

Have OS set the dirty bit itself.

Naive solution: Trap on every store & mark dirty.

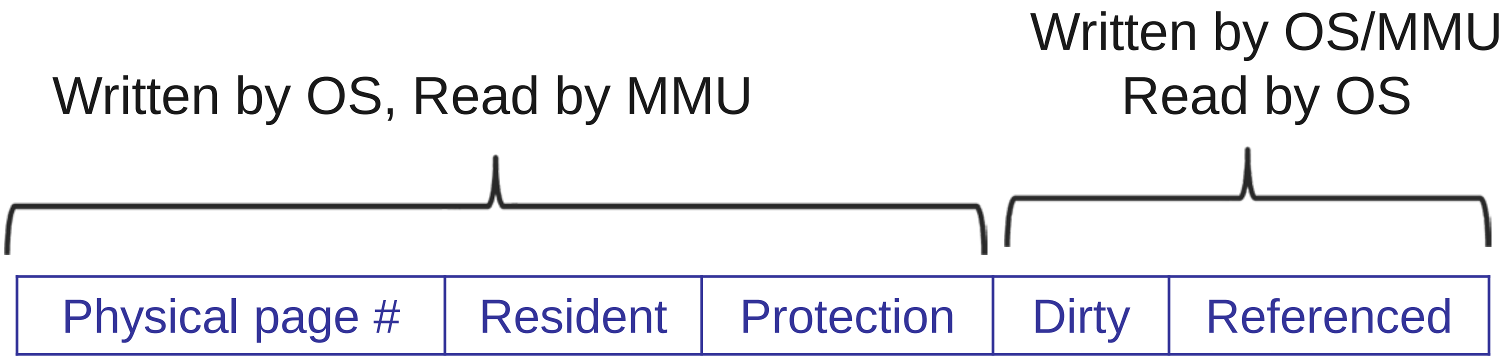
How to reduce # of page faults?

Only care about transition from clean to dirty.

Clean pages have 0 write protection bit.

Dirty pages have usual value.

Page table entry



Written by OS, Read by MMU

Written by OS/MMU
Read by OS

Physical page #	Resident	Protection	Dirty	Referenced

Can we make do without referenced bit?

Can use similar trick as that used for dirty bit.

Insight: only care about unreferenced → referenced.

These changes make the MMU hardware simpler but the page fault handler more complex.

Agenda

1. Midterm.
2. Review of paging and eviction.
3. Kernel vs. user.

Address space management

How to manage a process’s accesses to its address space?

1. Kernel sets up page table per process and manages which pages are resident.
2. MMU looks up page table to translate any virtual address to a physical memory addresses.

What about kernel’s address space?

How should MMU handle kernel’s loads and stores?

Storing page tables

Two options:

1. In physical memory.
2. In kernel's virtual address space.

Difference: Is the address held in the page table base register (PTBR) a physical or virtual address?

Pros and cons?

Simplicity versus being able to page parts of the kernel (which could be huge.)

Project 3 uses the second option:

Kernel's address space is managed by infrastructure.

Kernel vs. user address spaces

Can you evict the kernel’s virtual pages?

Yes, except code for handling paging in/out is pinned

How can kernel access specific physical memory addresses (e.g., to write to page table)?

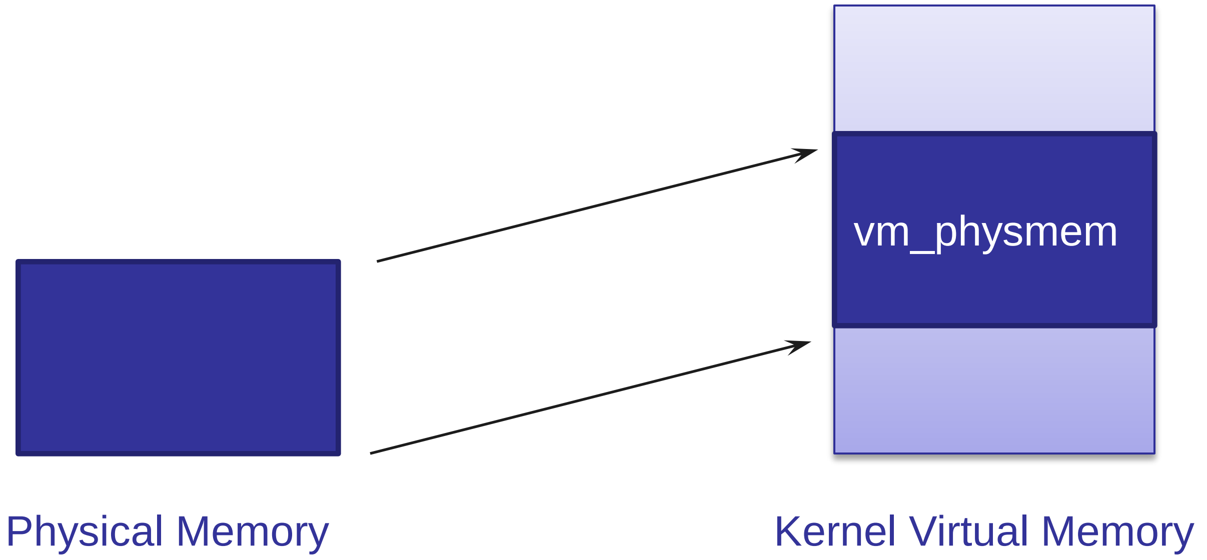
1. Kernel can issue untranslated address (bypass MMU).
2. Kernel can map physical memory into a portion of its address space (e.g., `vm_physmem` in Project 3).

Accessing physical memory

How does the kernel access physical memory?

Could map physical memory 1-to-1 into window in virtual address space

`vm_physmem[n]`: n^{th} byte of physical memory



How does the kernel access the user's address space?

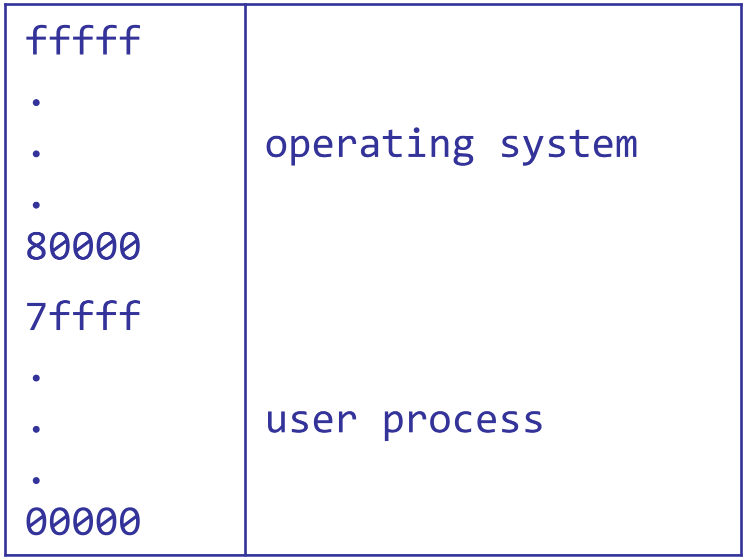
User and kernel often need to share data, e.g., arguments to a system call.

Options:

1. Kernel could manually translate a user virtual address to a physical address, then access the physical address.
2. Can map kernel address space into every process's address space.

Concerns?

Ordinary users should not have kernel access.



fffff
.
.
.
80000

operating system

7ffff
.
.
.
00000

user process

Protection: kernel/user mode

How are we protecting a process's address space from other processes?

1. Page tables: Dynamic translation to disjoint physical memory.
2. Must ensure only the kernel can modify translation data.

How does CPU know the kernel is running?

Hardware support with a mode bit to indicate when we're running in kernel (privileged) vs. user mode

How do we protect the mode bit from being changed by the user?

We make the user do a system call, but how does that work?

Switching to kernel mode

Faults and interrupts

1. Timer interrupts.
2. Page faults.

Why are these safe to transfer control to kernel?

User doesn't get to choose what runs when the interrupt is taken.

System calls

1. Process management: fork()/exec()
2. I/O: open(), close(), read(), write()
3. System management: reboot()
4. …

System calls

When you call `cin` in your C++ program:

1. `cin` calls `read()`, which executes assembly-language instruction `syscall`.
2. `syscall` traps to kernel at pre-specified location.
3. kernel’s `syscall` handler calls kernel’s `read()`.

To handle trap to kernel, hardware atomically:

1. Sets the mode bit to kernel.
2. Saves registers, PC, SP.
3. Changes SP to kernel stack.
4. Changes to the kernel’s address space.
5. Jumps to exception handler.

Arguments to system calls

Two options:

1. Store in registers.
2. Store in memory (in whose address space?)

Kernel first checks validity of arguments:

```
read( int fd, void *buf, size_t size )
```

Is `fd` a valid descriptor for open file?

Are all addresses in `[buf,buf+size)` valid?

Are all addresses in `[buf,buf+size)` writable?

Protection summary

Safe to switch from user to kernel mode because control only transferred to certain locations and the number of entry points to the system is kept limited to limit the *attack surface*.

Where are these locations stored?

Interrupt vector table.

Who can modify interrupt vector table?

Only the kernel. Set once at boot, then never changed.

Why is it easier to control access to interrupt vector table than mode bit?

The mode bit changes constantly and every change must be scrutinized. But the IVT never changes.

Address Space Protection

How are address spaces protected?

Separation of translation data, meaning the translation data must be protected.

How is translation data protected?

Can update translation data only if mode bit set.

How is mode bit protected?

Sets/reset mode bit when transitioning from user-level to kernel-level code and back.

Transitions limited by interrupt vector table.

Protection boils down to init process which sets up interrupt vector table when system boots up.

Project 3

Process view:

1. Every process has an address space starting from `VM_ARENA_BASEADDR` of size `VM_ARENA_SIZE`.
2. When a process starts, the entire address space is invalid.
3. Process calls `vm_map` to make pages valid.
4. Pages becomes invalid when a process ends.

Pager view:

1. One process runs at a time.
2. Sets up page table that the MMU uses for translation.
3. Handles `vm_create`, `vm_map`, and `vm_fault`.

Project 3

Swap-backed pages:

1. Global swap file shared by all processes.
2. Pager controls where in swap file page is stored.
3. Private to a process.

File-mapped pages:

1. Process specifies (file, offset).
2. Can be shared across processes.

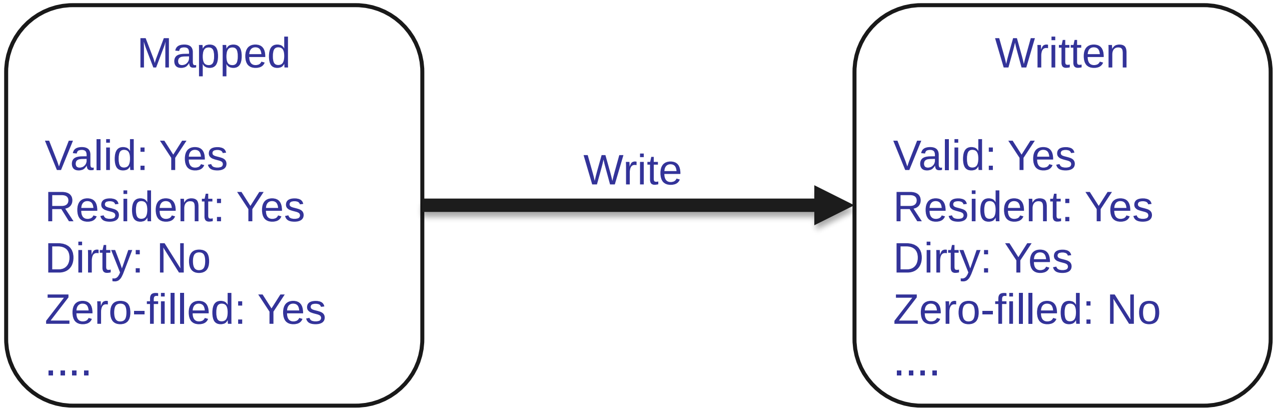
Project 3

1. Do the project incrementally.
2. Swap-backed pages only without fork.
3. Then add support for fork and file-backed pages one after the other.
4. Pro Tip: Start with state diagrams for swap-backed, file-backed pages.

Project 3: State Diagram

For each unique state, consider:

1. Transitions? Read, write, clock, copy, ...
2. Attributes? Valid, resident, dirty, ...
3. Protections? Enable read, enable write?



Mapped

Valid: Yes
Resident: Yes
Dirty: No
Zero-filled: Yes
....

Write

Written

Valid: Yes
Resident: Yes
Dirty: Yes
Zero-filled: No
....