Virtual Memory Part 3

Arkaprava Basu

Dept. of Computer Science and Automation
Indian Institute of Science
www.csa.iisc.ac.in/~arkapravab/

Acknowledgements

- Some of the slides in the deck were provided by Profs. Luis Ceze (Washington), Nima Horanmand (Stony Brook), Mark Hill, David Wood, Karu Sankaralingam (Wisconsin), Abhishek Bhattacharjee(Yale).

- Development of this course is partially supported by funding from Western Digital corporations.

CoLT : Coalesced Large reach TLB

- Observations:
  - Often a few contiguous VA pages maps to contiguous PA pages, naturally (e.g., 32KB)
  - Can use a single TLB entry to map it
- COLT:
  - Dynamically coalesce TLB entries when opportunity exists
  - Transparent to software –only hardware changes

- Reading assignment: “CoLT: Coalesced Large-Reach TLBs” –by Pham et. al.

Segmentation with paging

- 32-bit x86 machines allowed segmentation on top of paging
  - Virtual address is first translated via segment registers (CS,DS,ES etc.) to linear address
  - Linear address is then translated to physical address

Segmentation with paging

- 32-bit x86 machines allowed segmentation on top of paging
  - Virtual address is first translated via segment registers (CS,DS,ES etc.) to linear address
  - Linear address is then translated to physical address
- 64-bit x86 mostly got rid of segmentation

Agenda

What is virtual memory?

Hardware implementations of virtual memory

Software management of virtual memory

Research opportunity in virtual memory

Agenda

What is virtual memory?

Hardware implementations of virtual memory

Software management of virtual memory

Research opportunity in virtual memory

Journey so far….Next up

Mostly H/W

- Virtual memory basics
- Address mapping and translation

Mostly S/W

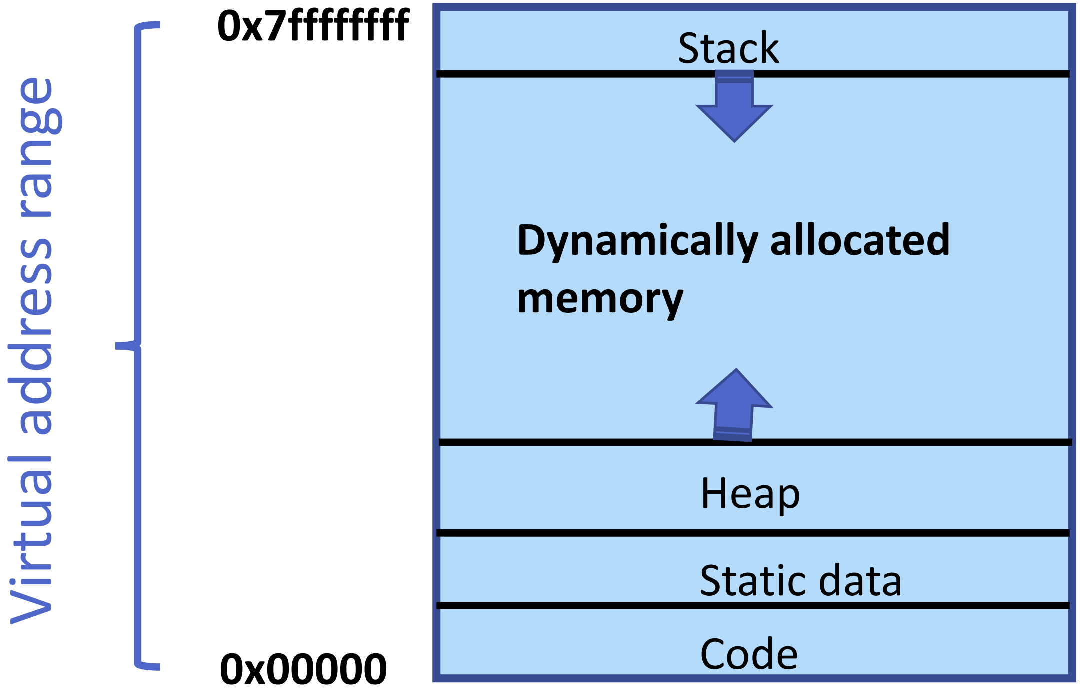
- Virtual address allocation
- Physical memory allocation and page table creation

Virtual address allocation

- "Memory allocation" is actually virtual address allocation
- Operating system is responsible for allocating virtual address for an application

Virtual address allocation

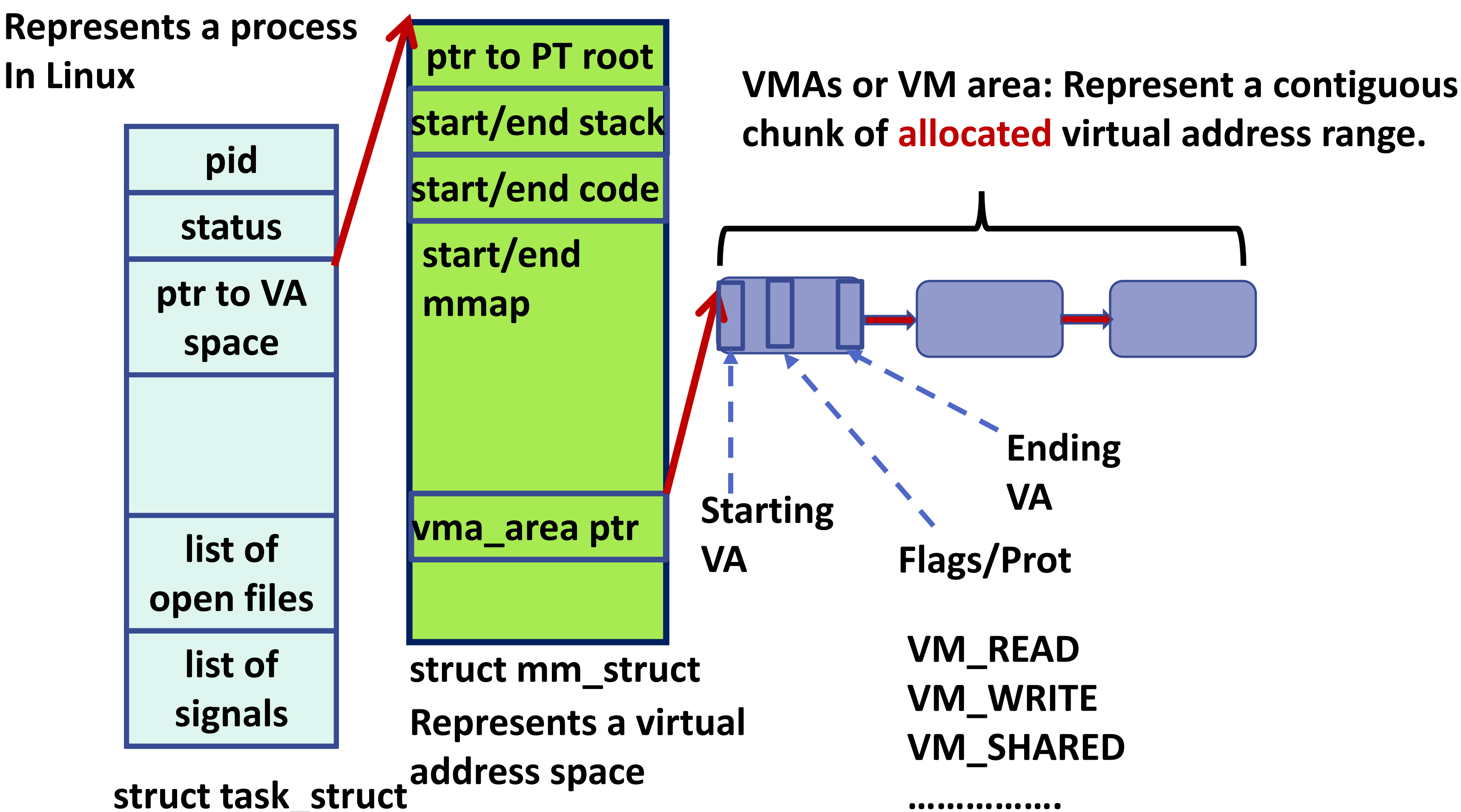
- "Memory allocation" is actually virtual address allocation
- Operating system is responsible for allocating virtual address for an application



Virtual address space

Typical virtual address layout of a process in Linux

Representation of VA (in Linux)



Dynamically allocating VA

- User application or library requests VA allocation via system calls

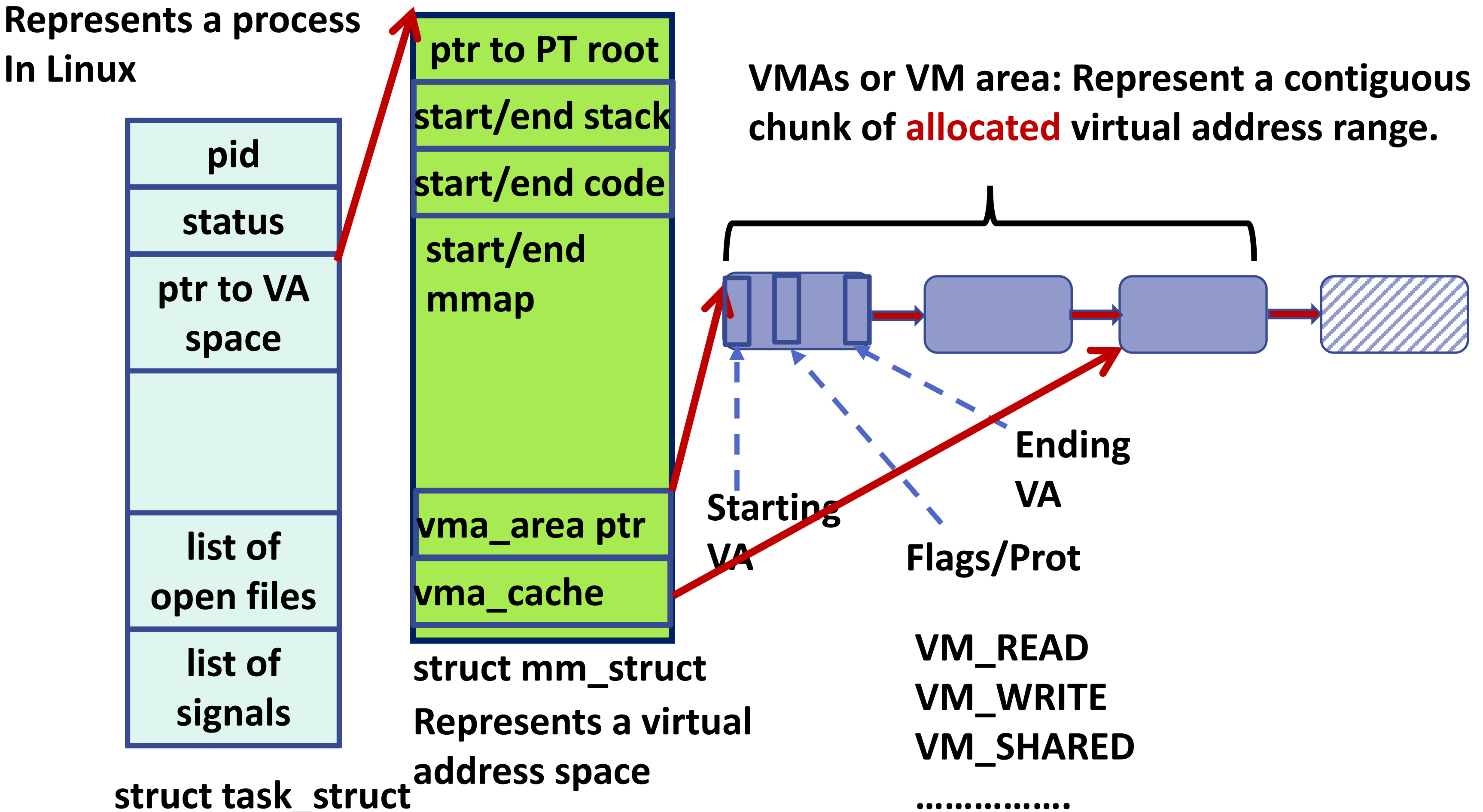
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);

Length has to be multiple of 4KB

Prot → PROT_NONE, PROT_READ, PROT_WRITE…

Flags→ MAP_ANONYMOUS, MAP_SHARED, MAP_PRIVATE, MAP_SHARED

What happens on dynamic memory allocation (e.g., mmap())?

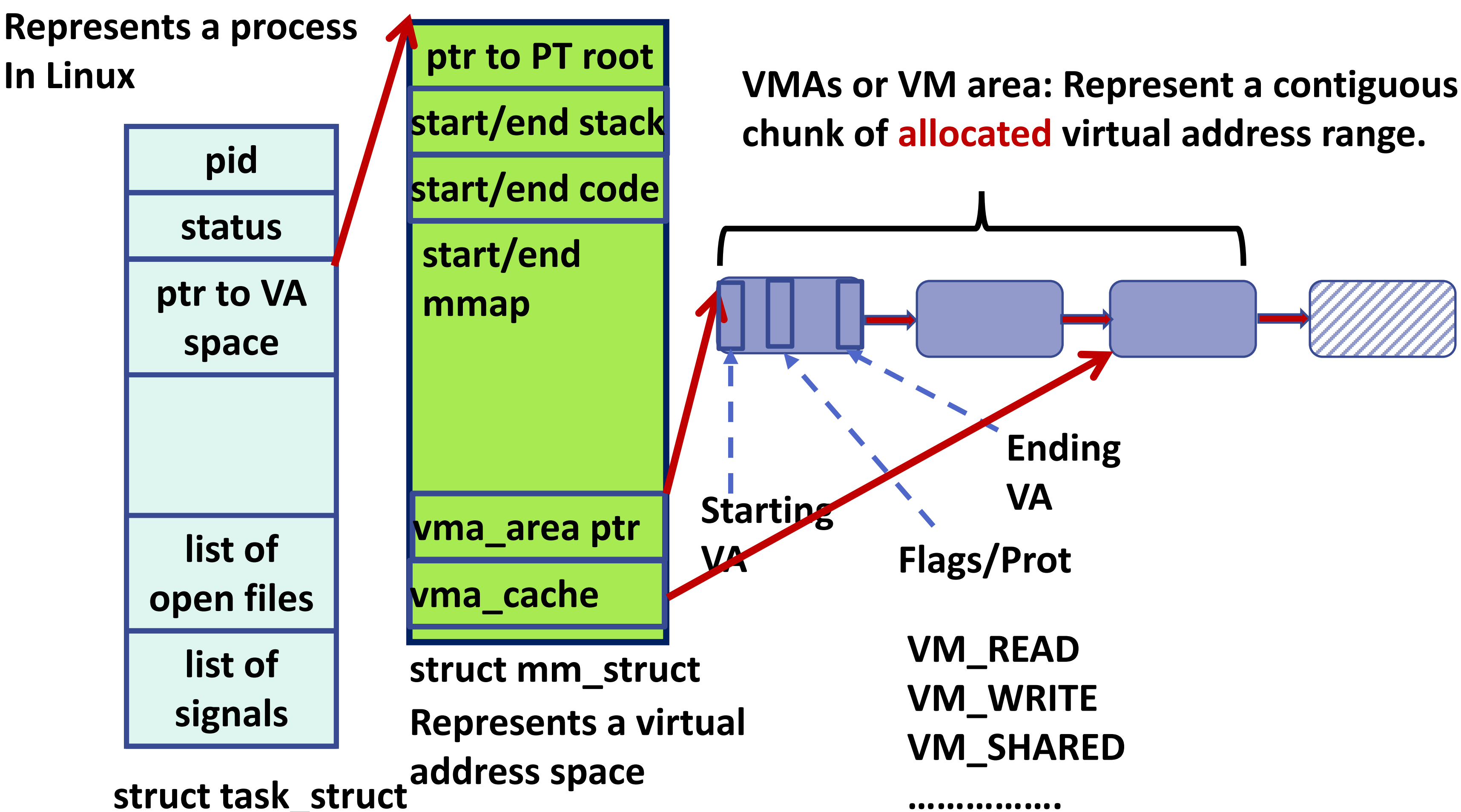


Extending heap memory

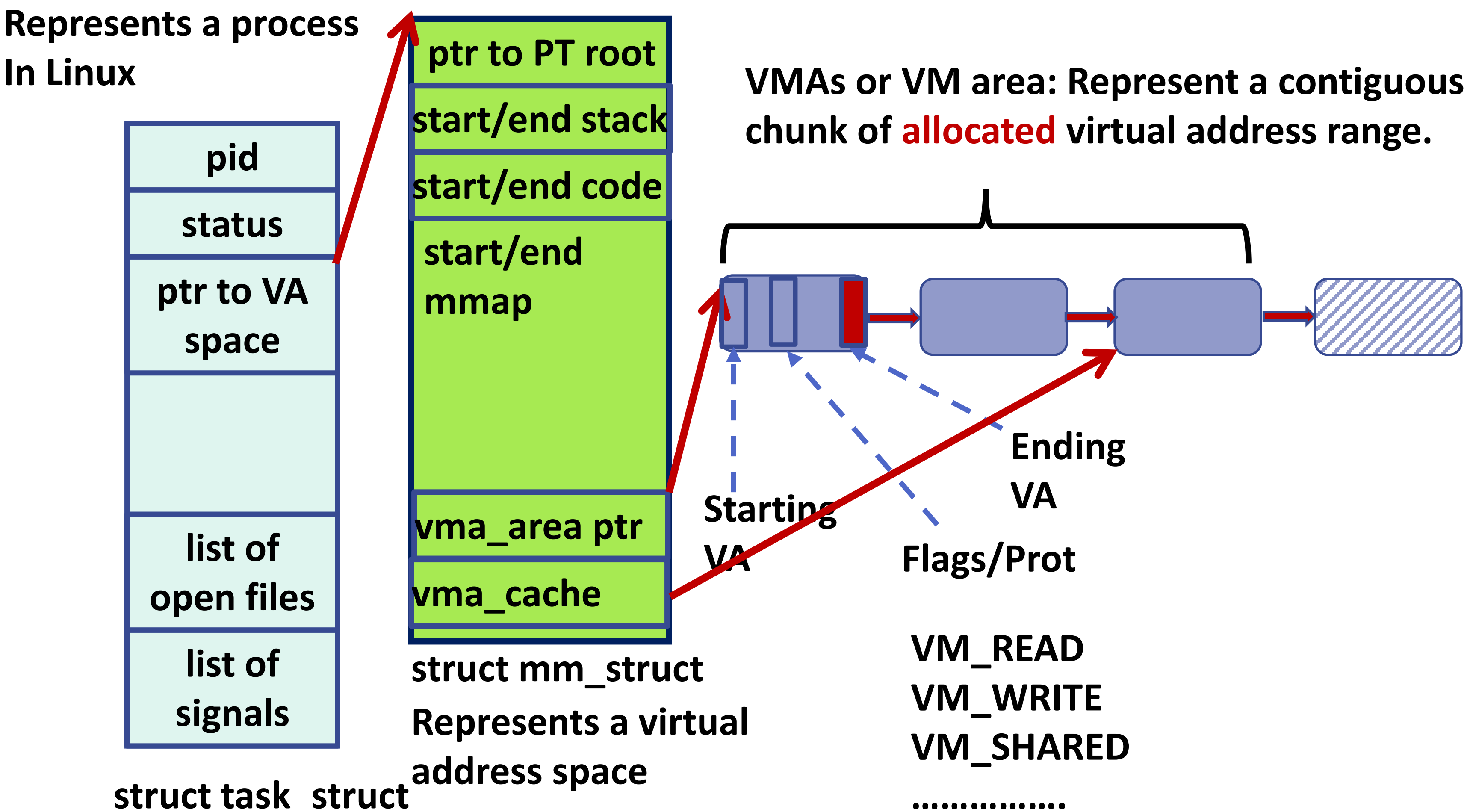
- **Heap**: Special type of dynamically allocated contiguous memory region that grows in upwards

- System calls in Linux to extend heap
  - int sbrk (increment _bytes)

Extending heap via *sbrk*



Extending heap via *sbrk*



Demand paging

- Note when “memory” (a.k.a, VA) is allocated **no** physical memory is allocated (default case)

Demand paging

- Note when "memory" (a.k.a, VA) is allocated **no** physical memory is allocated (default case)
- Why? Virtual address is in abundance; but physical memory is scarce resource.
- Allocate physical memory for when the virtual **address is accessed first time**

Demand paging

- Note when “memory” (a.k.a, VA) is allocated **no** physical memory is allocated (default case)
- Why? Virtual address is in abundance; but physical memory is scarce resource.
- Allocate physical memory for when the virtual **address is accessed first time**
- Lazily allocating physical memory is called *demand paging*
- Advantage: Commits physical memory only if used

Page fault in demand paging

- **Page fault:** When h/w page walker fails to find desired PTE the processor generates a general protection fault to the OS

Page fault in demand paging

- **Page fault:** When h/w page walker fails to find desired PTE the processor generates a general protection fault to the OS
- First access to an allocated memory generates a page fault

Page fault in demand paging

- **Page fault:** When h/w page walker fails to find desired PTE the processor generates a general protection fault to the OS
- First access to an allocated memory generates a page fault
- Page fault can also happen due to insufficient permission (e.g., write access to read-only page) → protection fault

Servicing a page fault

- Page fault routine is implemented inside the OS
- Argument routine are faulting VA and type of access (e.g., read or write)

Servicing a page fault

- Page fault routine is implemented inside the OS
- Argument routine are faulting VA and type of access (e.g., read or write)
- Steps of handling a page fault

Servicing a page fault

- Page fault routine is implemented inside the OS
- Argument routine are faulting VA and type of access (e.g., read or write)
- Steps of handling a page fault
  - Check VMA structures if the VA is allocated
  - If not allocated or insufficient permission, raise segmentation fault to application
  - If allocated with correct permission find a physical page frame to map the page containing the faulting VA
  - Update page table to note new VA->PA mapping
  - Return

Servicing a page fault

- Page fault routine is implemented inside the OS
- Argument routine are faulting VA and type of access (e.g., read or write)
- Steps of handling a page fault
  - Check VMA structures if the VA is allocated
  - If not allocated or insufficient permission, raise segmentation fault to application
  - If allocated with correct permission find a physical page frame to map the page containing the faulting VA
  - Update page table to note new VA->PA mapping
  - Return
- The faulting instruction retries after page fault returns

Servicing a page fault

- Page fault routine is implemented inside the OS
- Argument routine are faulting VA and type of access (e.g., read or write)
- Steps of handling a page fault
  - Check VMA structures if the VA is allocated
  - If not allocated or insufficient permission, raise segmentation fault to application
  - If allocated with correct permission find a physical page frame to map the page containing the faulting VA

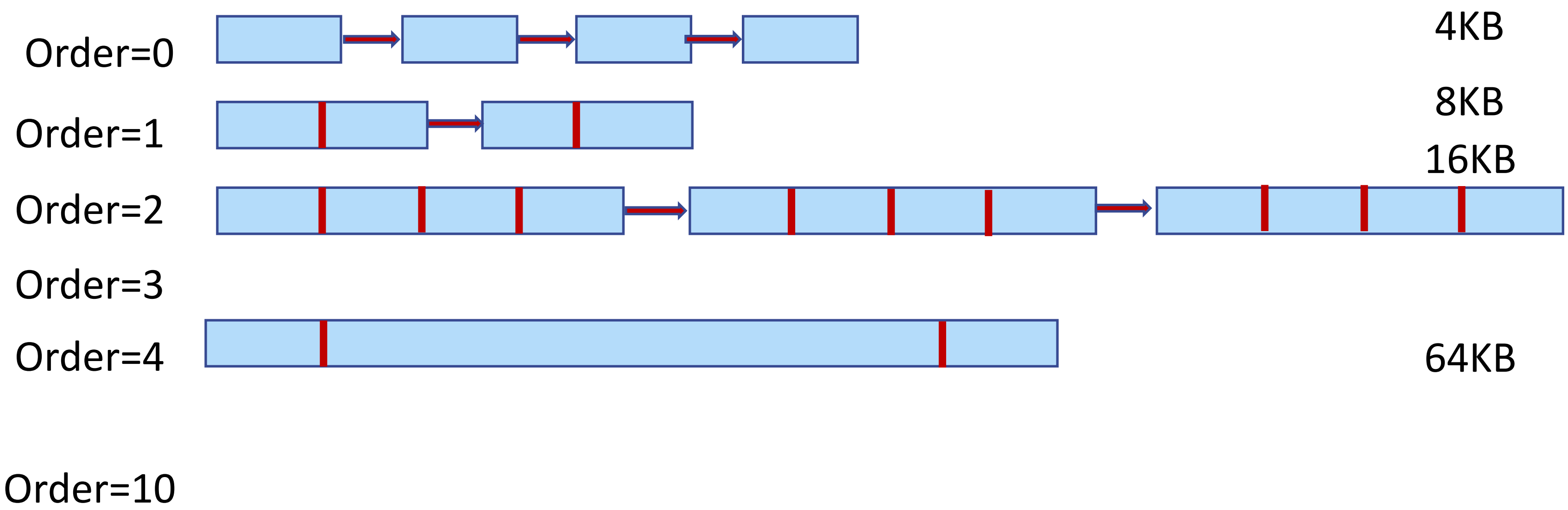
    Next
  - Update page table to note new VA->PA mapping
  - Return
- The faulting instruction retries after page fault returns

Allocating physical memory

- Buddy allocator in Linux: Goal is to keep free physical memory as contiguous as possible (why?)
- It is a list of free list of contiguous physical pages of different sizes (2^{order} x 4KB)

Allocating physical memory

- Buddy allocator in Linux: Goal is to keep free physical memory as contiguous as possible (why?)
- It is a list of free list of contiguous physical pages of different sizes (2^{order} x 4KB)



Buddy allocator operation

- Allocate from a free list which has smallest units that fits the requested allocation size
  - If no entry in the smallest list, go to next larger list and so on…
  - Put the leftover blocks in the lower order list

Buddy allocator operation

- Allocate from a free list which has smallest units that fits the requested allocation size
  - If no entry in the smallest list, go to next larger list and so on…
  - Put the leftover blocks in the lower order list
- Merge two contiguous blocks of physical memory in a free list and add the merged block in next higher order free list

Where does *malloc()* fits in?

- *malloc()* is function call implemented in a library. It is **not** part of OS.
- malloc allocates virtual address range (like mmap())

Where does *malloc()* fits in?

- *malloc()* is function call implemented in a library. It is **not** part of OS.
- malloc allocates virtual address range (like mmap())
- Then, why malloc()/free()?
- Limitation of mmap
  - Minimum granularity of VA allocation is a page (4KB)
  - But applications often allocates in chunks less than 4KB

Where does *malloc()* fits in?

- *malloc()* is function call implemented in a library. It is **not** part of OS.
- malloc allocates virtual address range (like mmap())
- Then, why malloc()/free()?
- Limitation of mmap
  - Minimum granularity of VA allocation is a page (4KB)
  - But applications often allocates in chunks less than 4KB
- malloc() maintains free list of small allocations
- If free memory is available in malloc()'s free list, no need to got to the OS
- malloc() with large size (e.g., >32KB) converts to mmap

Where does *malloc() fits in* ?

- Two key goals of a malloc library:
  - Reduce memory bloat, i.e., reduce any additional allocated memory than what application asked
  - Reduce the number of system calls to OS (e.g., mmap())
  - System calls are slow

Where does *malloc() fits in* ?

- Two key goals of a malloc library:
  - Reduce memory bloat, i.e., reduce any additional allocated memory than what application asked
  - Reduce the number of system calls to OS (e.g., mmap())
  - System calls are slow
- Many approaches to create malloc library
  - Best fit, first fit, worst fit
- Many malloc() libraries
  - glibc-malloc, tc-malloc, dl-malloc
- You can write your own malloc() library !

The overall picture of memory management

Application

The overall picture of memory management

malloc()

Application

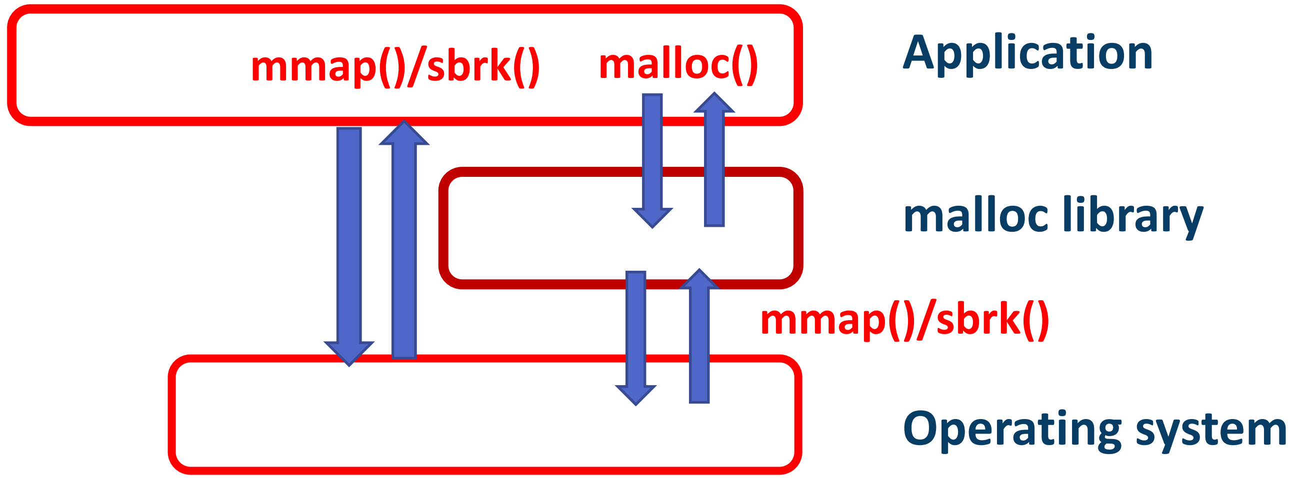
Small VA allocation

malloc library

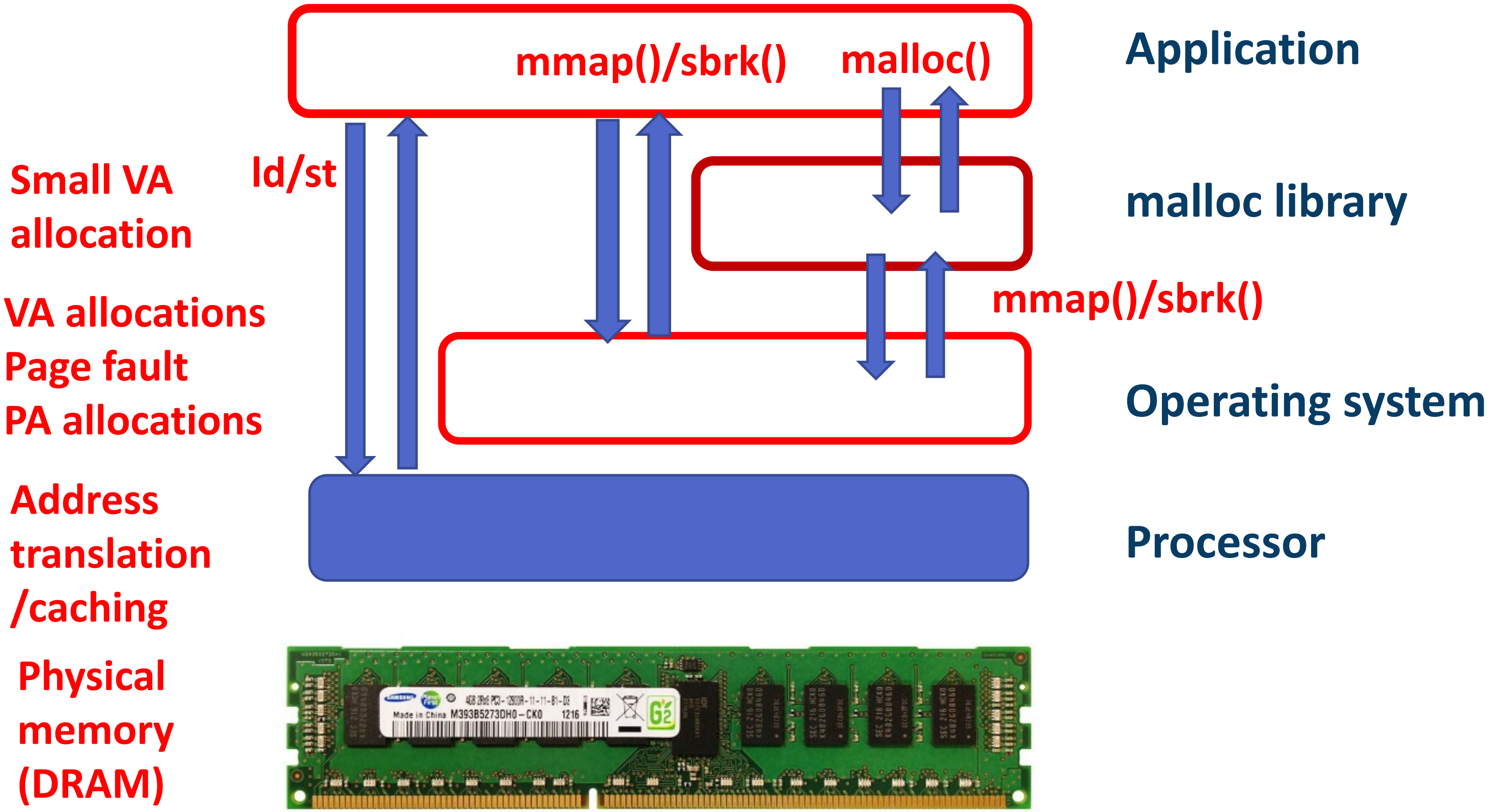
The overall picture of memory management

Small VA allocation

VA allocations
Page fault
PA allocations



The overall picture of memory management



Putting everything together

1. Application requests VA allocation via mmap()
2. OS creates/extends VMA regions to allocate VA range
3. OS returns the starting VA of just-allocated VA range

Putting everything together

1. Application requests VA allocation via mmap()
2. OS creates/extends VMA regions to allocate VA range
3. OS returns the starting VA of just-allocated VA range
4. Application performs load/store on the address in the VA range
5. H/W looks up TLB; misses (Why?)
6. H/W page table walker walks the page table

Putting everything together

1. Application requests VA allocation via mmap()
2. OS creates/extends VMA regions to allocate VA range
3. OS returns the starting VA of just-allocated VA range
4. Application performs load/store on the address in the VA range
5. H/W looks up TLB; misses (Why?)
6. H/W page table walker walks the page table
7. Desired entry not found on PTE (why?)
8. H/W generates a page fault to OS

Putting everything together

9. OS’a page fault handler checks VMA regions to ensure it’s a legal access
10. Page fault handler finds a free physical page frame to map the VA page from the buddy allocator
11. Updates page table to note new VA to PA mapping and return

Putting everything together

9. OS’a page fault handler checks VMA regions to ensure it’s a legal access
10. Page fault handler finds a free physical page frame to map the VA page from the buddy allocator
11. Updates page table to note new VA to PA mapping and return
12. Application retries the same instruction
13. This time page table walker finds it in page table and loads it into TLB
14. Next time if same page is accessed it may hit in TLB (→ no page walk)

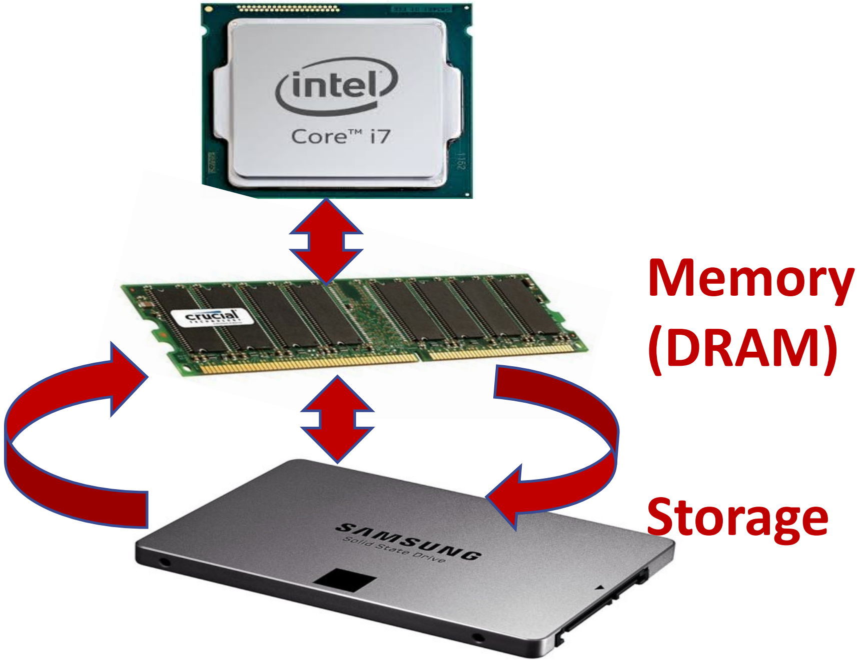
Miscellaneous related topics

Swapping

- **Goal**: Provide an illusion of larger memory than actually available

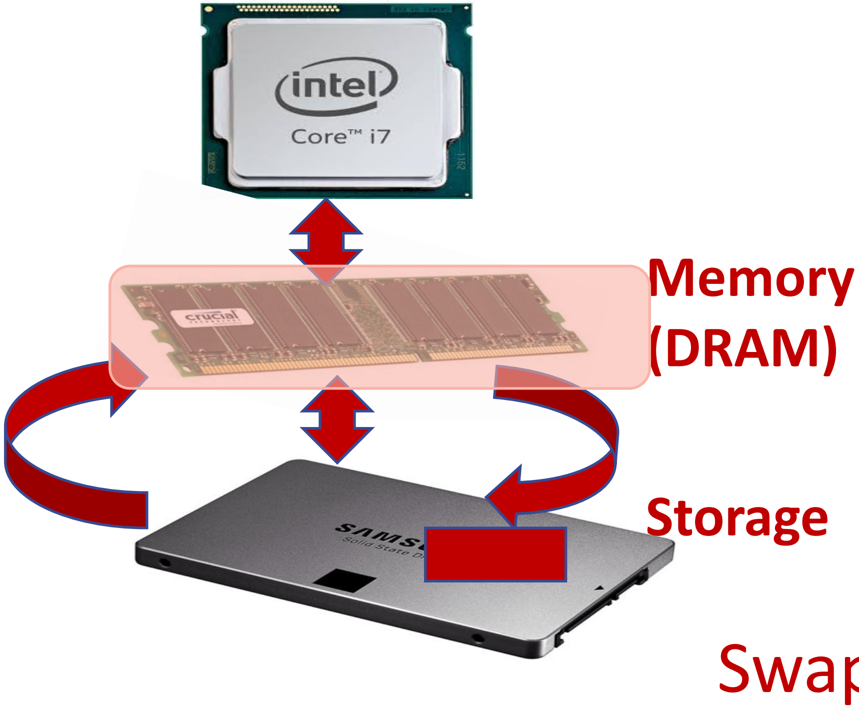
Swapping

- **Goal**: Provide an illusion of larger memory than actually available



Swapping

- **Goal**: Provide an illusion of larger memory than actually available



How swapping work?

- OS attempts to figure out which pages in memory are not actively used
  - Use access bit ("A" bit) in the PTE
  - OS periodically unsets "A" bits of pages in memory
  - After a little while, OS checks H/W has set "A" bit of those pages

How swapping work?

- OS attempts to figure out which pages in memory are not actively used
  - Use access bit (“A” bit) in the PTE
  - OS periodically unsets “A” bits of pages in memory
  - After a little while, OS checks H/W has set “A” bit of those pages
  - If “A” bit set for a page → actively used page
  - If “A” bit is unset for a page → not actively used → candidate for swapped out to storage

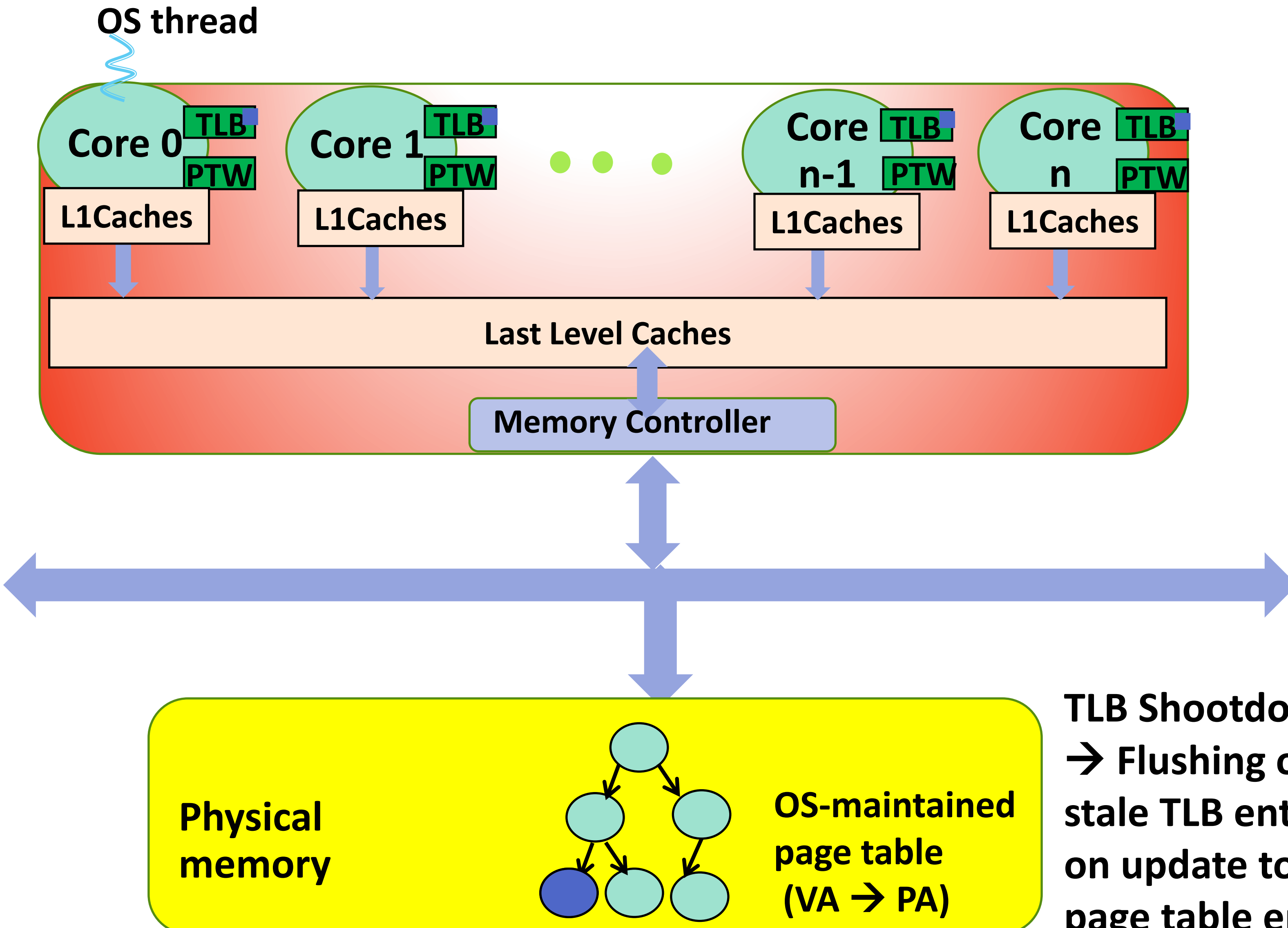
How swapping work?

- OS attempts to figure out which pages in memory are not actively used
  - Use access bit ("A" bit) in the PTE
  - OS periodically unsets "A" bits of pages in memory
  - After a little while, OS checks H/W has set "A" bit of those pages
  - If "A" bit set for a page → actively used page
  - If "A" bit is unset for a page → not actively used → candidate for swapped out to storage
- OS writes back data of the page to swap file
- Update PTE to unset present ("p") bit

How swapping work?

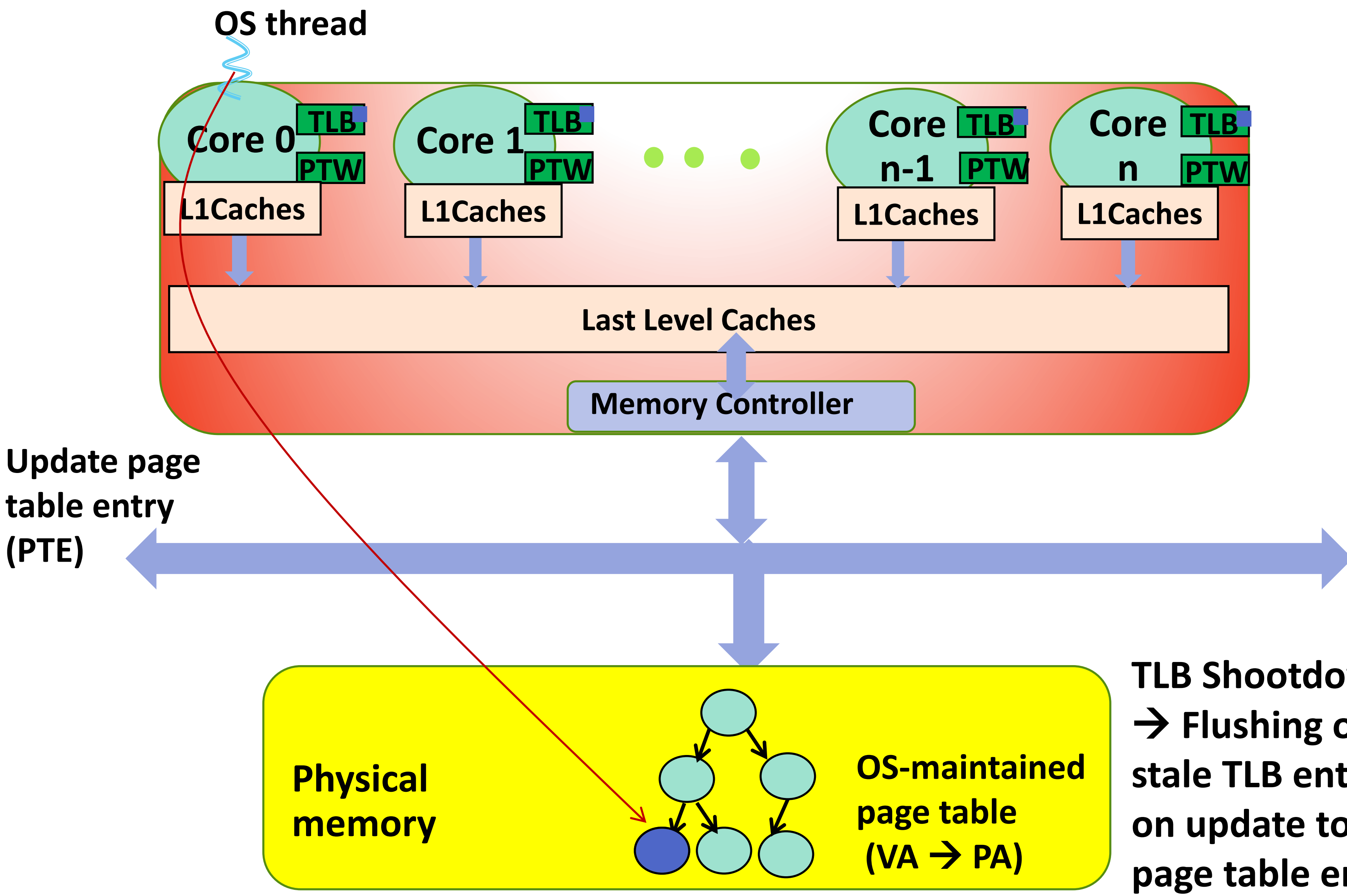
- There will be page fault if an application accesses a page that is swapped out
  - Because “p” bit is unset
- OS page fault handler figures out the page had been swapped out
- Page fault handler brings the page into memory
- Application retries the faulting instruction.

TLB shootdown (invalidation)



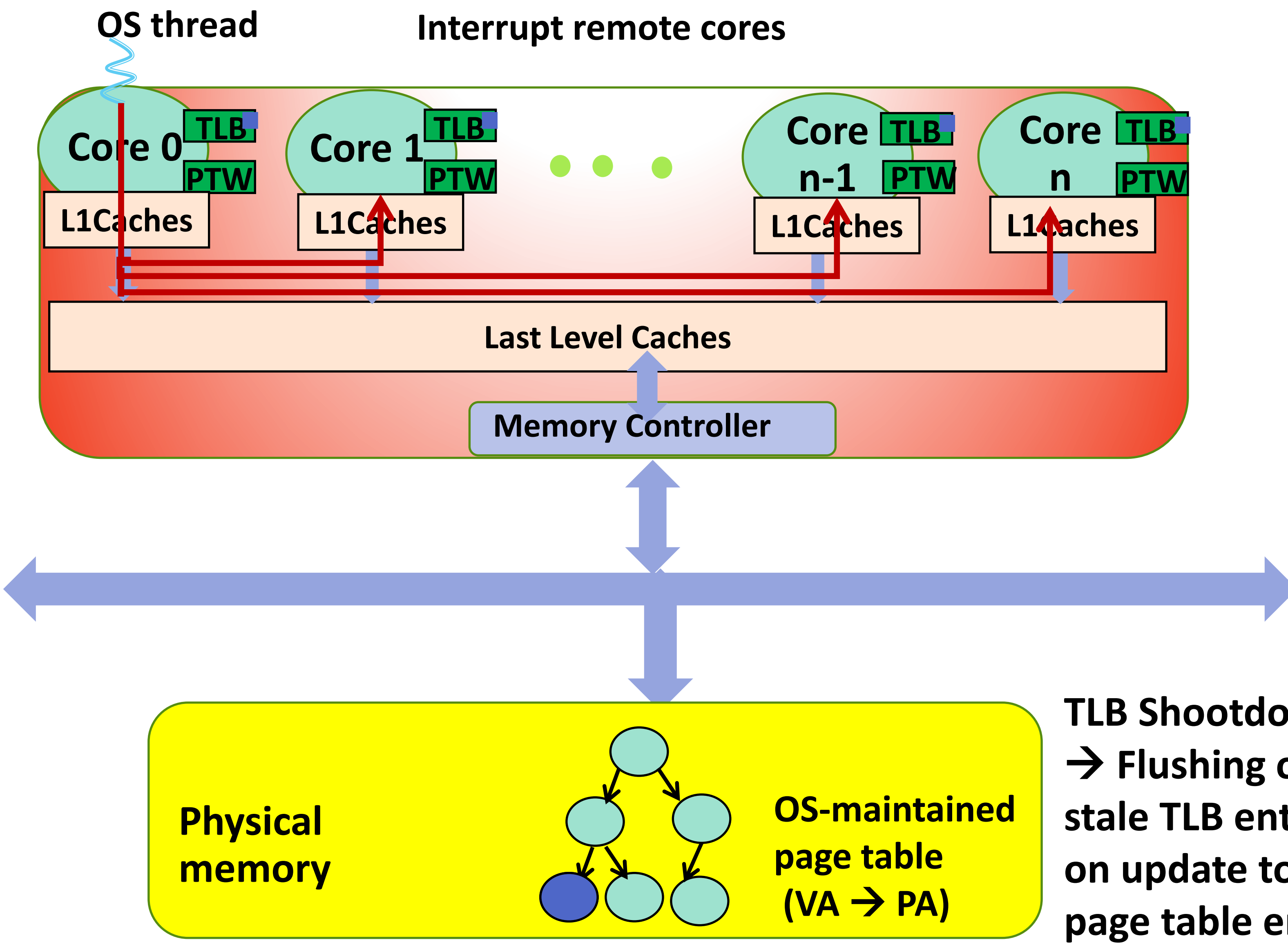
TLB Shootdown → Flushing of stale TLB entries on update to page table entry.

TLB shootdown (invalidation)



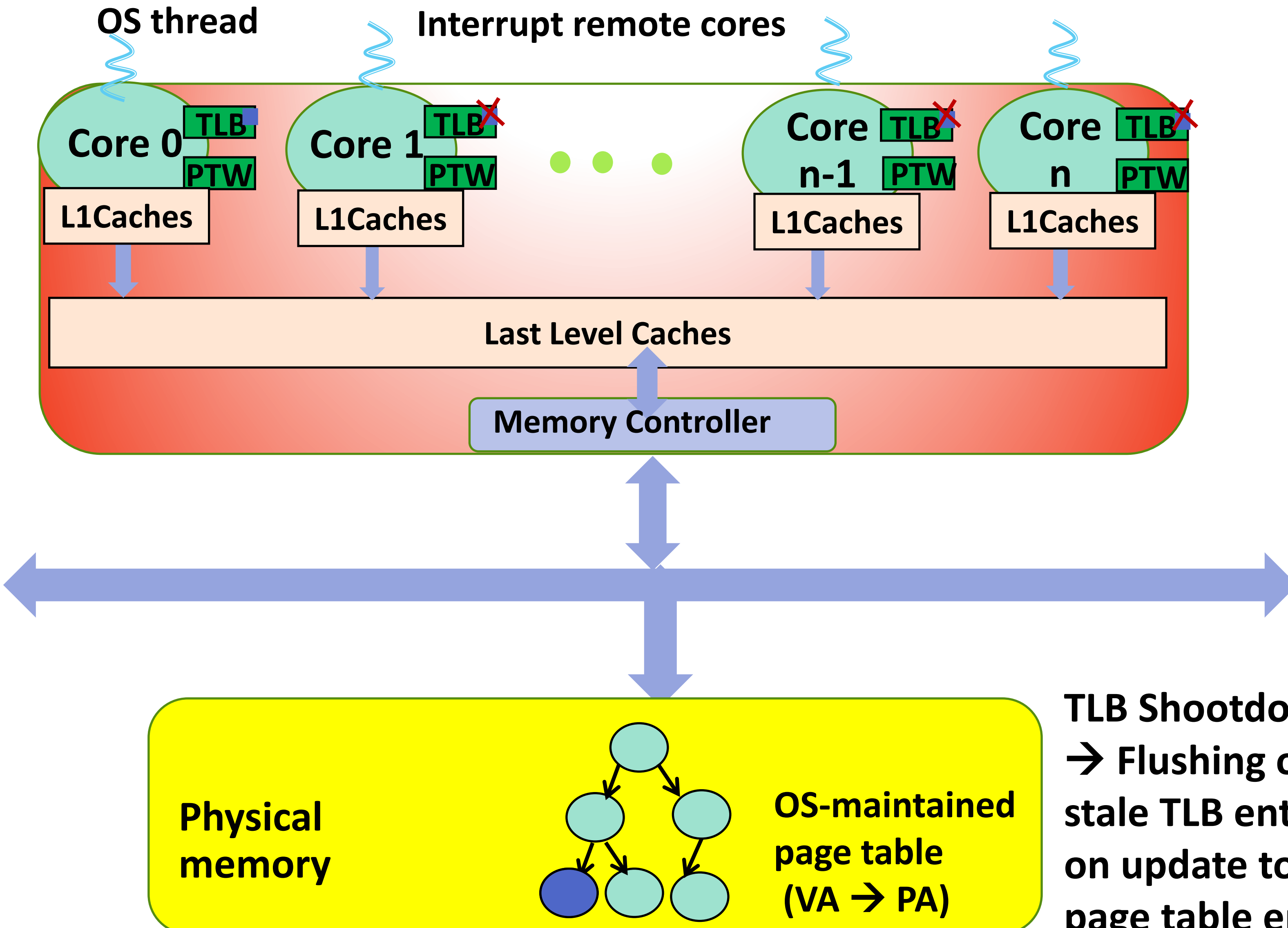
TLB Shootdown → Flushing of stale TLB entries on update to page table entry.

TLB shootdown (invalidation)



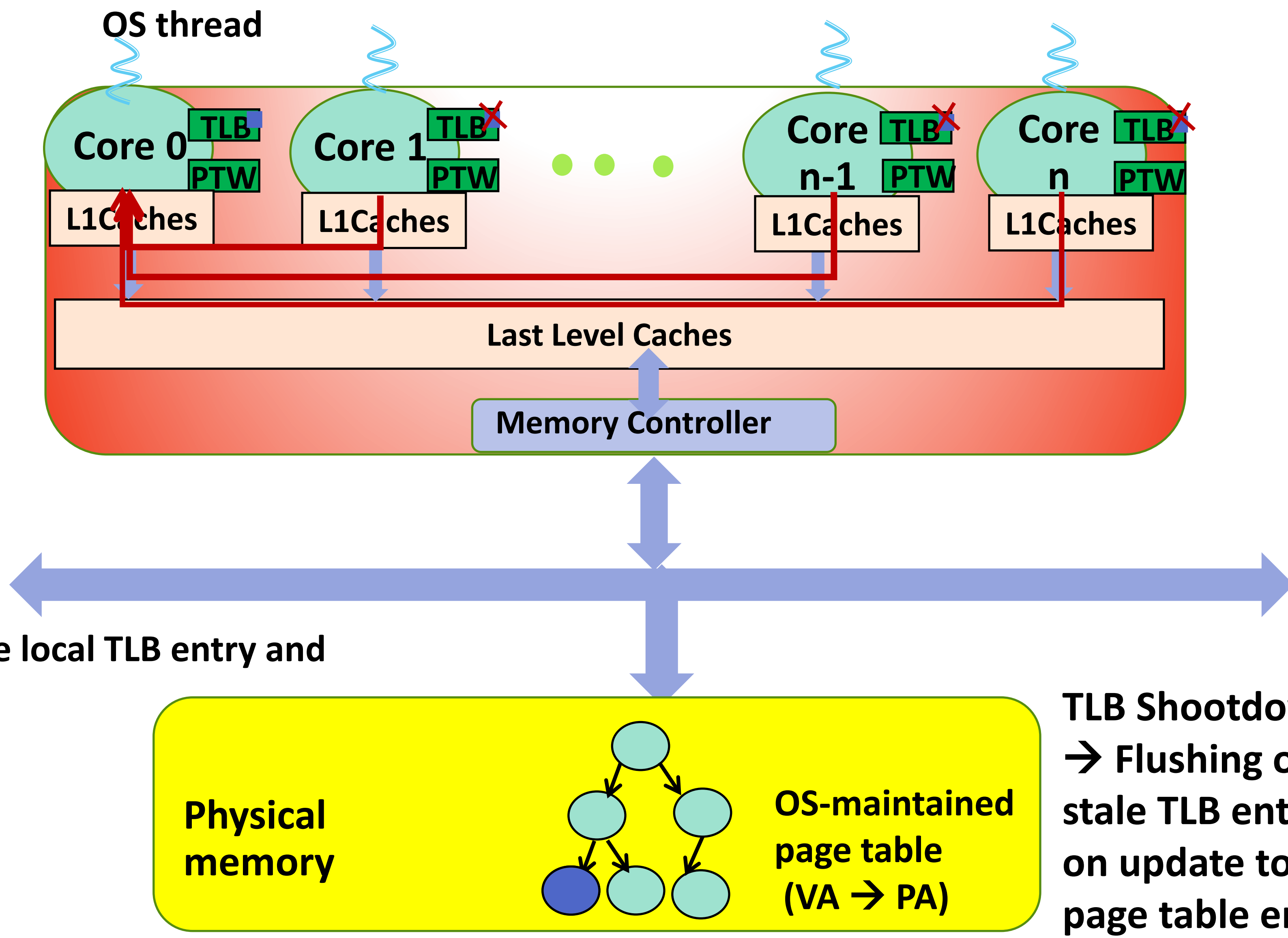
TLB Shootdown → Flushing of stale TLB entries on update to page table entry.

TLB shootdown (invalidation)



TLB Shootdown → Flushing of stale TLB entries on update to page table entry.

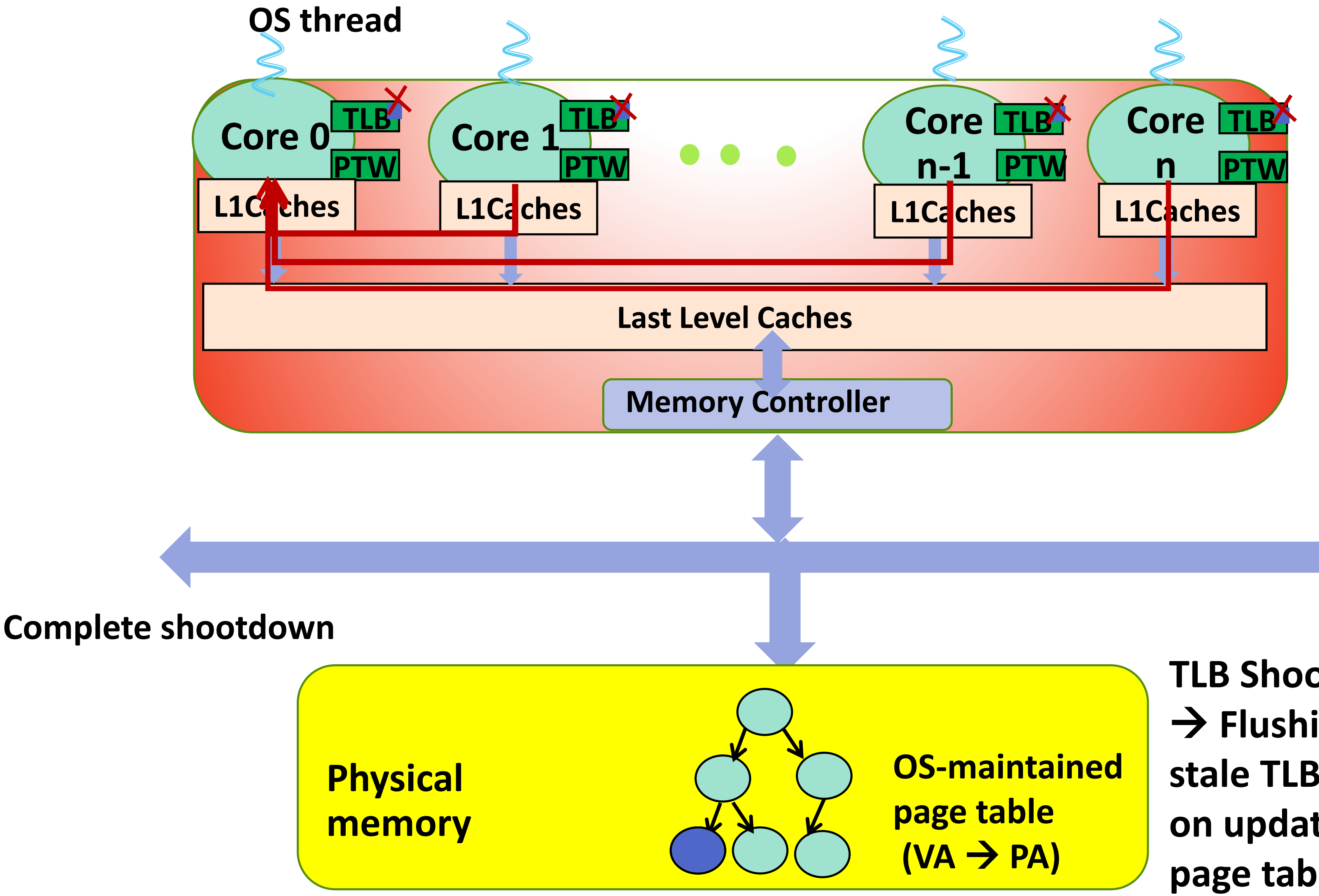
TLB shootdown (invalidation)



Invalidate local TLB entry and send Ack

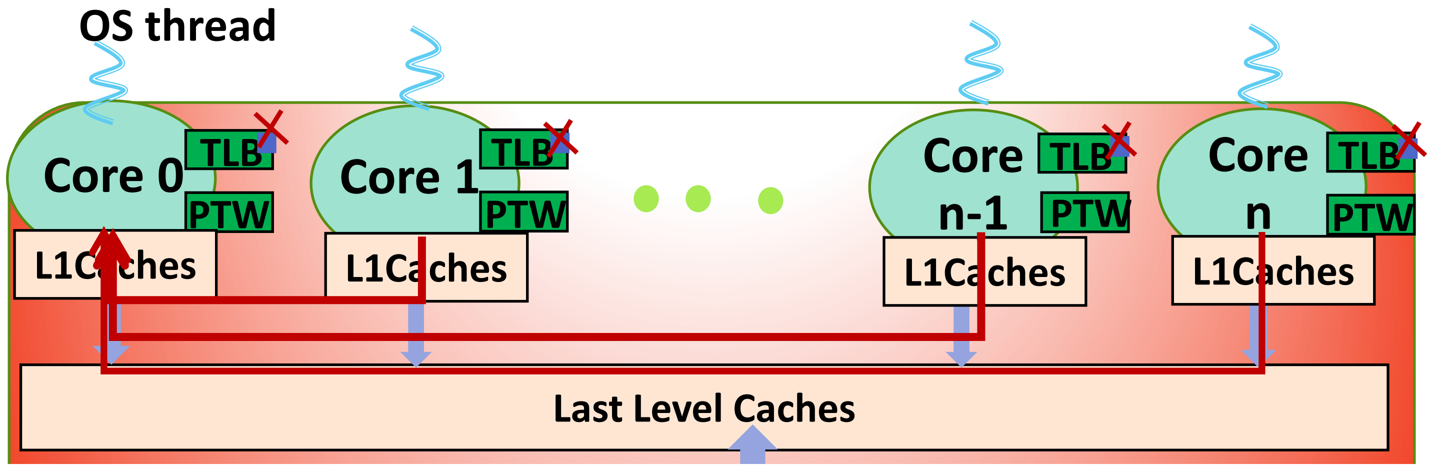
TLB Shootdown → Flushing of stale TLB entries on update to page table entry.

TLB shootdown (invalidation)

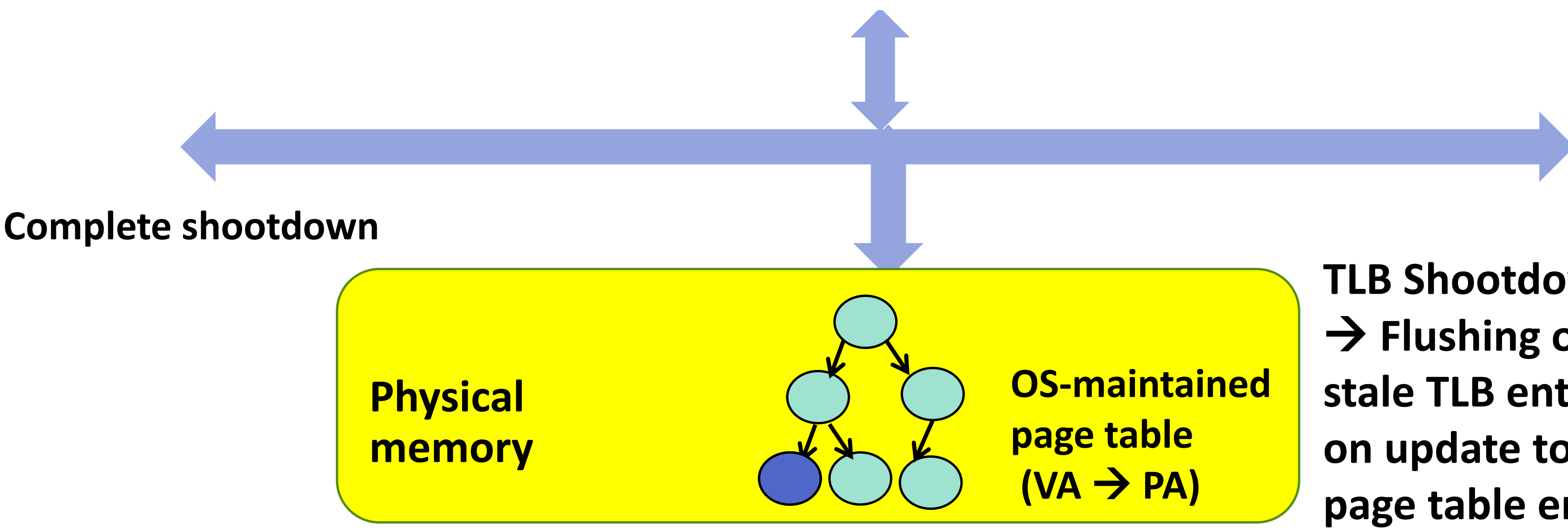


TLB Shootdown → Flushing of stale TLB entries on update to page table entry.

TLB shootdown (invalidation)



A shootdown on a x86-64 CPU could take 10s of micro-seconds



TLB Shootdown → Flushing of stale TLB entries on update to page table entry.