CS323 Operating Systems

Virtual Memory (Paging and Swapping)

Mathias Payer and Sanidhya Kashyap

EPFL, Fall 2021

Topics covered in this lecture

- Abstraction: address space
- Mechanism: virtual address translation
- Mechanism: paging
- Mechanism: swapping

This slide deck covers chapters 18–22 in OSTEP.

Address spaces

- The ***address space*** encapsulates all addressable memory
- A system has a certain amount of physical memory, this results in the available physical memory
- For simplicity, we assume that each address points to a byte
- Virtual and physical address space size can differ both ways
- The physically present memory is smaller or equal to the physical address space size

Example: modern 64-bit CPUs have a 48-bit virtual address space (with 64 bit pointers) and map to 48 bit of physical address space. Most machines have less than 256 TiB of memory.

Memory abstraction

The seven layers of memory abstraction complexity:

- No virtualization, program runs on bare metal
- Time sharing, one program at a time
- Space sharing, relocate programs in one shared address space
- Space sharing, virtual address space through segment (base)
- Space sharing, virtual address space through segment (base+bounds)
- Space sharing, virtual address space through multiple segments
- Space sharing, virtual address space through paging

Match the description

- One process uses all of memory
- Share address space, use a per-process offset
- Verify address is in usable part of the address space
- Several base+bounds pairs per process
- Rewrite code and data before running

Options: base register, static relocation, segments, base & bounds, time sharing

Limits of segmentation

Segmentation allows reasonable sharing of a physical address space among several processes. What are the drawbacks?

Limits of segmentation

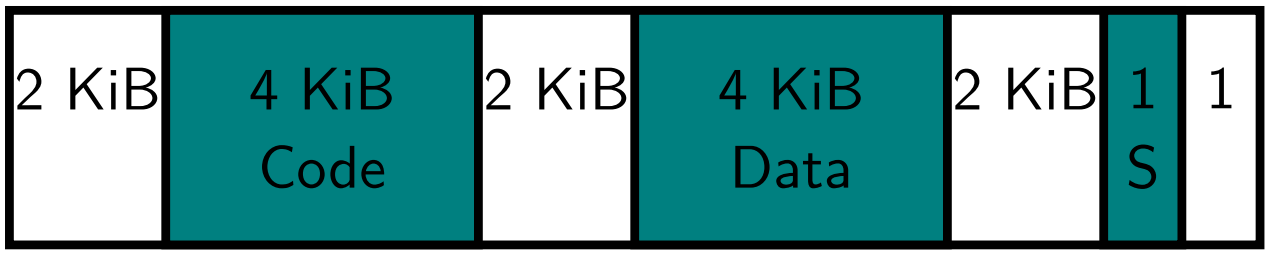
Segmentation allows reasonable sharing of a physical address space among several processes. What are the drawbacks?

- Space: each segment must be fully backed by memory
- Space: physical memory area must be contiguous (fragmentation)
- ISA: instructions explicitly or implicitly encode target segment

Fragmentation

Fragmentation: *unused memory that cannot be used.*
External fragmentation: *visible to allocator (OS), e.g., between segments.*
Internal fragmentation: *visible to requester, e.g., rounding if segment size is a power of 2.*

For example: we have 16 KiB of memory. Process A has a code segment of 4 KiB starting at 2,048, a data segment of 4 KiB starting at 8,096 and a 1 KiB stack segment starting at 14,336. The system has $2+2+2+1 = 7$ KiB of free memory, but the maximum contiguous space is 2 KiB. Starting process B that requires a code segment of 3 KiB would not be possible.



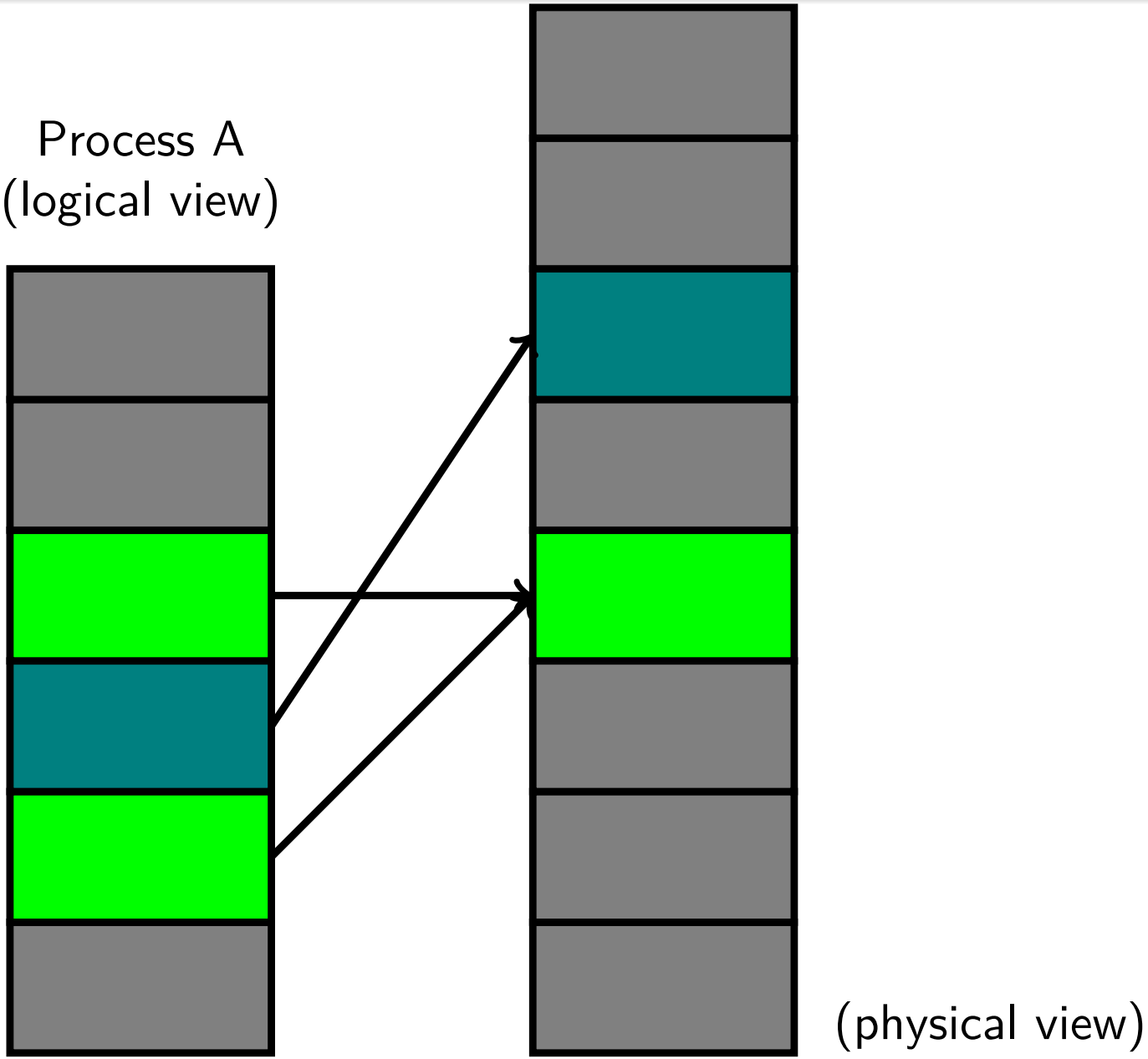
A more complex MMU: paging

- Goal: eliminate requirement that physical memory is contiguous
  - Eliminate external fragmentation
  - Grow (and shrink?) segments as needed
- Idea: break address spaces and physical memory into pages
  - Assign memory based on page granularity
  - Still prone to internal fragmentation (round to page size)

Page

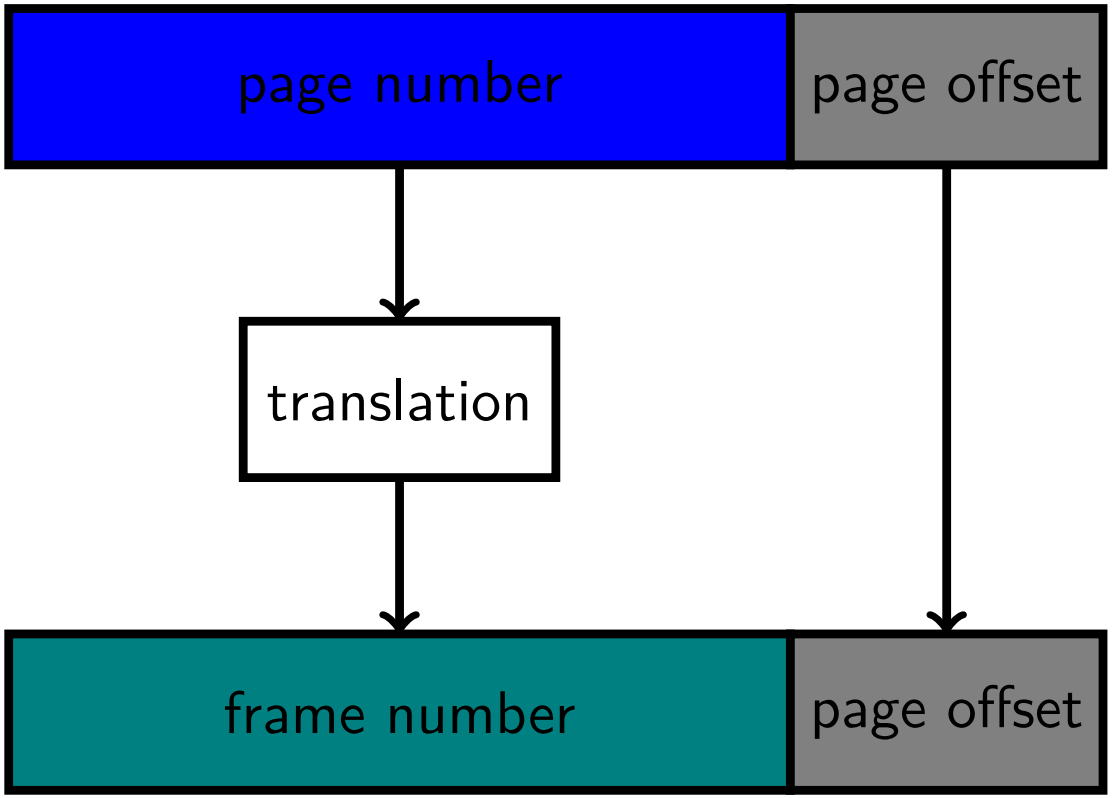
- A page is the minimal unit to break up an address space
- For processes (virtual address space) this is called a (virtual) page
- As part of the physical address space this is called a frame (or physical page)
- A system with 12 bit pages implies that
  - the lowest 12 bit of the address specify the page offset
  - A page's size is 2^12 bytes (4 KiB)

Paging



Paging: address translation

- How can the MMU translate virtual to physical addresses?
  - High order bits designate page number
  - Low order bits designate offset in page
  - Note: size of virtual and physical address space may be different



Paging: address space calculations

- Virtual address space: 4 GiB (i.e., 32 bits)
- Physical address space: 256 TiB (i.e., 48 bits)
- Page size: 4 KiB (i.e., 12 bits)

Paging: address space calculations

- Virtual address space: 4 GiB (i.e., 32 bits)
- Physical address space: 256 TiB (i.e., 48 bits)
- Page size: 4 KiB (i.e., 12 bits)

- Page numbers use 20 bits (i.e., each process may reference 2^{20} pages of 4 KiB–2^{12}–size)
- Frame numbers use 36 bits (there is a maximum of 2^{36} physical 4 KiB frames in the system)

Insight: not all virtual or physical space must be allocated.

Paging: how to translate from virtual to physical?

Let's assume a 16 bit virtual, a 20 bit physical address space, and 12 bit for pages.

Paging: how to translate from virtual to physical?

Let's assume a 16 bit virtual, a 20 bit physical address space, and 12 bit for pages.

This means there are 16 pages per process and 256 physical pages.

Paging: how to translate from virtual to physical?

Let's assume a 16 bit virtual, a 20 bit physical address space, and 12 bit for pages.

This means there are 16 pages per process and 256 physical pages.

The simplest approach to translate from virtual to physical addresses is through a lookup table for each process. The pointer to the lookup table is stored in a special register (CR3 on x86):

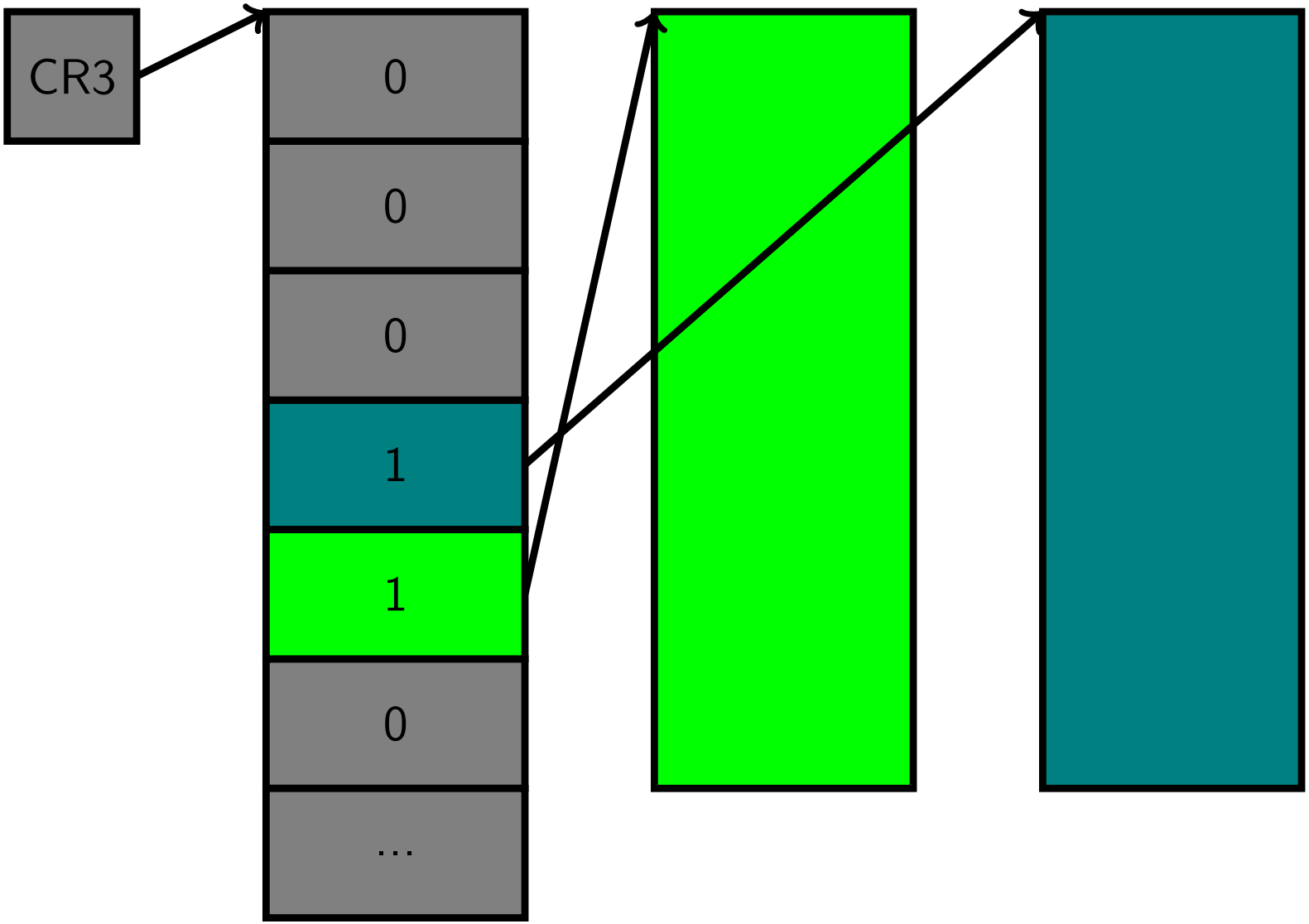
```
unsigned char v1[16];

void *toPhys(void *ptr) {
  void *offset = ptr & (1<<12-1);
  unsigned int idx = (unsigned int)((ptr>>12) & (1<<4-1));
  return (void*)(((unsigned int)(v1[idx])<<12) | offset);
}
```

Paging: storing other per page information

- To align accesses, each entry in 32-bit or 64-bit page tables is 32 or 64 bit (address spaces are byte addressable and bit operations are costly; similarly, power of 2 alignments are faster due to caching).
- Modern systems store much more information in the page table than just the physical frame number:
  - Is the mapping valid?
  - Protection bits (read, write, execute)
  - Is the page present?
  - Is the page referenced?
  - Is the page dirty?

The OS and the MMU agree on the interpretation of these bits.

Paging: example of a flat page table



Paging: pros and cons

- Advantages
  - No external fragmentation
  - Fast to allocate (no searching for space) and free (no coalescing)
  - Simple to adjust what subset is mapped *in core* (later)
- Disadvantages
  - Internal fragmentation (tension regarding page size)
  - Additional memory reference to page table (hint: use a cache)
  - Required space for page table may be substantial

Working set and locality

A program's working set is defined as the subset of the program's code and data that is currently used. This means that the working set is a temporal metric that changes over time as the program makes progress. Most accesses will be confined to few addresses, exhibiting locality.

Working set and locality

> *A program's working set is defined as the subset of the program's code and data that is currently used. This means that the working set is a temporal metric that changes over time as the program makes progress. Most accesses will be confined to few addresses, exhibiting locality.*

Observation: at a given time, a program does not need all data and code to make progress. The OS may reuse memory that is currently not used for other processes (space sharing). The hardware may optimize memory accesses and translation accordingly.

Paging: avoid additional memory access

- Simple idea: remember previous lookups
- Store the last accessed pages in a cache
- This cache is called TLB (Translation Lookaside Buffer)
- Reduces requirement memory accesses for page walk
- If there is an entry in the TLB, reuse!

Paging: 32 bit address space

Assume 12 bit for pages, 32 bit virtual and 32 bit physical address space, 4 byte page table entries.

- What is the size of a flat page table?

Paging: 32 bit address space

Assume 12 bit for pages, 32 bit virtual and 32 bit physical address space, 4 byte page table entries.

- What is the size of a flat page table?

- Page table size: entries * size of entry
- entries: $2^{log(virtualaddressspace)-log(pagesize)}$
- entries: $2^{32-12} = 2^{20} = 1MiB$
- Page table size: $1MiB * 4B = 4MiB$

Paging: 32 bit address space

Assume 12 bit for pages, 32 bit virtual and 32 bit physical address space, 4 byte page table entries.

- What is the size of a flat page table?
- Page table size: entries * size of entry
- entries: $2^{log(virtualaddressspace)-log(pagesize)}$
- entries: $2^{32-12} = 2^{20} = 1MiB$
- Page table size: $1MiB * 4B = 4MiB$
- Let's assume a 32 bit virtual and 48 bit ***physical address space*** and 8 byte entries: 8 MiB

Paging: 32 bit address space

Assume 12 bit for pages, 32 bit virtual and 32 bit physical address space, 4 byte page table entries.

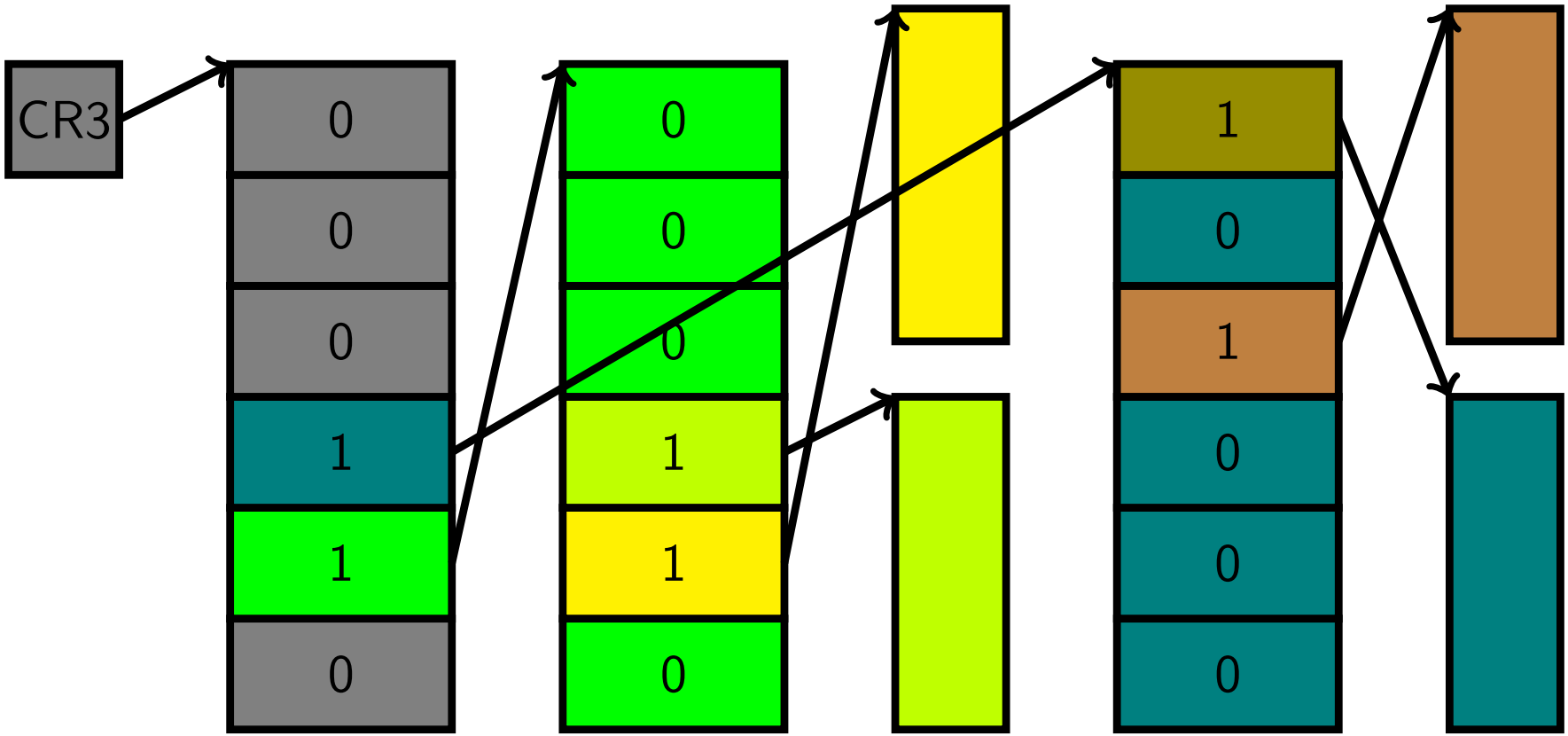
- What is the size of a flat page table?
- Page table size: entries * size of entry
- entries: $2^{log(virtualaddressspace)-log(pagesize)}$
- entries: $2^{32-12} = 2^{20} = 1MiB$
- Page table size: $1MiB * 4B = 4MiB$
- Let's assume a 32 bit virtual and 48 bit ***physical address space*** and 8 byte entries: 8 MiB
- Let's assume a ***64 bit virtual address space*** and 8 byte entries: $2^{52} * 8B$ (32 PiB). Urks, we need something better!

Paging: a multi-level table

Paging was introduced when the average 32 bit system had a total of 4 MiB of memory. 4 MiB of metadata of the page table per process would leave no memory for the OS or the process.

- Insight: most processes only need a fraction of the address space.
- Goal: only allocate metadata for this fraction
- Mechanism: a multi-level lookup table.
  - One or more levels of indirection allow space efficient encoding
  - Each level adds one more memory lookups during address translation
  - What is a good trade-off?

Paging: a multi-level table



Paging: multi-level table

- What size should the individual levels be?
- Idea: use page granularity for 32 bit virtual/physical!
  - 4 KiB pages: 1024 x 4 B entries
  - 1024 entries correspond to 10 bit
  - break address into 10 bit first level, 10 bit second level, 12 bit offset: `0x3ff 0x3ff 0xfff`

Paging: multi-level table

- Let's move to a 64 bit virtual/physical address space, 4 KiB pages
  - 4 KiB pages have space for 512 entries (9 bit)
  - $64 - 12 = 52$, i.e., we would need 6 levels of page tables

Paging: multi-level table

- Let's move to a 64 bit virtual/physical address space, 4 KiB pages
  - 4 KiB pages have space for 512 entries (9 bit)
  - $64 - 12 = 52$, i.e., we would need 6 levels of page tables
- Let's shrink the address space to 48 bit!
  - $48 - 12 = 36$, i.e., we need 4 levels of page tables
  - Page walks are still expensive, need high TLB hit rate for efficiency

Swapping: when main memory runs out

- Observation: main memory may not be enough for all memory of all processes
- Idea: store unused pages of address space on disk
  - Allows the OS to reclaim memory when necessary
  - Allows the OS to over-provision (hand out more memory than physically available)
  - When needed, the OS finds and pushes unused pages to disk
  - Careful strategy needed to avoid deadlock or performance degradation

Swapping: page fault

- The MMU translates virtual to physical addresses using the OS provided data structures (page tables)
- The present bit for each page table entry at each level indicates if the reference is valid, i.e., resides in memory, or not.
  - MMU checks present bit during translation
  - If a page is not present then the MMU triggers a page fault
  - The OS then enforces its policy to handle the page fault
- Virtual to physical translation is transparent to executed instructions, requires HW support

Swapping: page fault handling

- MMU signals CPU to trap and switch to the OS
- Page fault handler checks where (which process and what address) the fault happened
  - Which process? (Locate data structures)
  - What address? (Search page in page table)
- If page is on disk: OS issues load request and tells scheduler to switch to another process
- If page is still in memory or can be reproduced (e.g., a zero page), then the OS creates it and updates data structures
- The OS then continues the faulting process by reexecuting the faulting instruction.

Swapping: page fault handler

- Virtual to physical translation happens for every memory access
- Why does the MMU switch to the OS during a page fault?

Swapping: page fault handler

- Virtual to physical translation happens for every memory access
- Why does the MMU switch to the OS during a page fault?
- OS handles ***policy***, MMU is the ***mechanism***
- MMU handles common case: path to page is valid, page is present
  - Page walk can be highly optimized
  - OS and MMU agree on data structures
- OS handles policy decisions
  - Allows implementation of flexible policies
  - OS decides which pages are allocated and replaced
  - OS may also pass information on to the process, e.g., if an illegal access was made
  - No need for extremely high speed as page faults are rare

How does the CPU execute a read/write operation?

- CPU issues a load for a virtual address (as part of a memory load/store or an executed instruction)
- MMU checks TLB for virtual address
  - TLB miss: MMU executes page walk
    - page table entry is present: update TLB, continue
    - not present but valid: page fault, switch to OS, OS maps page and transparently returns to the process
    - invalid: page fault, switch to OS, OS raises SEG fault and passes it to the program
  - TLB hit: obtain physical address, fetch memory location and return to CPU

Summary

- Fragmentation: space lost due to internal or external padding
- Paging: MMU fully translates between virtual and physical addresses
  - One flat page table (array)
  - Multi-level page table
  - Pros? Cons? What are size requirements?
- Paging and swapping allows process to execute with only the working set resident in memory, remaining pages can be stored on disk

Don't forget to get to fill out the Moodle quiz!