Virtual Memory: Address Translation

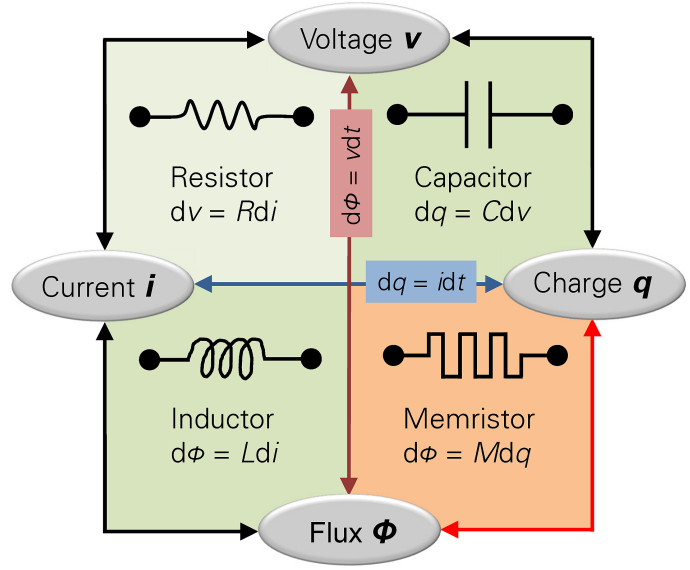
Memory management

Issues

- Ideal main memory
  - Very fast
  - Very large
  - Non-volatile
- In practice
  - Not so fast *--need caches*
  - Not large and volatile *--need persistent mass-storage*

Future

- In the future, it might be!



Past and present

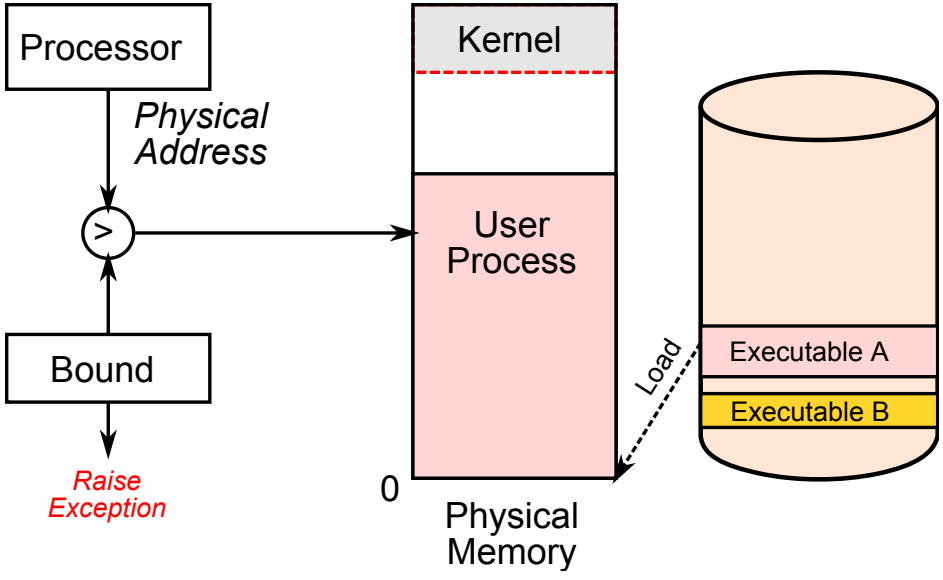
- Use of a memory hierarchy
  - Small amount of fast, expensive, transparent memory cache
  - Some medium-speed, medium-price, addressable main memory
  - Lots of slow, cheap, second-level persistent storage
- Need for a memory manager to handle this hierarchy at best

Uniprogramming

Concept

- One process executes at a time
  - Loaded at physical address 0, contiguously in memory
  - Uses physical addresses directly
- OS gets a fixed part of memory
  - Maximum process address = Memory size - OS size
  - OS protected from process by address checking

Example



Issues

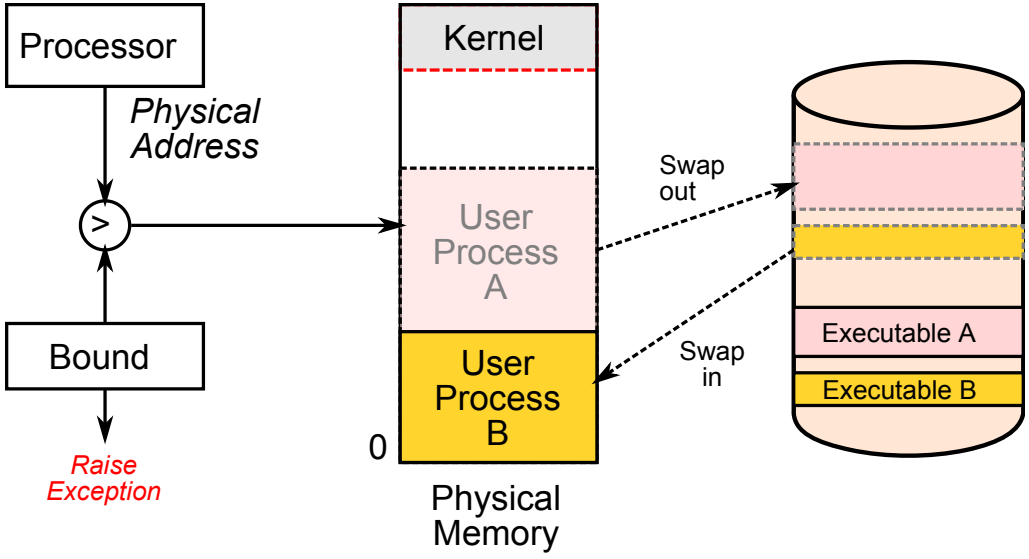
- Only one process at a time
  - Degraded usability
  - Waste of CPU when process blocked on I/O accesses
  - Waste of memory if process smaller than memory
- Need for *multiprogramming*!

Multiprogramming

Swapping

- One process at a time
- But use swapping when blocked
  - Temporarily swap out entire blocked process to disk
  - Swap in another entire process from disk to run

Example



Issues

- Swapping is slow if processes are large
  - Performance limited by storage device's speed
- Not *true* multiprogramming

Multiprogramming

Requirements for true multiprogramming

Transparency

- Need for multiple processes to coexist in memory
- Processes not aware that memory is shared
- Processes not care which physical portion of memory they get

Safety

- Processes not able to corrupt each other
- Processes not able to corrupt the kernel

Efficiency

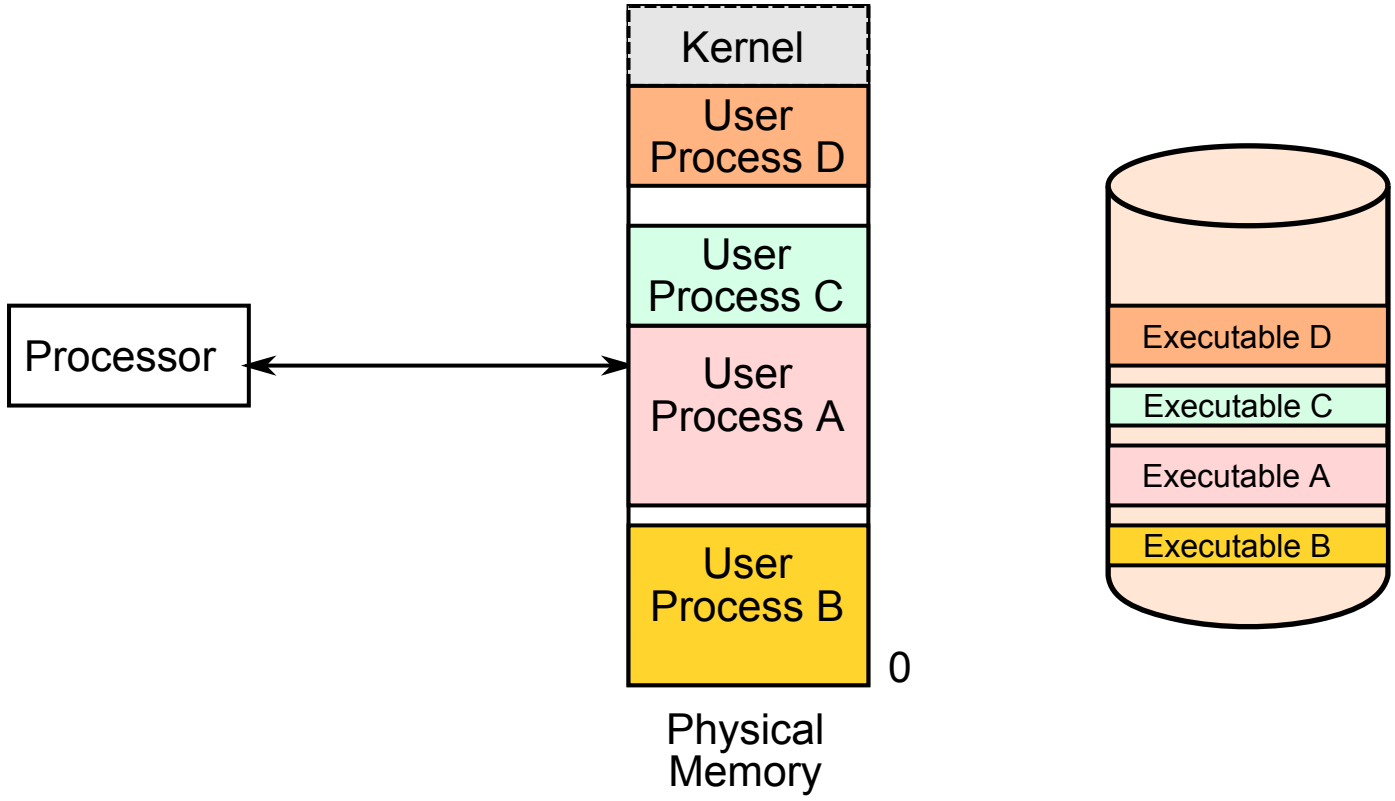
- Performance of CPU and memory not degraded badly due to sharing

Flexibility

- Many processes whose combined size may exceed available memory
- One process may exceed available memory
- Processes might need to share some portions of memory

Multiprogramming

Illustration



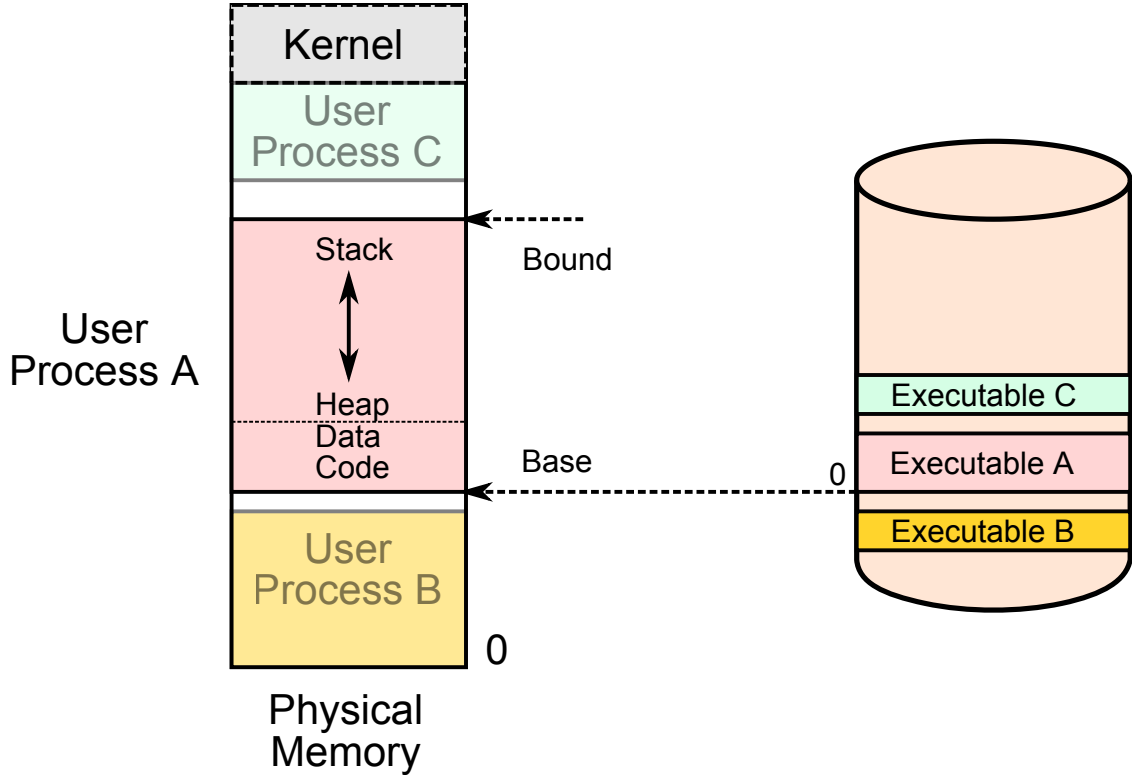
Main requirements

- Relocation
  - Ability to load and run a process anywhere in memory
- Protection
  - Protection between processes

Contiguous memory allocation

Idea

- At compile/link time, program set to start at 0
- Process' smallest physical address is *base* address (aka *relocation address*)
- Largest address process can access is *bound* address

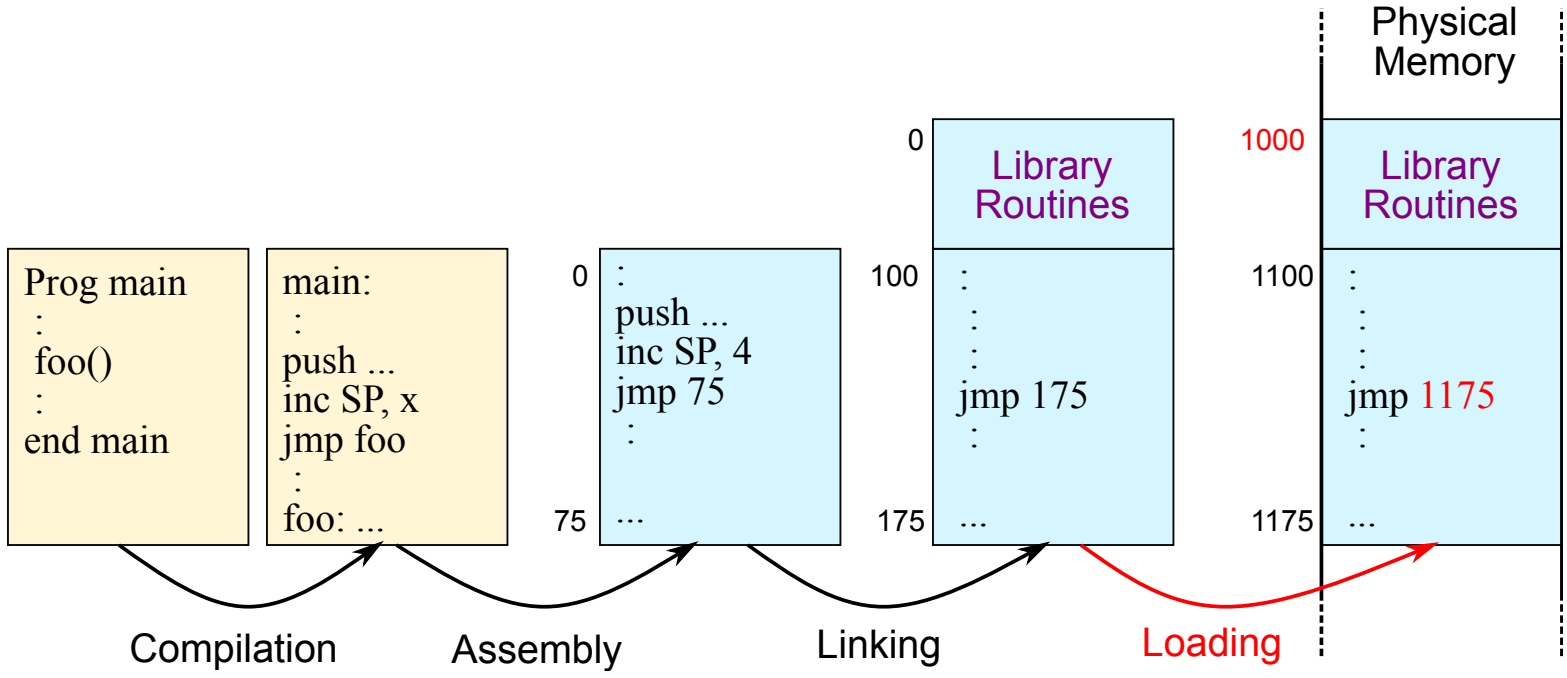


- Two approaches: static vs dynamic

Contiguous memory allocation

Static approach

- OS adjusts process' addresses *at load time*, based on its location in memory



Issues

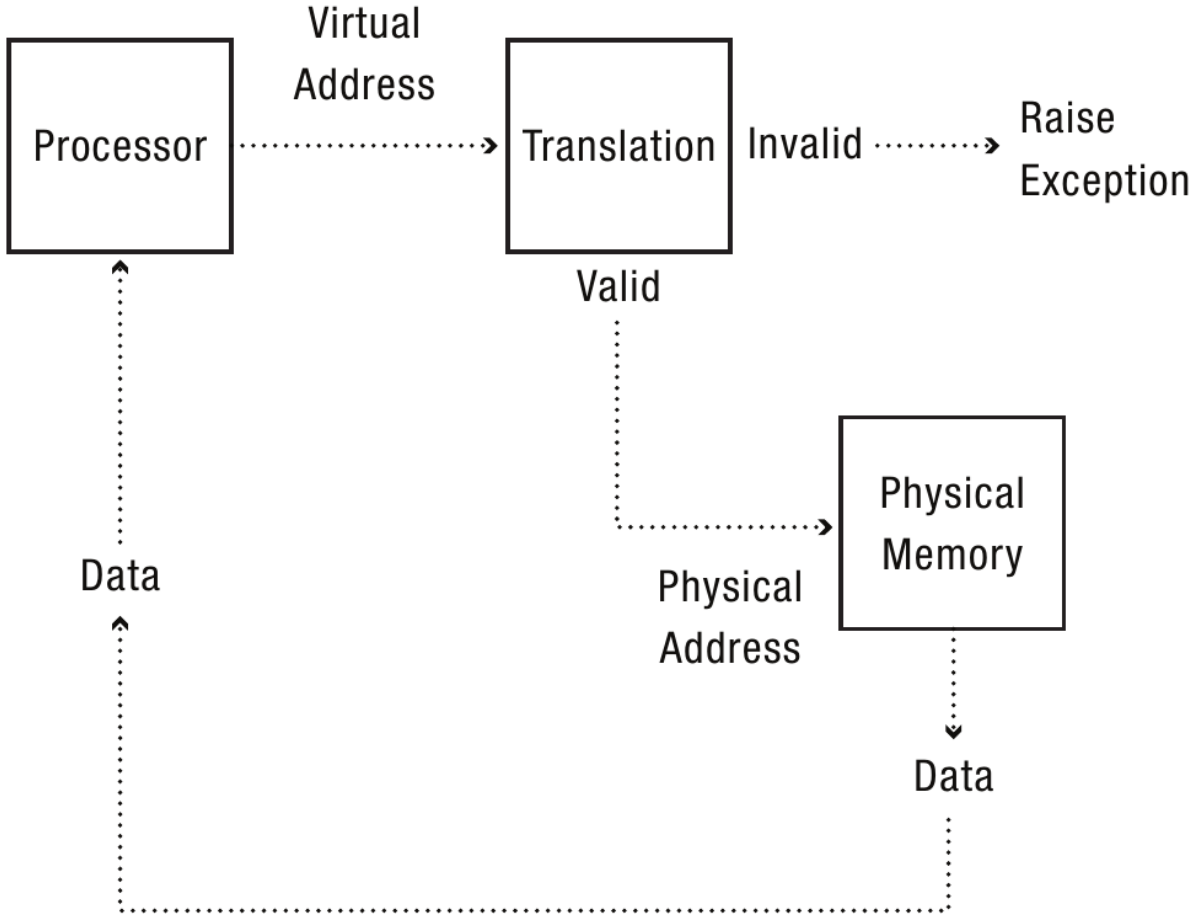
- Initial relocation only
- No flexibility (impossible to relocate again)
- No actual protection

Contiguous memory allocation

Dynamic approach

Idea

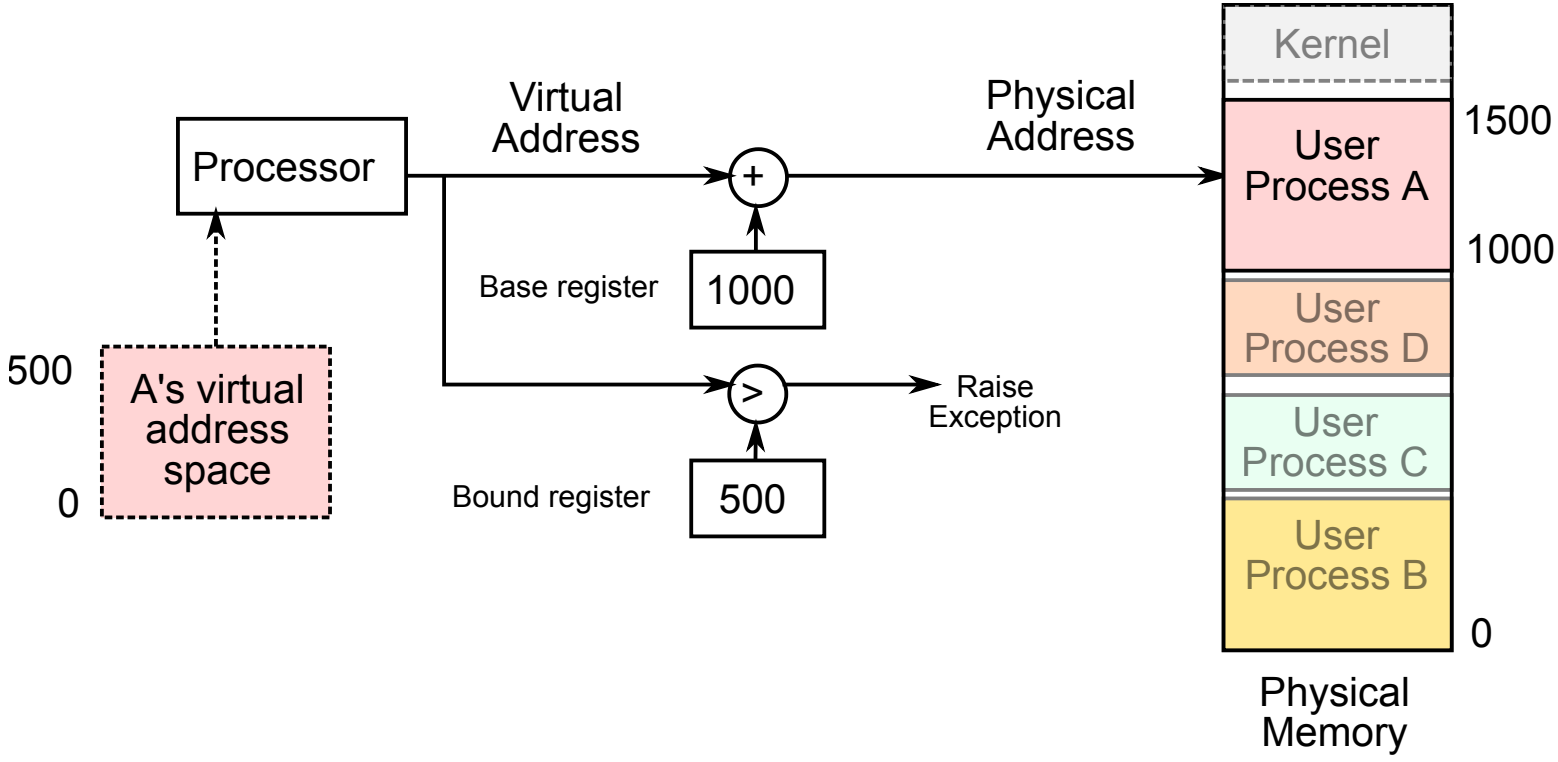
- Relocation and protection performed at runtime, using hardware support



Contiguous memory allocation

Implementation

- *Base register* to perform runtime relocation
- *Bound register* to perform runtime protection



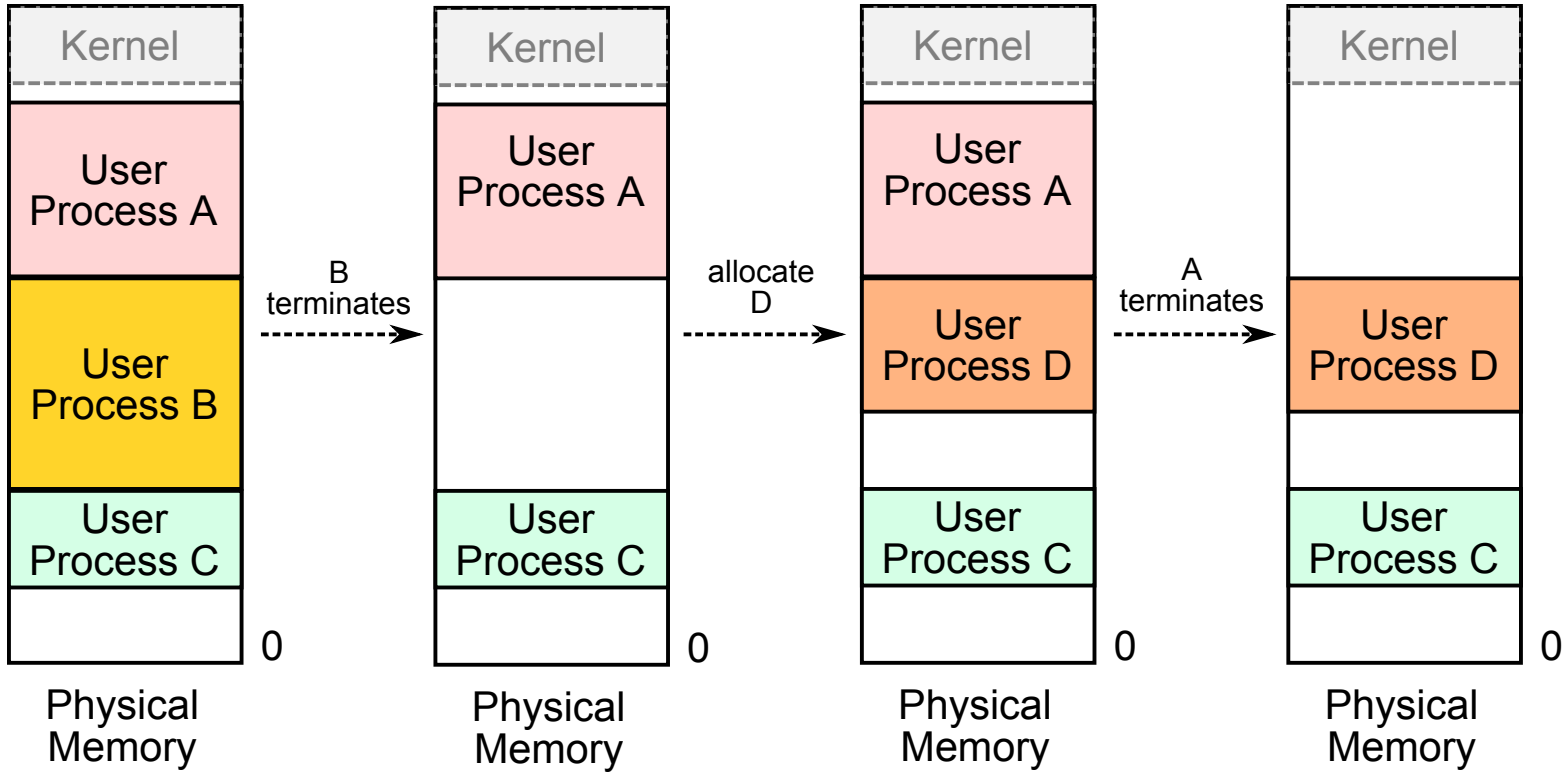
Pros/cons

- Dynamic relocation and some protection
- Need contiguous memory space

Contiguous memory allocation

Memory allocation

- Typical strategies: first-fit, best-fit, worst-fit



Issues

- External fragmentation

Contiguous memory allocation

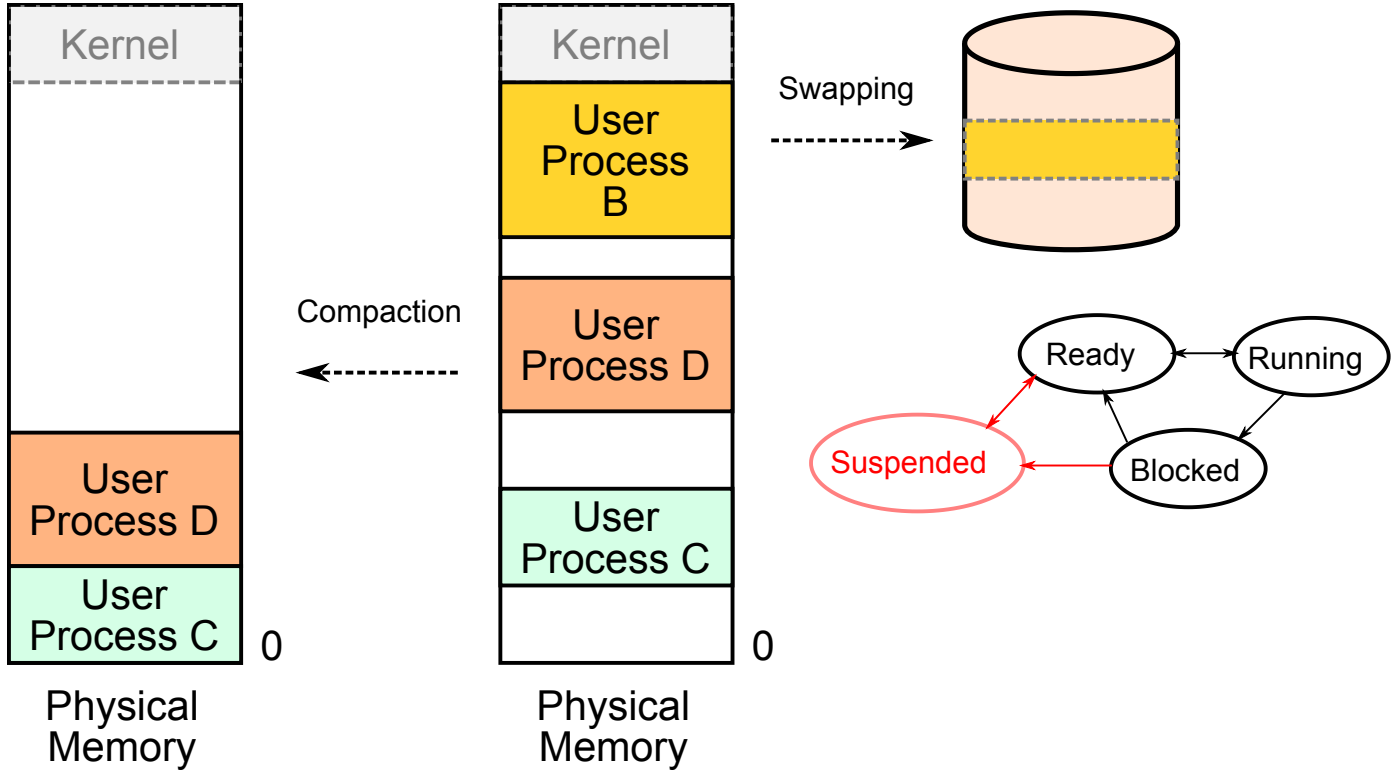
Improving memory allocation

Compaction

- Relocate processes to merge free space
- Prevents external fragmentation

Swapping

- Preempt processes and reclaim their memory
- Allow for more processes than memory size



Contiguous memory allocation

Conclusion (dynamic approach)

Advantages

- Processes can be relocated during execution
- Protection of each process' memory segment
- Simple hardware
  - Two registers
  - Two comparison operators

Drawbacks

- External fragmentation
- Internal fragmentation
- Sharing is near impossible
- Can't grow stack/heap as needed
- Low degree of multiprogramming
  - All *active* processes must entirely fit in memory
  - Each process is limited to physical memory size
- Hardware slowdown due to operations on every memory reference
- Permissions are the same for the entire address space
- Etc.

ECS 150 - Virtual Memory: Address Translation

Prof. Joël Porquet-Lupine

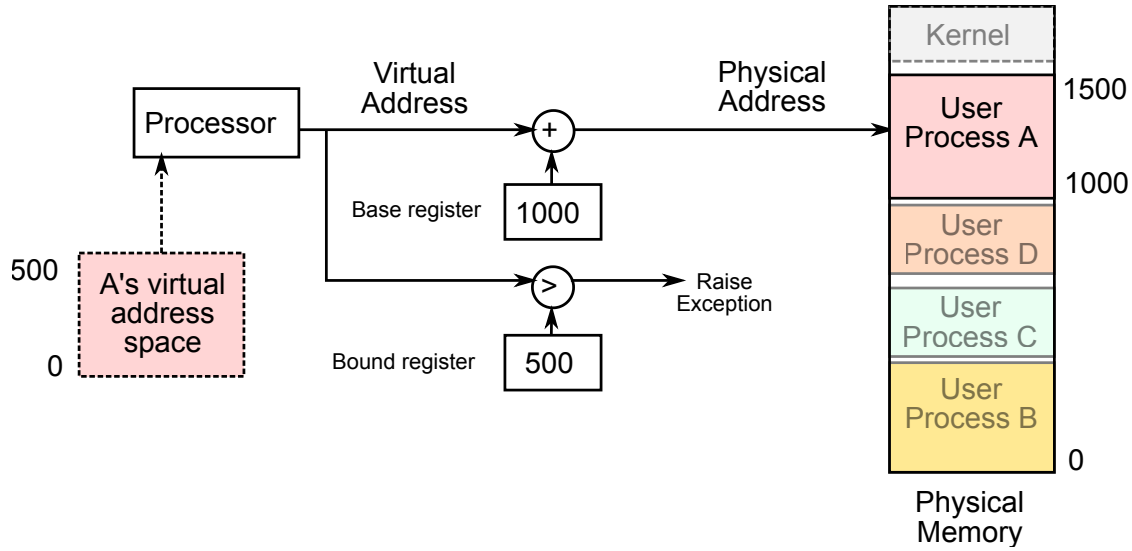
UC Davis - 2020/2021

Recap

Memory management

- Main memory is not fast, not large, and volatile
- Need memory management strategy
  - Use of slow but persistent second-level mass-storage

Contiguous memory allocation



- Dynamic relocation
- Memory protection
- Simple hardware

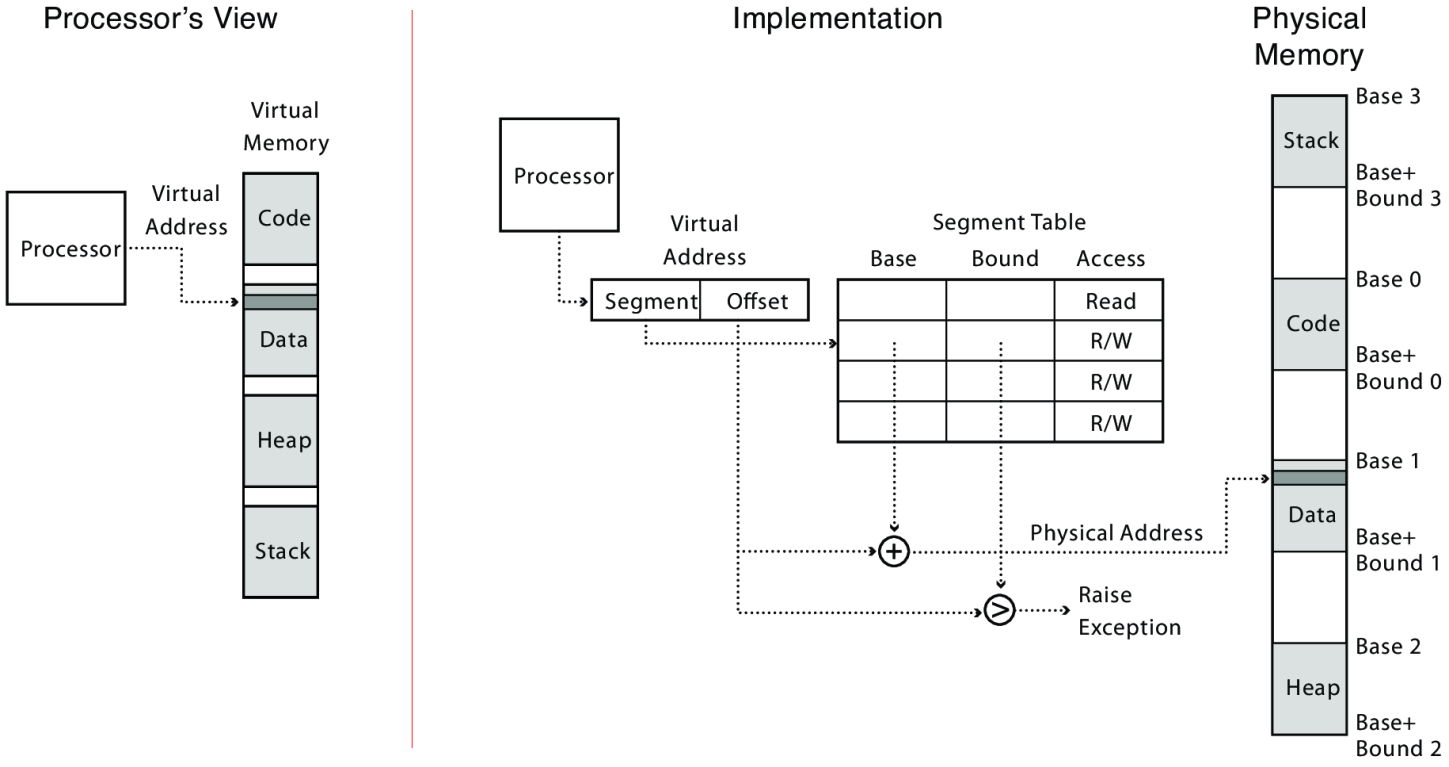
- External/internal fragmentation
- No memory sharing
- Fixed stack and heap sizes
- Same permissions for whole segment

Memory segmentation

Idea

- Single pair of base and bound per process is too limiting
- Instead, support an array of them
- Higher bits of the virtual address are used to index the array

Example

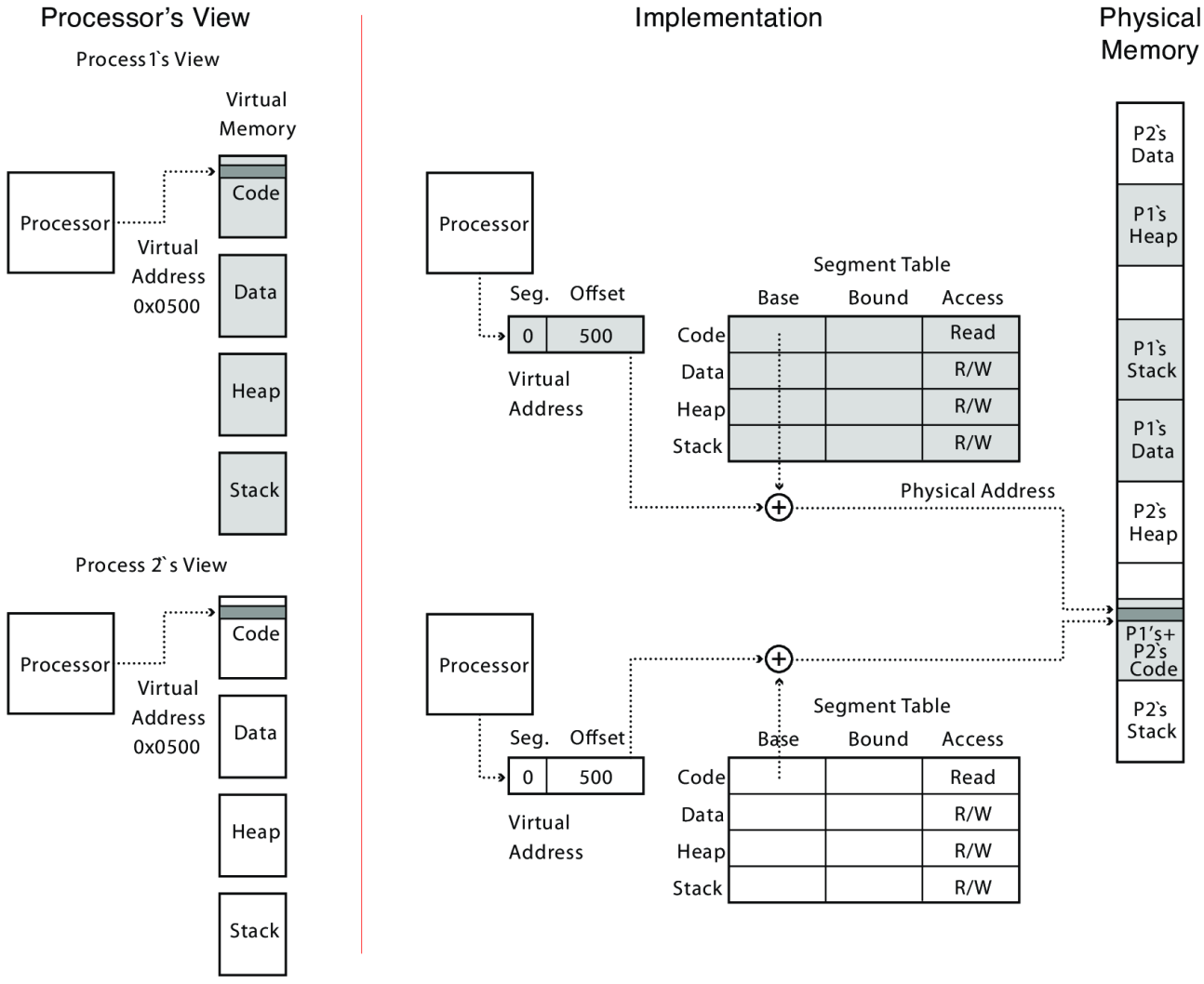


- Accesses outside of defined segments are *segmentation faults*!

Memory segmentation

Segment sharing

- Processes can share entire segments
  - E.g., a child process after forking



Memory segmentation

Conclusion

Advantages

- Segment sharing
- Segment permissions
  - E.g., protect code segment from being overwritten
- Extensible stack/heap segments

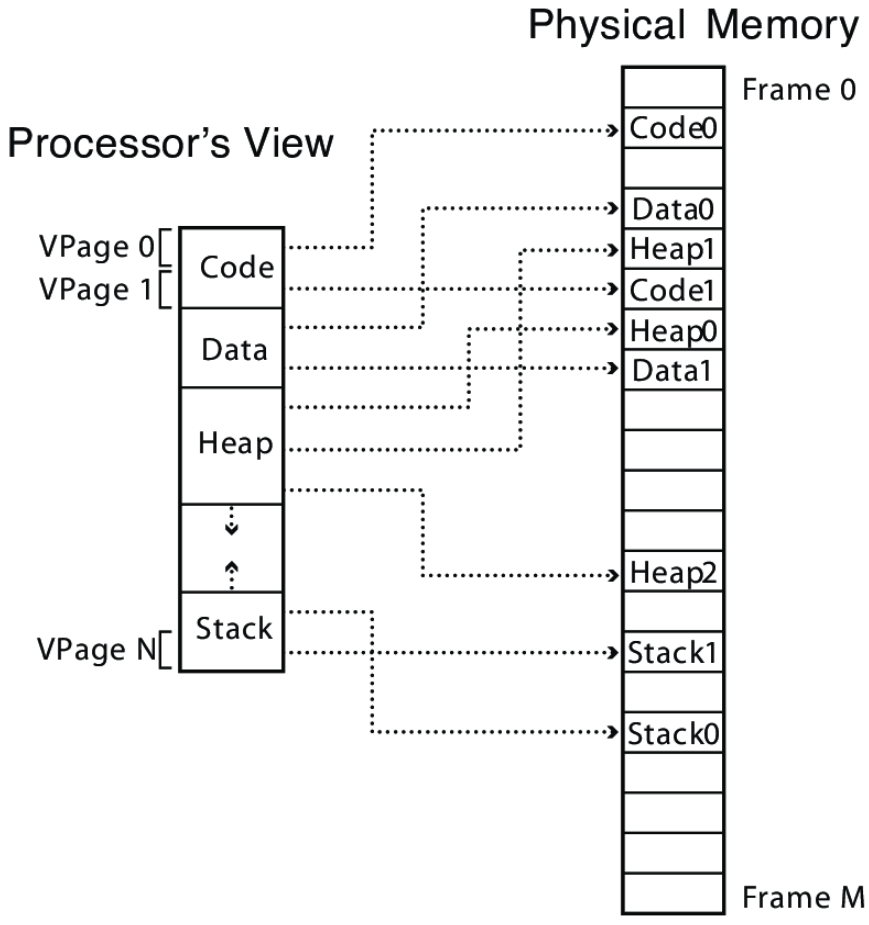
Drawbacks

- Segments need to be contiguously allocated
- External fragmentation
  - Possible use of swapping or compaction
- Limited number of segments

Paged memory

Idea

- Memory is divided into fixed-size chunks called pages
  - **Virtual pages** in the virtual address space
  - **Page frames** in the physical memory



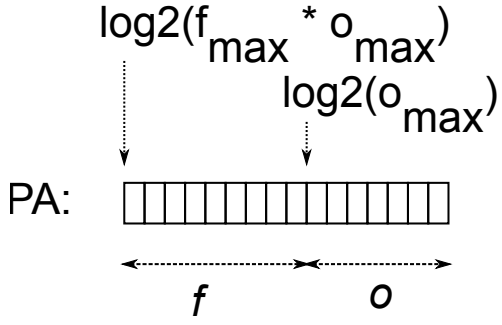
Observations

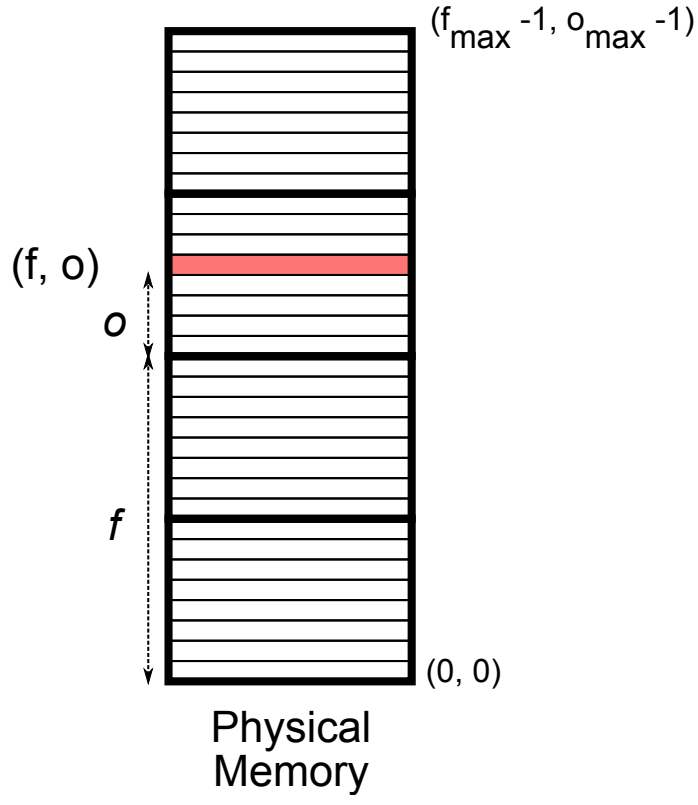
- Solves external fragmentation!
- To the price of internal fragmentation

Paged memory

Page frames

- *Physical memory* is partitioned into equal-sized *page frames*
- A physical memory address (**PA**) is a pair *(f, o)*
  - *f* is the frame number (f_{max} frames)
  - *o* is the offset within the frame (o_{max} bytes/frame)
  - PA = o_{max} * f + o

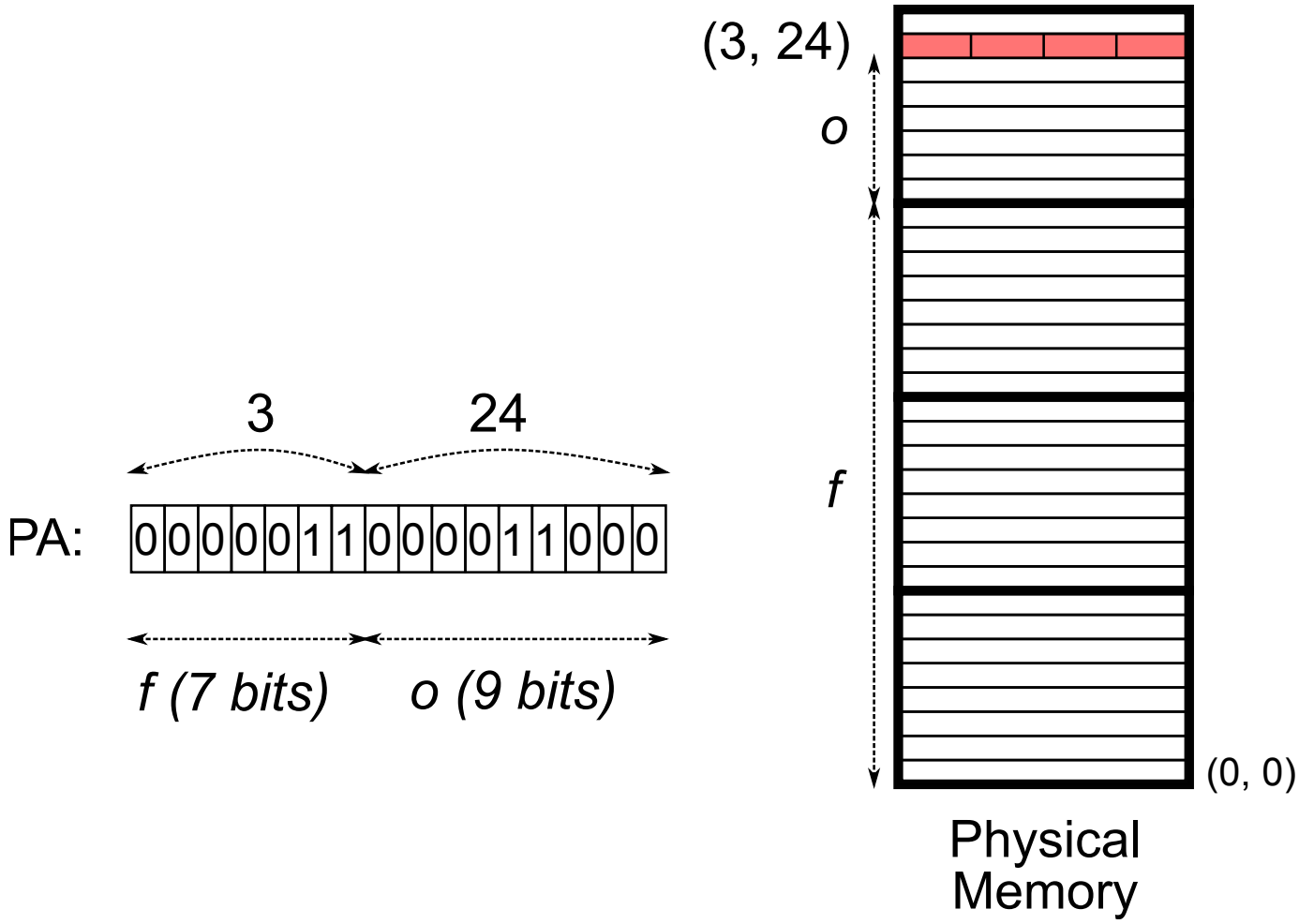




Paged memory

Physical address example

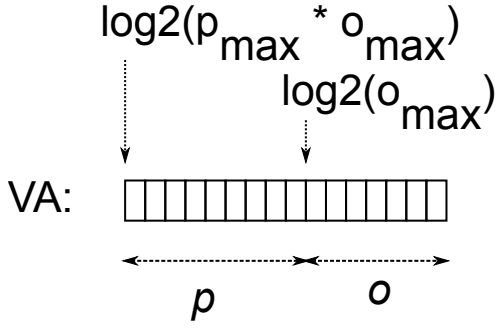
- Assuming a **16-bit** address space with **512-byte** page frames
- Where is physical address `0x0618`? (Which frame? Which offset?)

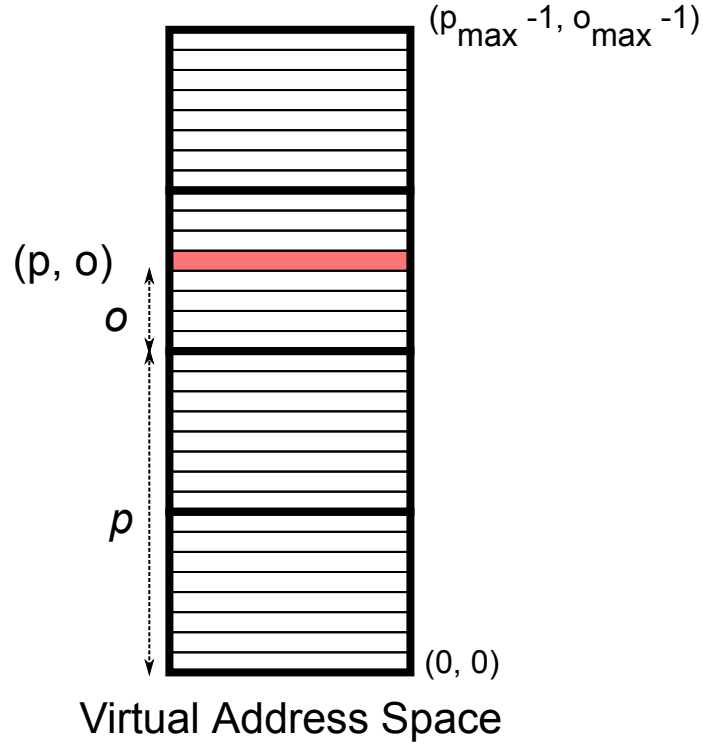


Paged memory

Virtual pages

- A process' *virtual address space* is partitioned into equal-sized *virtual pages*
  - Virtual page size is equivalent to physical page frame size
- A virtual memory address (**VA**) is a pair *(p, o)*
  - *p* is the page number (p_{max} pages)
  - *o* is the offset within the page (o_{max} bytes/page)
  - VA = o_{max} * p + o

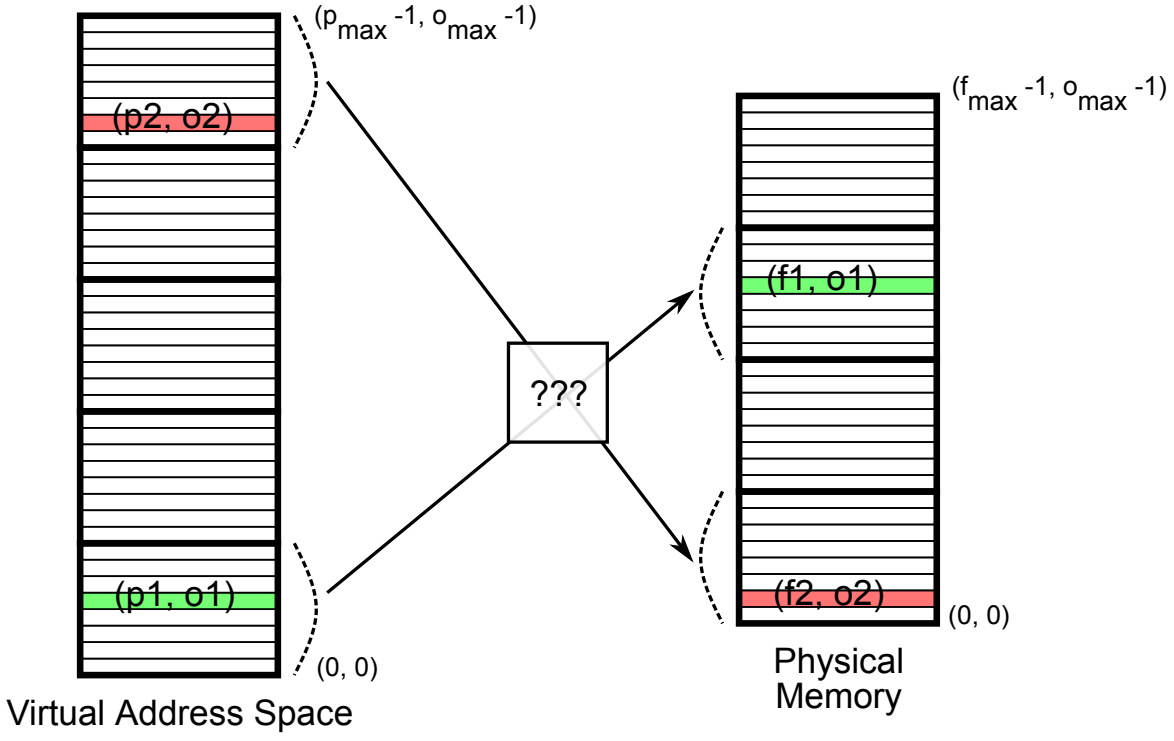




Paged memory

From virtual to physical

- Virtual pages are *mapped* to page frames
- Virtual pages are contiguous in the virtual address space, but…
  - Page frames are arbitrarily located in physical memory
  - Not all pages need to be mapped at all times (*demand-paging*)

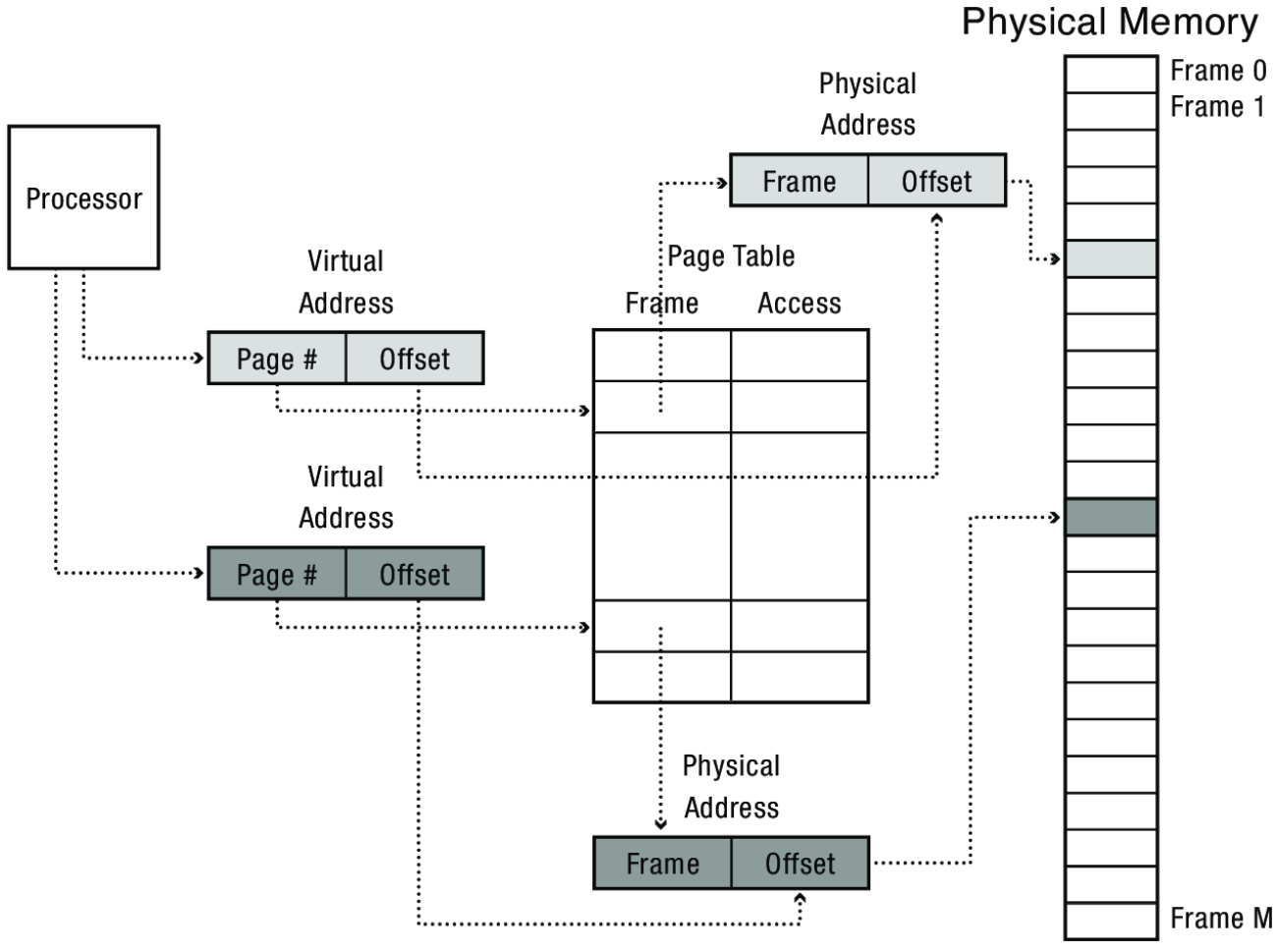


- Need translation mechanism from virtual space to physical space

Page table

Concept

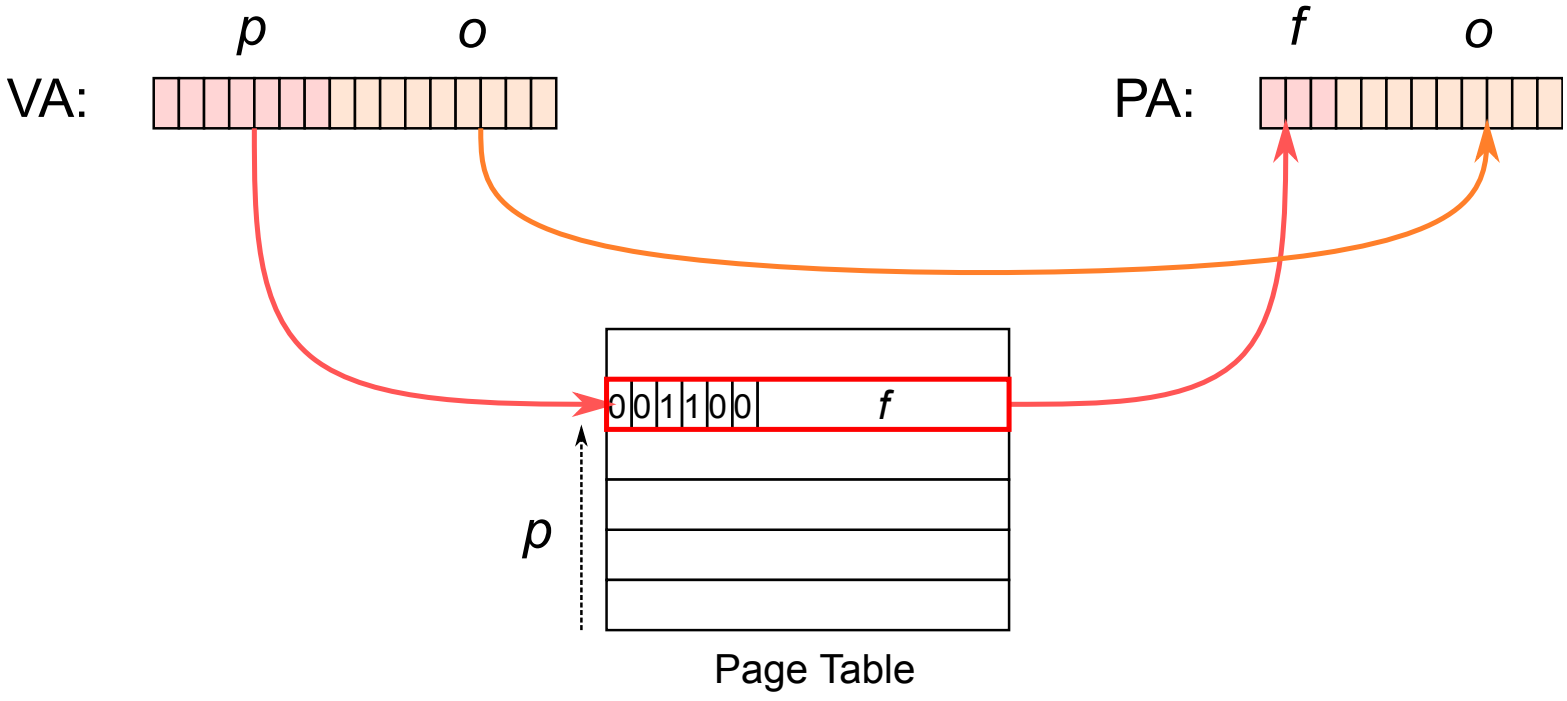
- A page table is an array of mapping entries
  - Translate virtual addresses into physical addresses
  - Invisible to the process



Page table

Details

- One page table per process
  - Part of process' state
- Page tables are stored in physical memory
- One page table entry (PTE) for each virtual page
- PTE contains
  - Frame number
  - Protection flags: R/W/E
  - Other flags: valid bit, dirty bit, use bit, etc.



Page table

Advantages

- Elimination of external fragmentation
  - To the price of some internal fragmentation
- Easy to allocate and deallocate memory from/to processes
  - Find a free page frame
  - Add a new mapping in the process' page table
- Allow processes to use more memory than the amount of physical memory
  - Swapping
  - Demand-paging (see later)
- Ability to share memory between processes
  - At page granularity
- Protection
  - At page granularity

ECS 150 - Virtual Memory: Address Translation

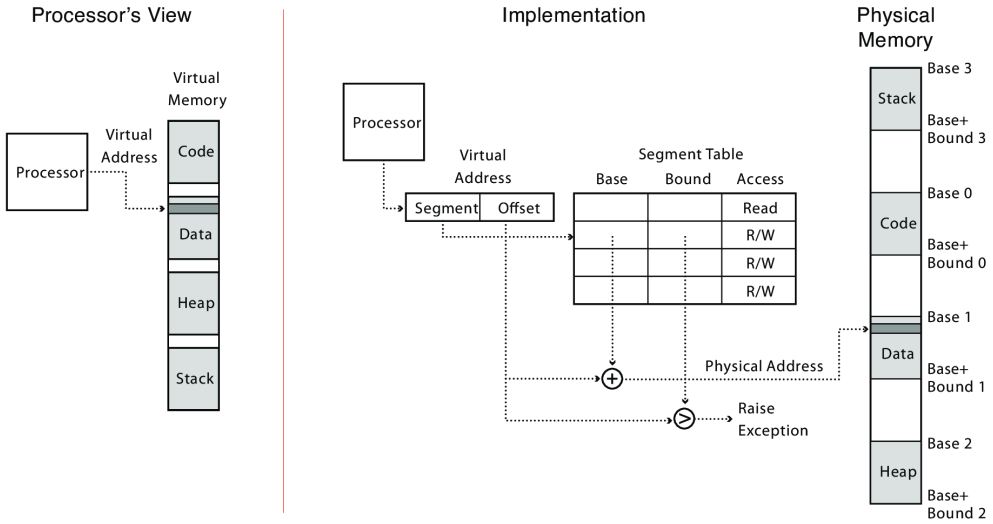
Prof. Joël Porquet-Lupine

UC Davis - 2020/2021

Recap

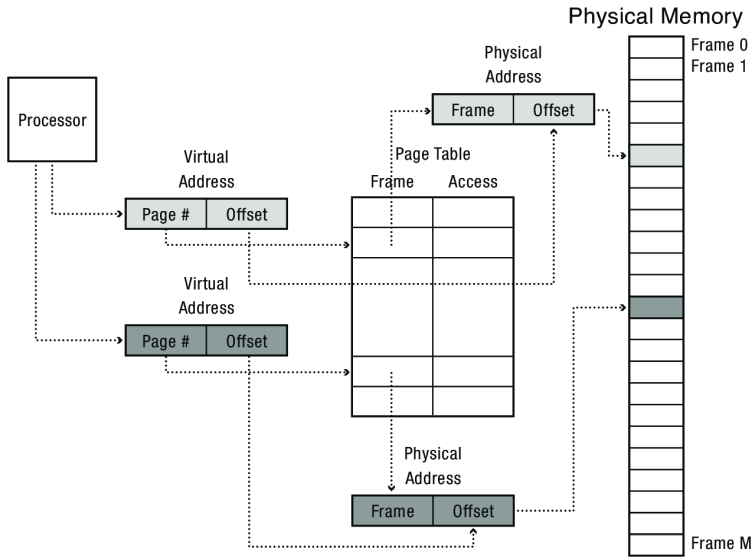
Memory segmentation

- Each segment separately mapped
  - Contiguous mapping
  - Limited number
- Segment sharing, protection and extensibility



Paged memory

- Memory divided in pages
  - Each virtual page separately mapped to physical page frame
- Use of page table
  - Page mapping



Memory efficient paging

Issue (example)

32-bit computer

- Assume a 32-bit processor, with 4 KiB pages
  - 12 bits for offset
  - 20 bits for page number
- Each page table has 1M entries, 4 bytes per entry
  - **4 MiB for each *linear* page table**
- Every process has its own page table
  - > 100 processes running on a typical computer...!

64-bit computer

- Assume a 64-bit processor, with 4 KiB pages
  - 52 bits for page number...
- Each page table would need 4 *quadrillion* entries!

Memory efficient paging

Observations

Page size

Page size	Page table	Internal fragmentation
Big (e.g., 32 KiB)	Small	Big (lots of wasted space)
Small (e.g., 1 KiB)	Big (lots of entries)	Small
Medium (e.g., 4 KiB)	Medium	Medium

Sparse virtual address space

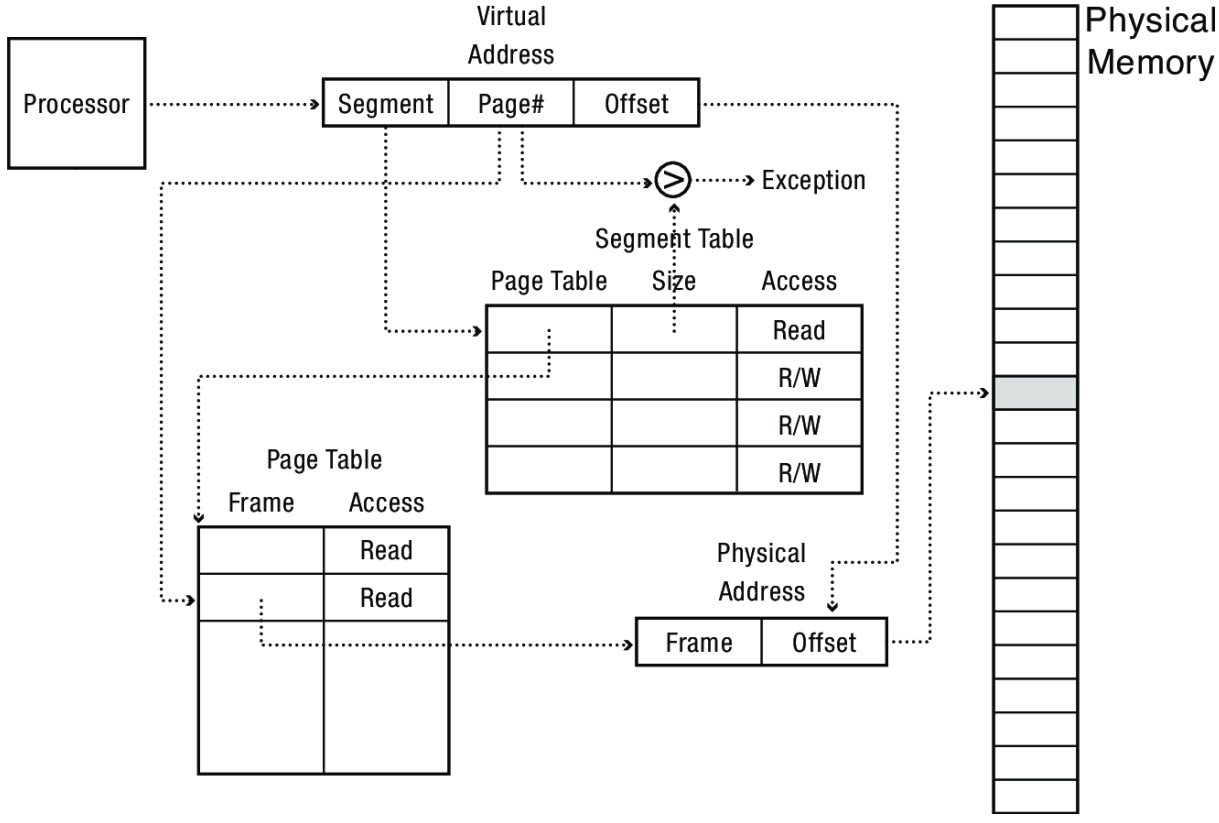
- Most processes only use a fraction of their address space
  - Typical segments
    - Code, data, heap, stack
  - Additional segments
    - Per-thread stacks (if multithreaded)
    - Memory-mapped files (`mmap()`)
    - Dynamically linked libraries

- The vast majority of page table entries will be empty

Memory efficient paging

Paged segmentation

- Use segmentation to only map actual segments
  - Use page table to map each page of each segment
  - No unnecessary page table mapping for space between segments

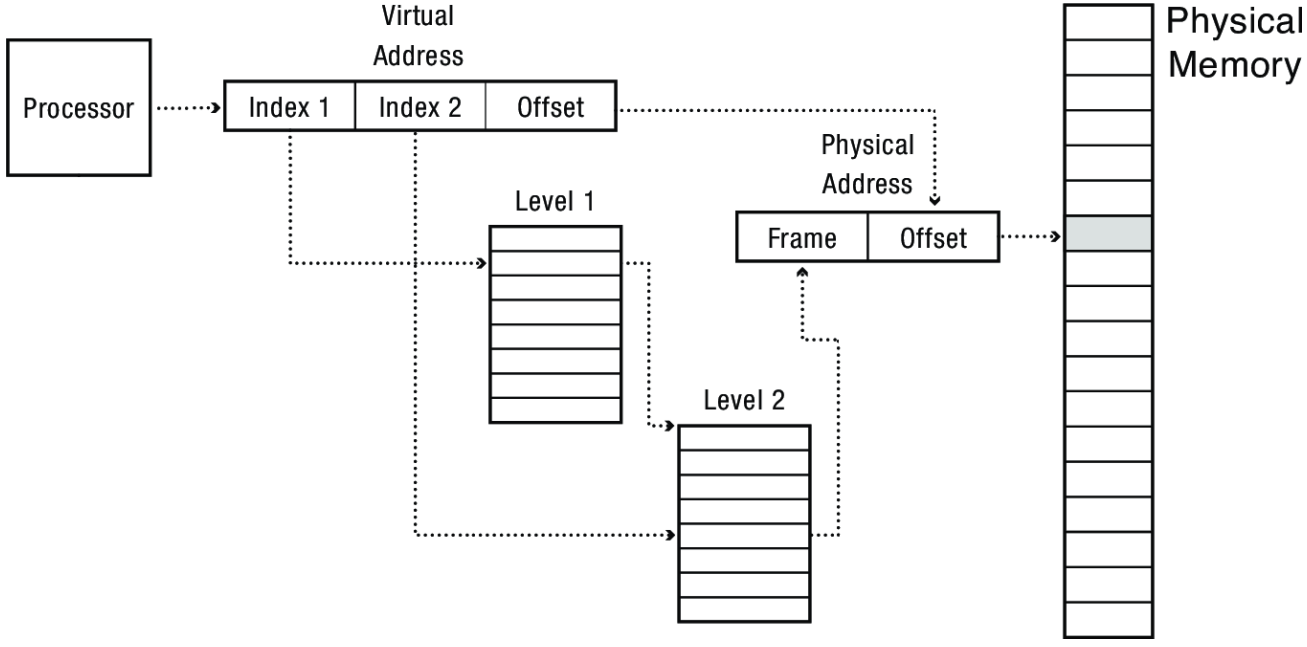


- But limited number of segments...

Memory efficient paging

Multi-level paging

- Divide virtual address space into multiple levels of fixed-size pages
  - First level covers big (fixed-size) pages (e.g., 4 MiB)
  - Second level covers actual virtual page
- Empty big pages don't point to second level



- But 2 additional memory lookups for each memory access!

Speed efficient paging

Issue

- Assuming a two-level paging strategy
- For every memory access, three accesses total
  - Two accesses to get the page table entry
  - One access to get the actual data

Observations

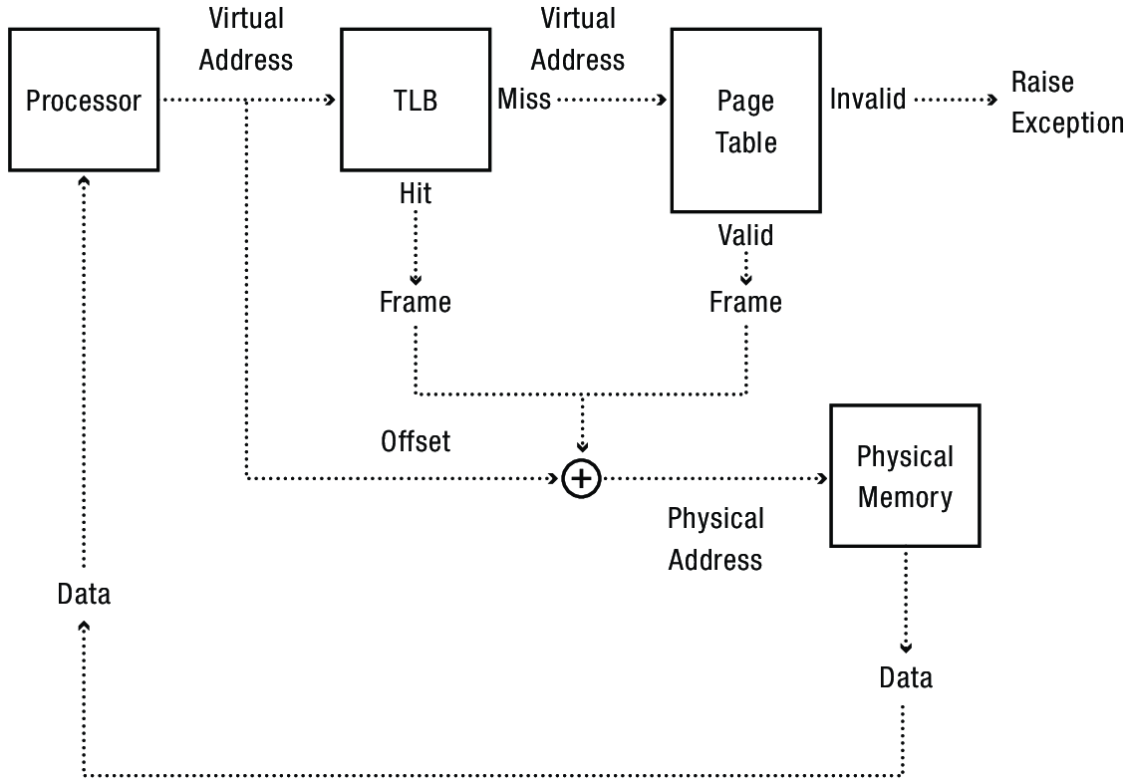
- *Locality of reference*
  - Same set of memory locations are repetitively accessed over a short period of time
  - E.g. code goes into loops, functions modify their own local variables, etc.
- *Working set* of process changes slowly
  - Processes typically only use a subset of their memory pages at a time
  - Pages containing current code, current data, current stack frame, etc.

- The same set of page table entries will be repetitively accessed

Speed efficient paging

Translation lookaside buffer (TLB)

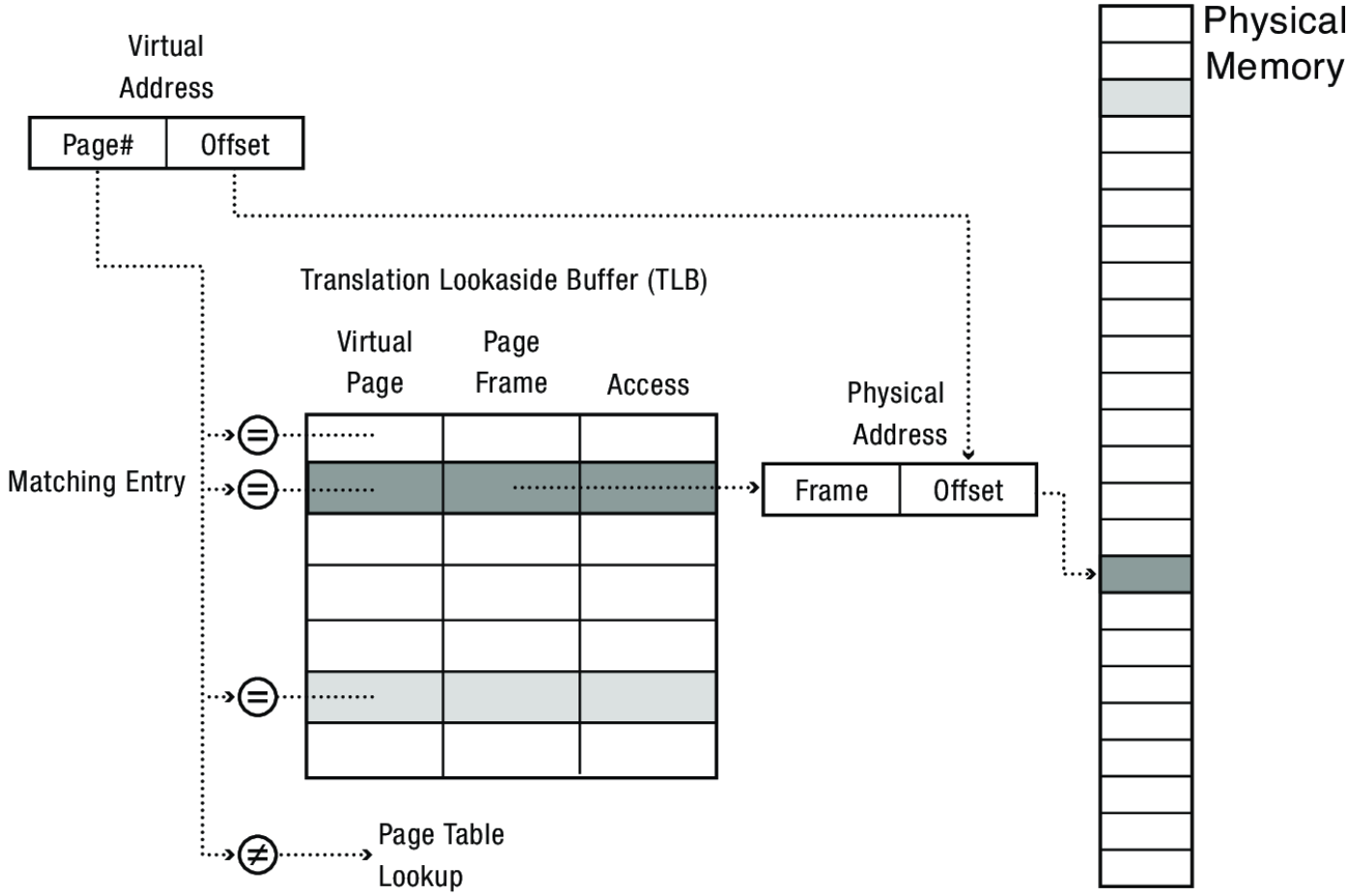
- Cache of recent virtual page to page frame translations
  - If *TLB hit*, use translation
  - If *TLB miss*, walk the page table and retrieve the translation
- Cost of translation is reduced
  - = Cost of TLB lookup + probabibity(TLB miss) * cost of page table walk



Speed efficient paging

TLB Lookup

- (Set-)associative cache of entries
- Part of processor (MMU)

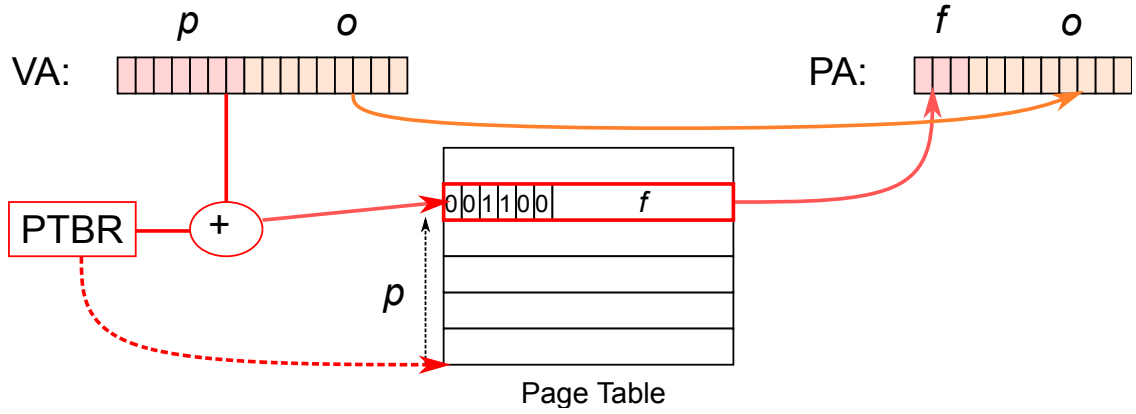


- Need to deal with TLB misses

TLB management

Hardware-controlled TLB miss

- E.g., x86 processors
- Hardware walks the page table (from the *Page Table Base Register*)
  - PTBR is part of the process' state (PCB) upon context switching
- Loads the PTE into the TLB
  - Evict existing TLB entry if necessary (e.g., least recently used)
- Goto TLB hit



Pros/cons

- Efficient (~100 cycles)
- Fixed page table structure
- Hardware cost (complex MMU)

TLB management

Software-controlled TLB miss

- E.g., many RISC processors
- Trap to kernel
- Software walks page table and loads PTE into TLB
  - Evict existing TLB entry if necessary
- Resume to faulting instruction

Pros/cons

- No extra hardware cost
- Flexible page table structure
  - E.g. Inverted page table
- Slow (> 1000 cycles)

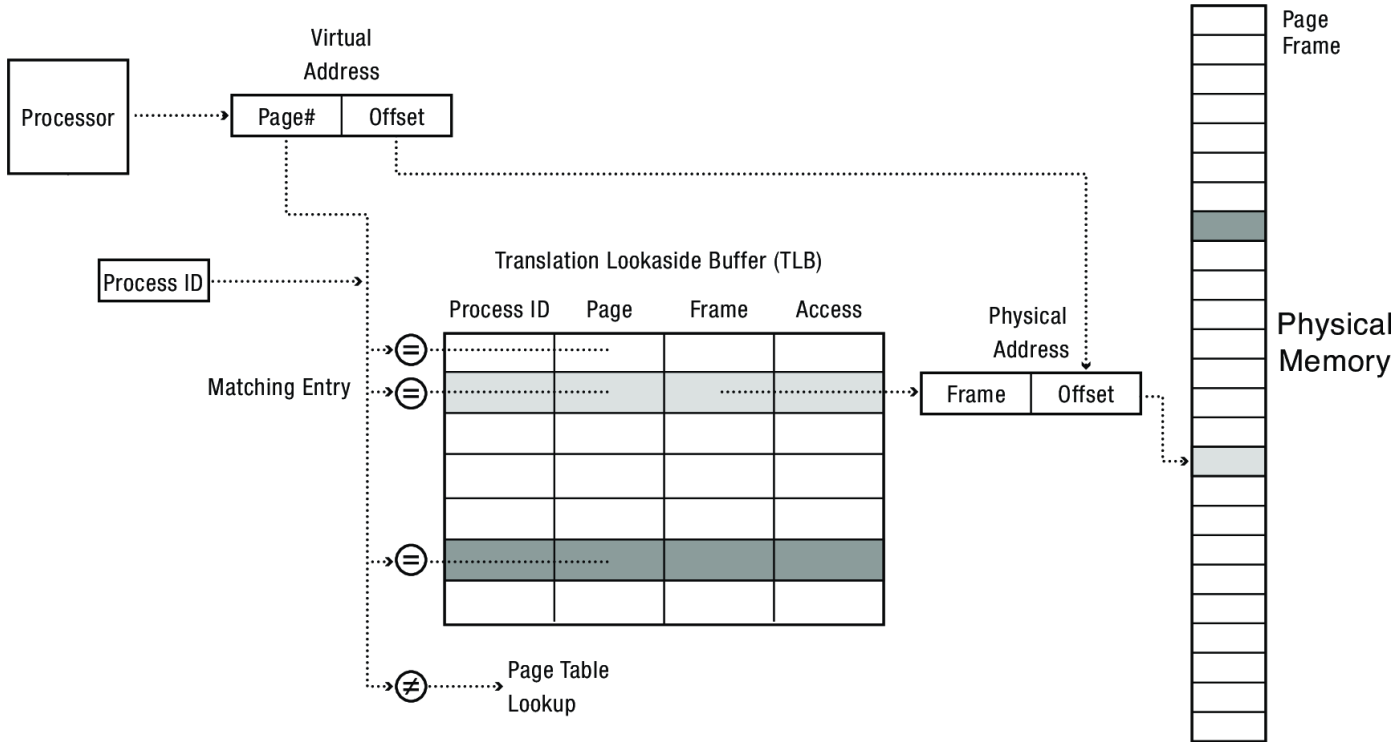
TLB management

Issue: context switch

- Each process has its own virtual address space
- Upon context switching, TLB entries need to be purged
  - But will incur many TLB misses until working set is mapped again

Tagged TLB

- Each TLB entry is associated to a *process ID*
- TLB hit only if entry's PID matches current process's PID



TLB management

Issue: consistency

- OS might want to change permissions on a page for a process
  - Especially important on *permission reduction* (e.g. from RW to R)
- But TLB may contain old translation

Local processor

- OS requests local processor to purge specific TLB entries
  - Via specific processor instructions

Multicore

- OS must request each core to purge specific TLB entries
  - Via Inter-Processor Interrupts
  - *TLB shootdown*

		Process ID	VirtualPage	PageFrame	Access
Processor 1 TLB	=	0	0x0053	0x0003	R/W
	=	1	0x40FF	0x0012	R/W
Processor 2 TLB	=	0	0x0053	0x0003	R/W
	=	0	0x0001	0x0005	Read
Processor 3 TLB	=	1	0x40FF	0x0012	R/W
	=	0	0x0001	0x0005	Read

Address translation

- Process isolation
  - Against other processes and the kernel
- Page sharing
  - Efficient interprocess communication: shared memory buffers
  - Code sharing (especially useful for shared libraries)
- Dynamic memory allocation
  - Allocate/initialize stack and heap on demand
- Program debugging
  - Protect code segment from being overwritten
  - Data breakpoints
- Memory-mapped files
  - Access file data using regular memory load/store instructions
- Demand-paging virtual memory
  - Artificially increase the amount of available memory, using the disk
  - Program initialization: start running process before it's even loaded