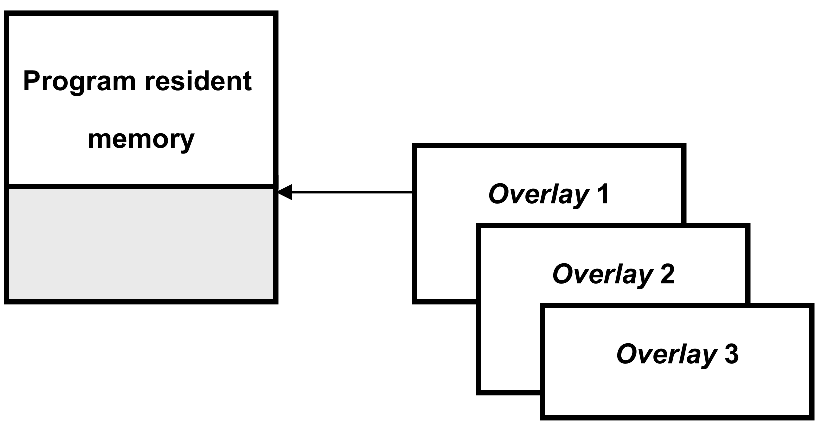
Memory management

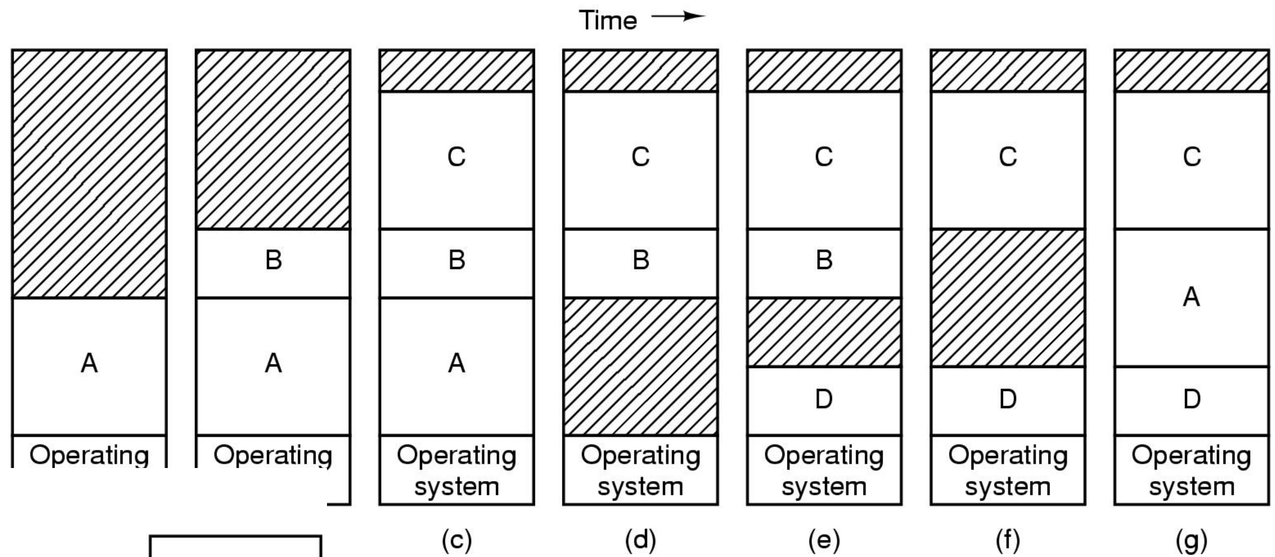
- Single process
  - All memory assigned to the process
  - Addresses defined at compile time
- Multiple processes. How to:
  - assign memory
  - manage addresses?
  - manage relocation?
  - manage program grow?

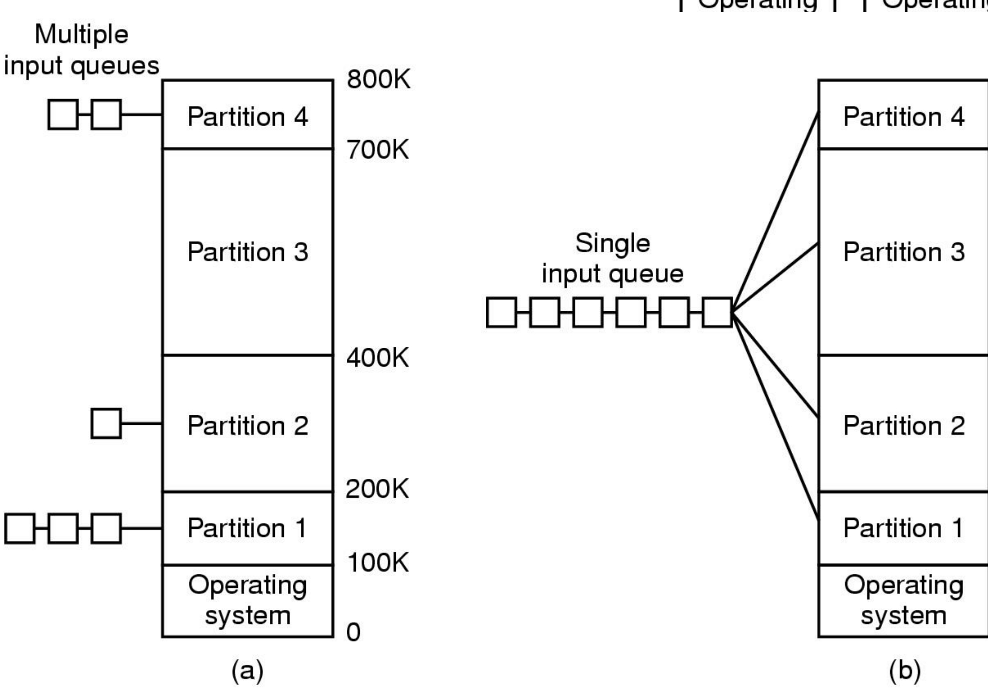
manage program grow?

- overlay
  - Statically defined memory zone
  - loaded on demand
    - By the program
  - may be replaced



Memory management



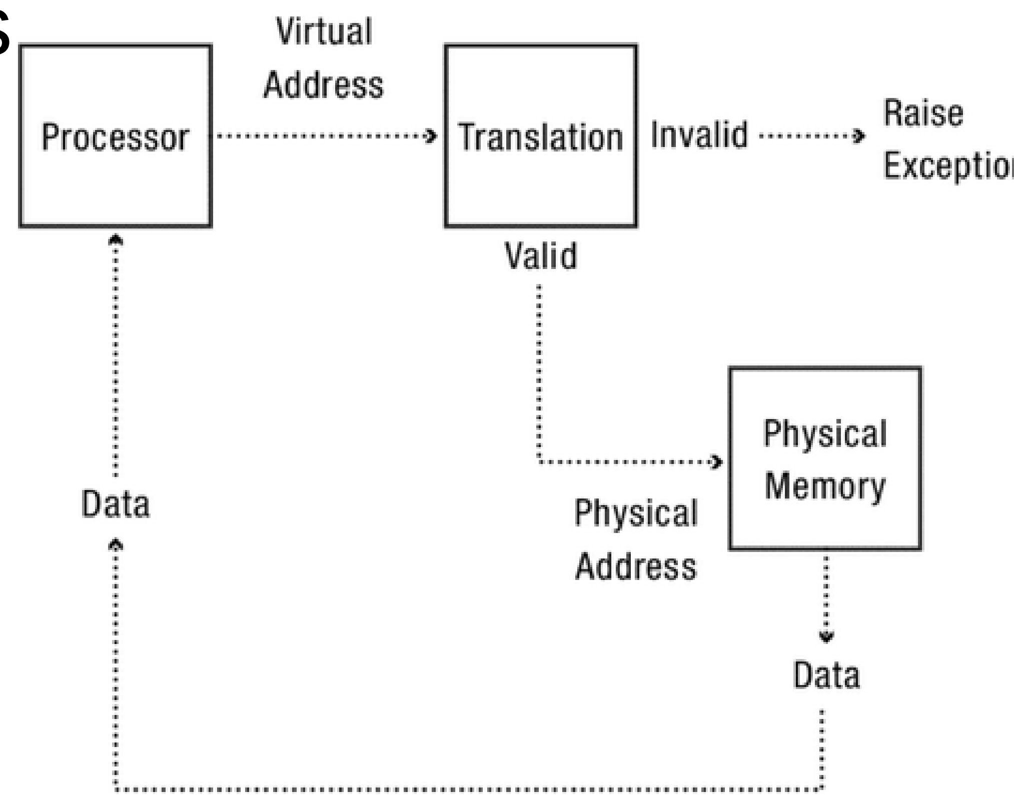


```c
for(i = 0; i< 1000; i++){
    vect[i] = c;
}
```

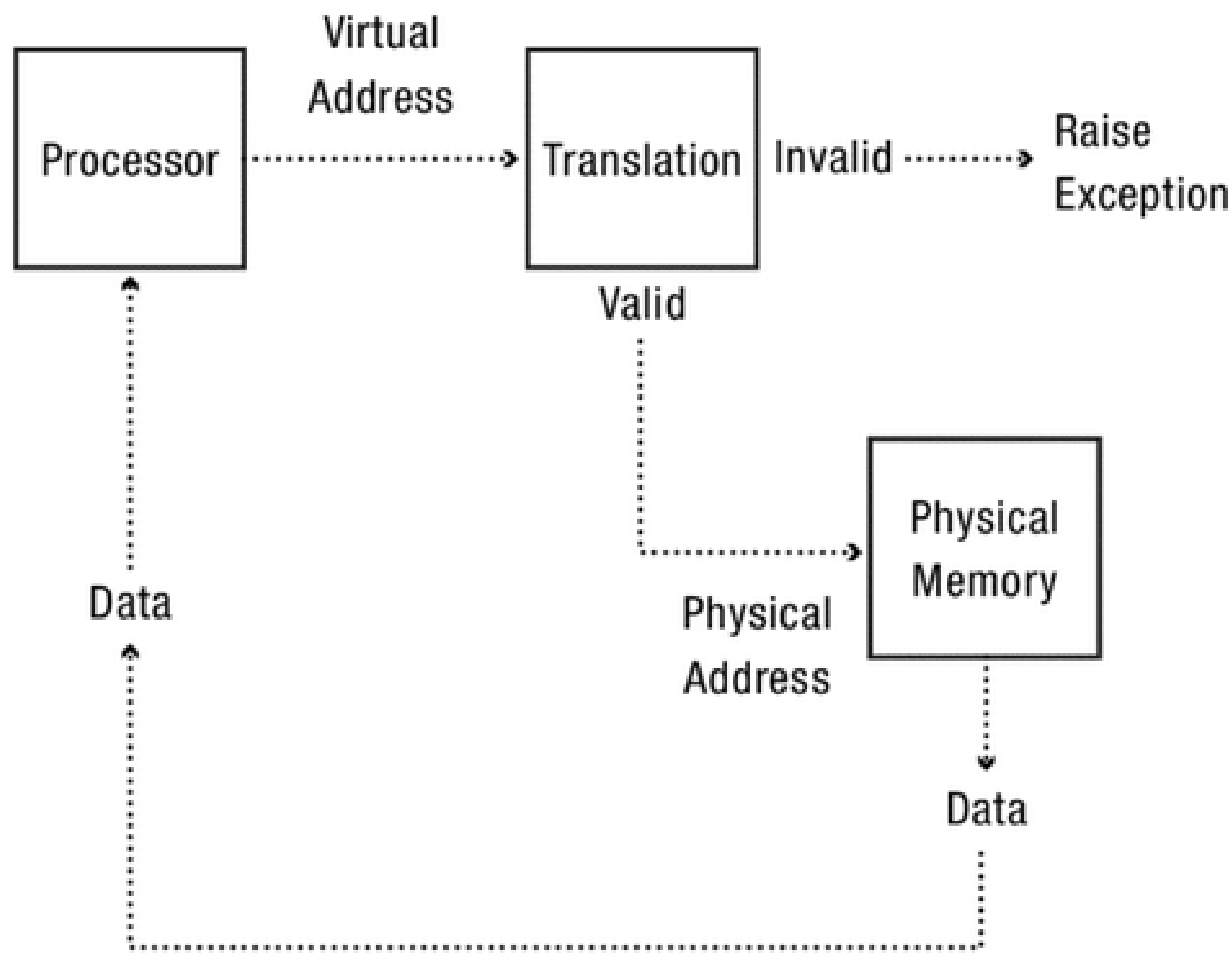
```
.L3:
movl	-4(%rbp), %eax
cltq
movzbl	-5(%rbp), %edx
movb	%dl, -1008(%rbp,%rax)
addl	$1, -4(%rbp)
.L2:
cmpl	$999, -4(%rbp)
jle	.L3
```

Virtual memory

- Program addresses
  - are independent on the physical location
- memory manager
  - translates addresses
    - virtual -> physical
  - verifies permitions
- Address Translation



Address Translation Concept

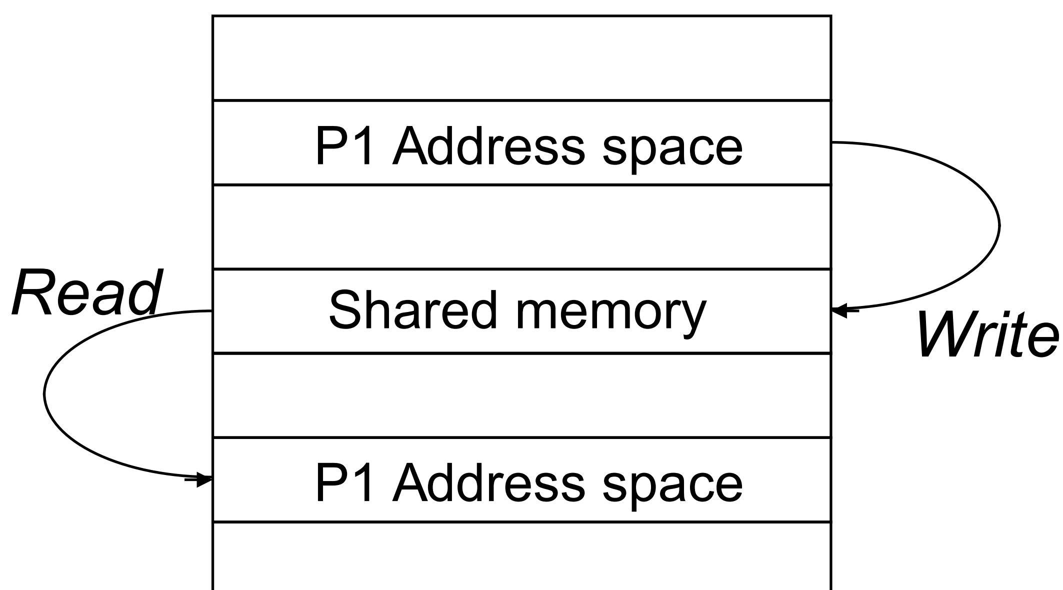


Adress translation

- Process isolation
- Interprocess communication
- Shared code
- Program debugging
- Efficient I/O
- Memory mapped files
- Virtual memory
- Checkpointing
- Process migration
- Information flow control
- DSM

Shared memory

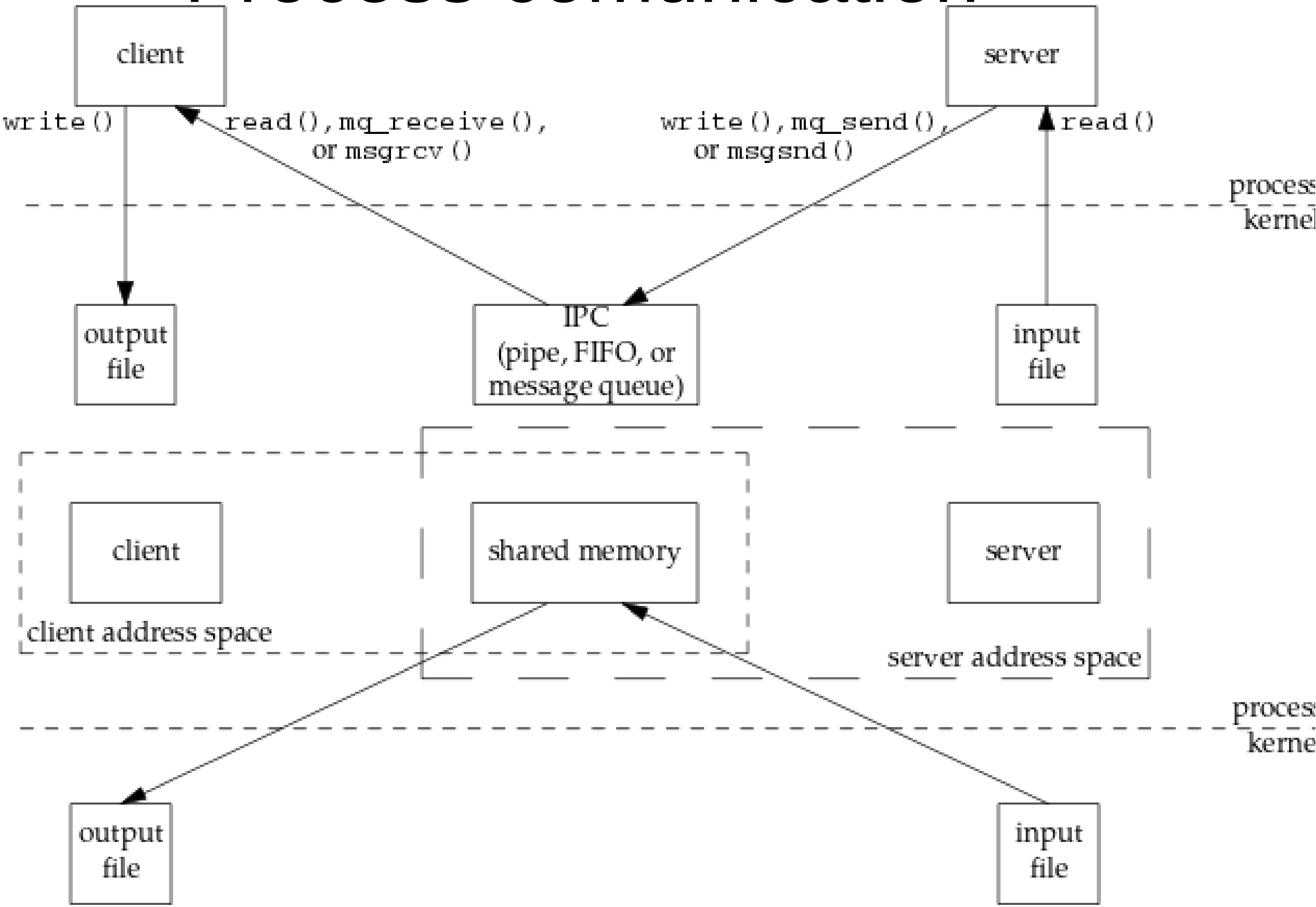
- Processes use regular variables/vector to communicate
  - Available in shared memory
- Should be explicitly created.
  - Each process has its own address space



Shared memory Advantages

- Random Access
  - you can update a small piece in the middle of a data structure, rather than the entire structure
- Efficiency
  - unlike message queues and pipes,
    - copy data between user memory ↔ kernel memory
  - shared memory is directly accessed
  - Shared memory resides in the user process memory
    - Is shared among other processes

Process comunication



Shared memory Disadvantages

- No automatic synchronization
  - In pipes or message queues
  - Programmer has to provide synchronization
    - Semaphores or signals.
- Pointers are only valid within a given process.
  - Pointer offsets cannot be assumed to be valid across inter-process boundaries.
  - This complicates the sharing of linked lists or binary trees.
- Variables are “produced by the compiler”
  - Names can not be used to access shared memory

Shared memory in *NIX

- System V shared memory
  - Original shared memory mechanism, still widely used
  - Sharing between unrelated processes
- Shared mappings – mmap
  - Shared file mappings
  - Sharing between unrelated processes, backed by filesystem
  - Shared anonymous mappings
    - Sharing between related processes only (related via fork())
- POSIX shared memory
  - Sharing between unrelated processes, without overhead of filesystem I/O
  - Intended to be simpler and better than older APIs

Shared memory in *NIX

- Programming steps
  - Define Shared data structure
  - Creation of memory segment
  - Configuration
  - Assignment to address
  - Access
  - Disconnection
  - Destruction

In multiple processes

System V shared memory

- Shared memory operations
  - shmget
    - allocates a shared memory segment
  - shmctl
    - allows the user to receive information on a shared memory segment,
    - set the owner, group, and permissions of a shared memory segment,
    - destroy a segment
  - shmat
    - attaches the shared memory segment (identified by shmid) to the address space of the calling process
  - shmdt
    - detaches the shared memory segment (located at the address specified by shmaddr) from the address space of the calling process

System V shared memory

- int shmget(key_t key, size_t size, int shmflg);
  - key known by all processes
  - Flags - IPC_CREAT | 0666

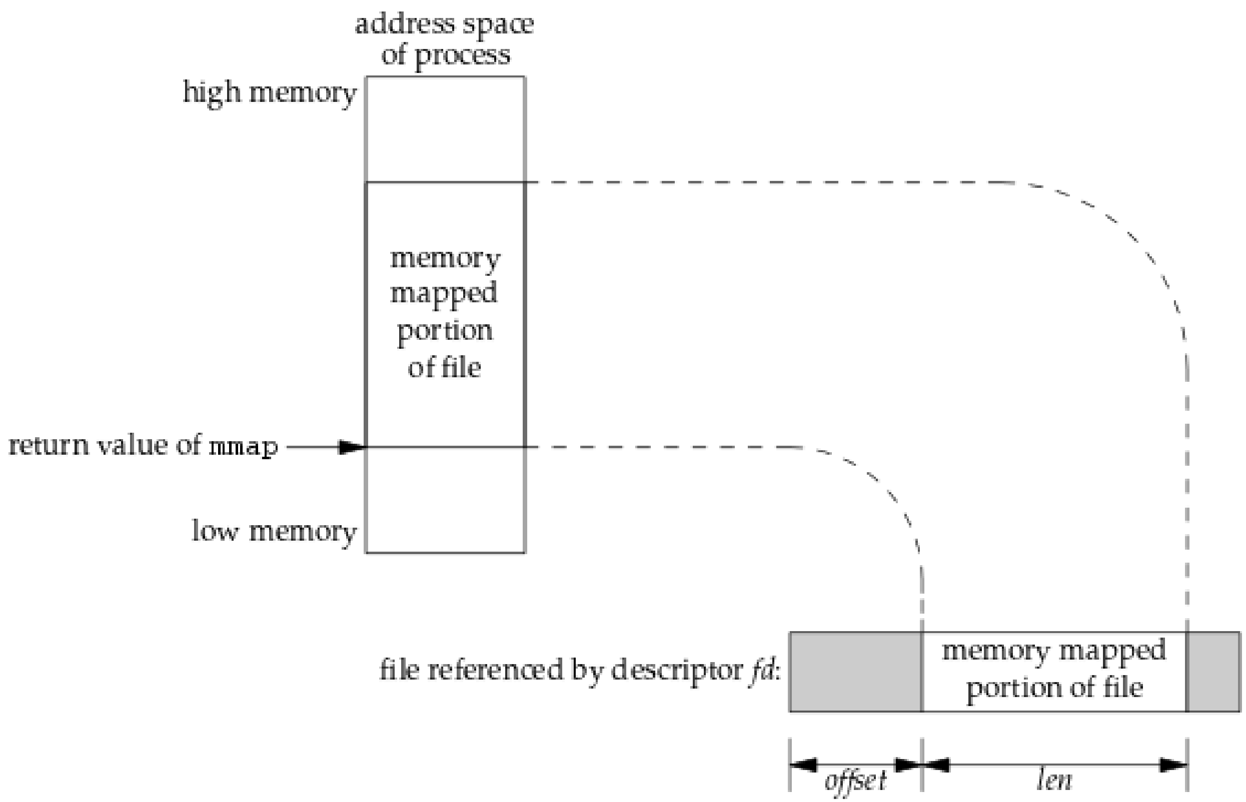
- void *shmat(int shmid, const void *shmaddr, int shmflg);
    - Shmid – returned by shmget
    - shmaddr – NULL or other address
    - Shmflg - SHM_EXEC   SHM_RDONLY

System V shared memory

- char * shm;
- key = 5678;
- /* Create the segment */
- if ((shmid = shmget(key, SHMSZ, IPC_CREAT | 0666)) < 0) {
- perror("shmget"); exit(1);
- }
- /*Now we attach the segment to our data space.*/
- if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
- perror("shmat"); exit(1);
- }

- Repeated on several processes
  - Key must be known
- Followed by fork
  - Shm value is shared

Memory mapped files

- Access a file contect
  - With memory access operations
  - Assignments e accesses
- 

Memory mapped files

- void *mmap(void * addr, size_tlen ,

  int prot, int flags,

  int fd, off_toffset);

  - Prot –
    - PROT_READ PROT_WRITE PROT_EXEC PROT_NONE
  - Flags
    - MAP_SHARED MAP_PRIVATE MAP_FIXED
- int munmap(void * addr, size_t len);

Memory access

- After mmap a variable contains a pointer to region
  - Programmer can access that memory as a
    - Pointer to variable
    - Vector
- Is it possible to create linked lists in shared memory?
  - No. mmap in different processes returns different addresses

Memory mapped files

- Int * ptr;
- fd = open(argv[1], O_RDWR | O_CREAT, FILE_MODE);
- ptr = Mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
- Close(fd);

Mmap and fork

- Open could be avoided
  - Parent and son share the address to memory
  - Memory is shared among processes
- 4.4BSD introduced anonymous memory sharing
  - Flag – MAP_SHARED|MAP_ANON
  - Fd -1
- ptr = Mmap(NULL, sizeof(int),

```
                PROT_READ | PROT_WRITE,
                MAP_SHARED | MAP_ANON,
                 -1, 0);
```

POSIX shared memory

- Mmap
  - File system incurrs overhead
- Sharing between unrelated processes, without overhead of filesystem I/O
- Intended to be simpler and better than older APIs
- New function to create memory regions

POSIX shared memory

- Create/opens a shared memory space
  - fd_mem = shm_open("/myregion", /*region name*/
  - O_CREAT | O_RDWR, 0600);
  - Memory regions are created with size 0
- Assign a size
  - ftrucate (fd_mem, sizeof(int))
  - If the object has already been sized by another process, you can get its size with the fstat function
- A global region has been created
  - Data stored is kernel persistent
  - But still inaccessible by processes → use mmap with fd_mem

- int fd = shm_open(memname,
- O_CREAT | O_TRUNC | O_RDWR, 0666);
- if (fd == -1)
- error_and_die("shm_open");
- int r = ftruncate(fd, sizeof(int));
- if (r != 0)
- error_and_die("ftruncate");
- int *v_int = mmap(0, sizeof(int),
- PROT_READ | PROT_WRITE, MAP_SHARED,
- fd, 0);

Shutdown

- Close the shared memory object
  - close(fd_mem)
- Unmap the shared memory object:
  - munmap (pointer, SHM_SIZE);
  - The address become free to be used.
- At this stage if other processes open and map a memory region
  - They can access data previously written
- To remove permanently the shared memory object:
  - shm_unlink (SHARED_MEMORY_NAME);
  - The object is effectively deleted after the last call to shm_unlink

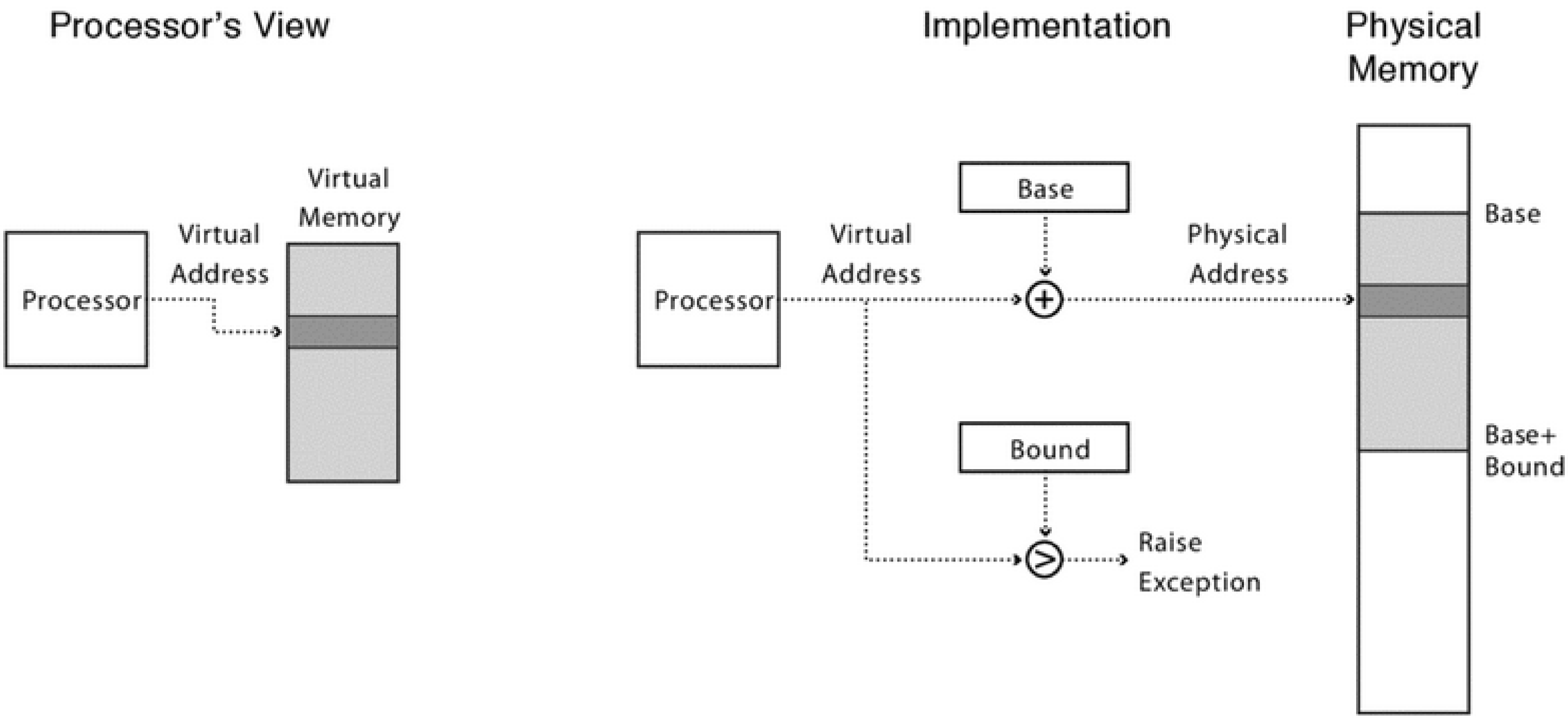
Address Translation

- Flexible Address Translation
  - Base and bound
  - Segmentation
  - Paging
  - Multilevel translation
- Efficient Address Translation
  - Translation Lookaside Buffers
  - Virtually and physically addressed caches

Address Translation Goals

- Memory protection
- Memory sharing
  - Shared libraries, interprocess communication
- Sparse addresses
  - Multiple regions of dynamic allocation (heaps/stacks)
- Efficiency
  - Memory placement
  - Runtime lookup
  - Compact translation tables
- Portability

Virtually Addressed Base and Bounds



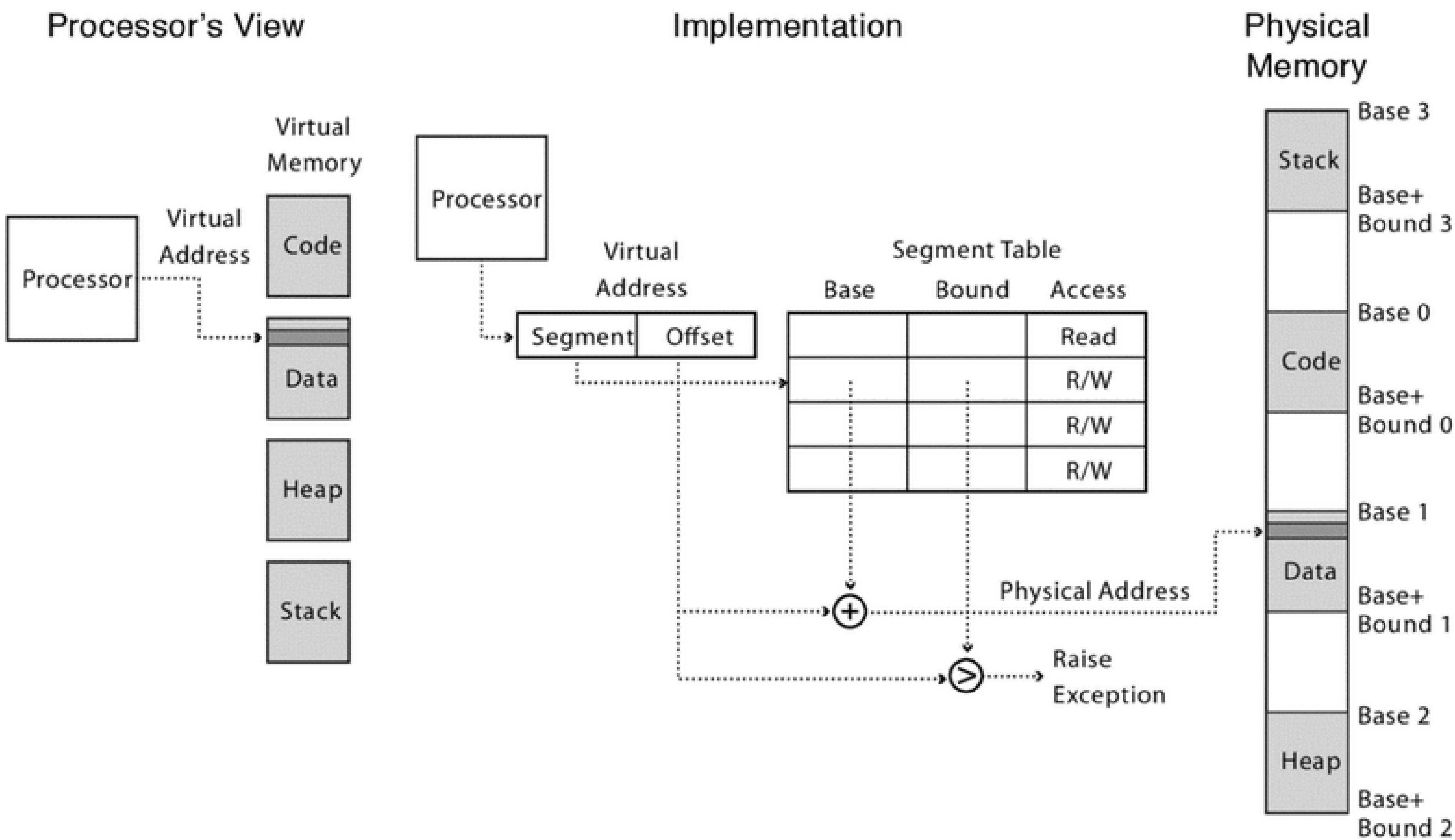
Virtually Addressed Base and Bounds

- Simple and fast
- coarse-grained protection
  - at the process levels
  - impossible to prevent self overriding of code
- Difficult to share memory regions
- Memory needs to be continuous
  - hard to implement dynamic memory
    - stack heap, …

Segmentation

- Evolution of base and bound
  - one process -> multiple bases+bounds
  - one process -> multiple segments
- Segment: base + bound
  - multiple sizes
  - each segment stored continuously
  - multiple access permissions
    - execute-only (code) or read-write (data)

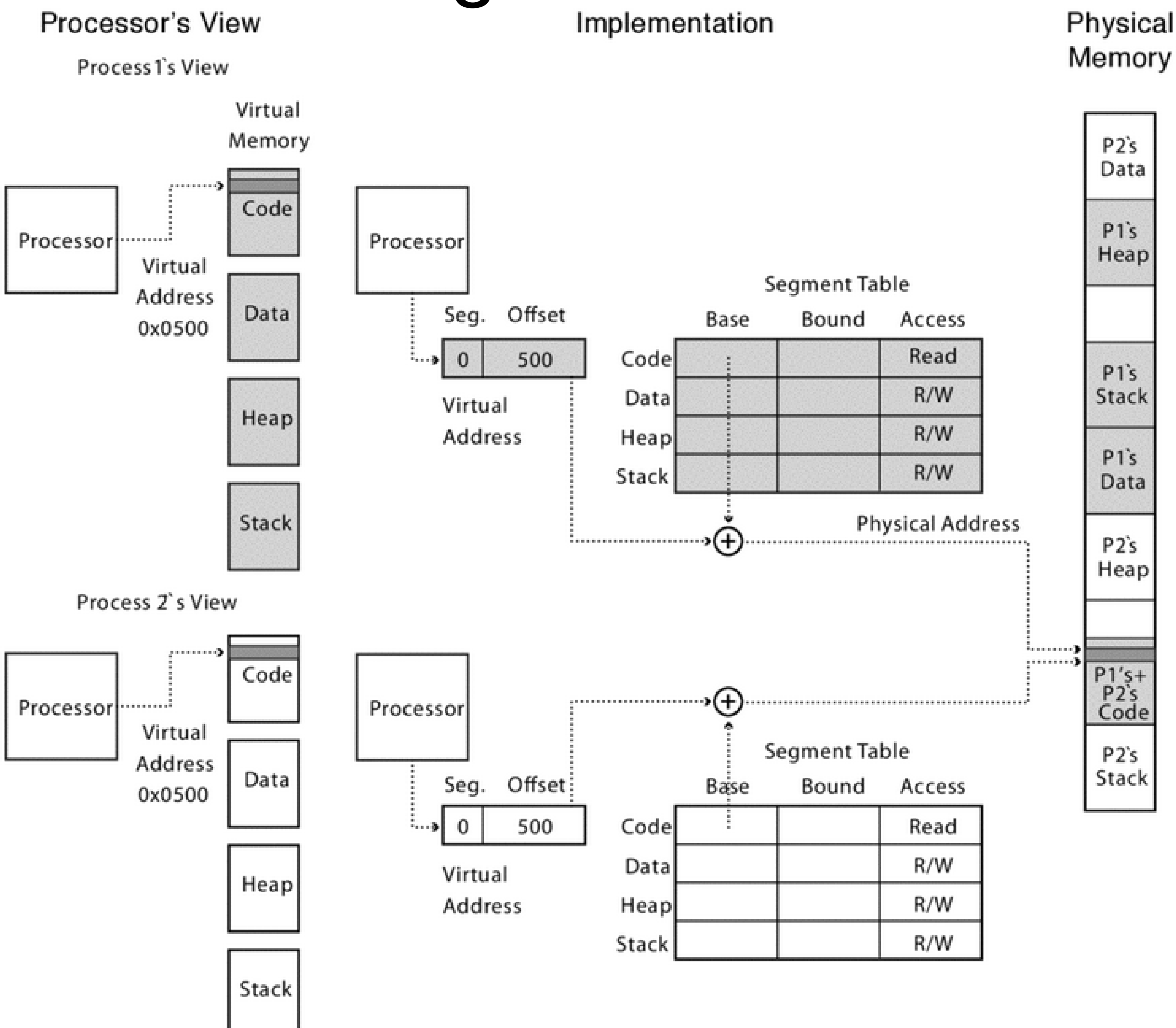
Segmentation



Segmentation

- Segmented memory has gaps
  - program memory is not a single continuous regions
  - each segment start at a different base
- Access outside segments
  - HW generates interrupt
  - Kernel generates segmentation fault
- Addressable space
  - depends on the segment+offset length
-

Segmentation

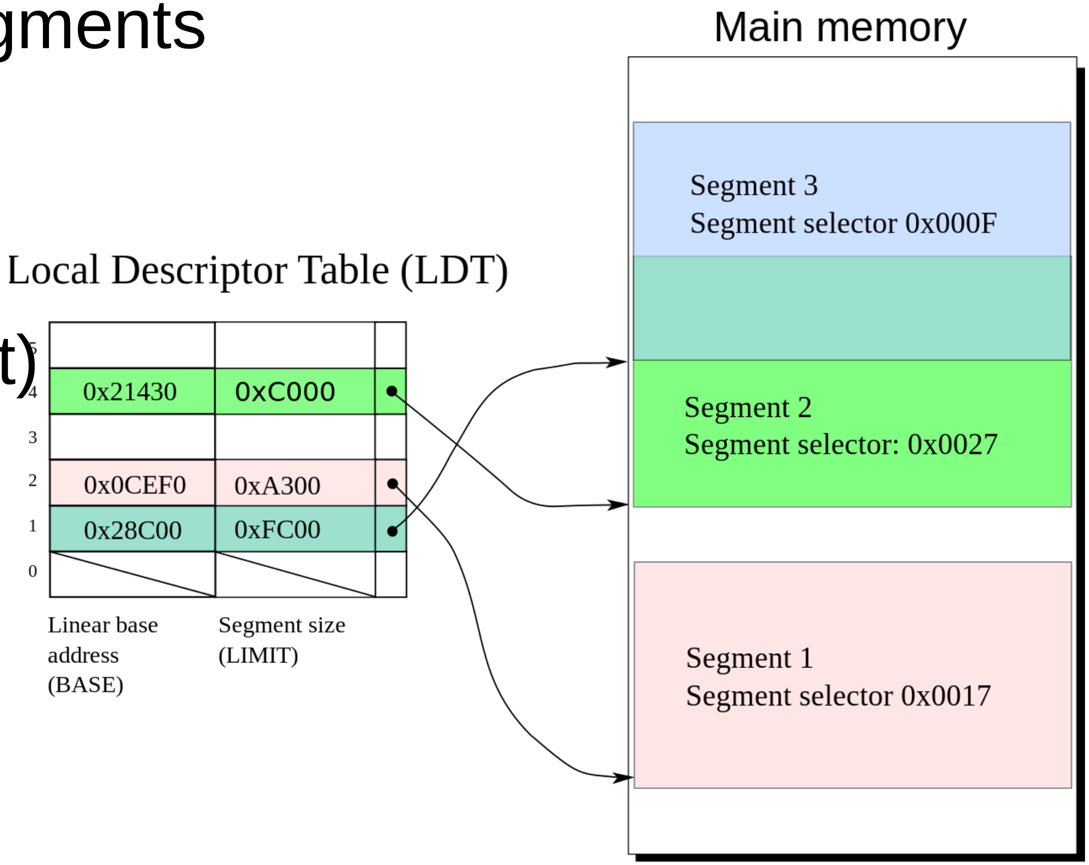


Segmentation

- Total memory depends
  - base + offset
- Kernel should manage
  - placement of each segment on physical memory
  - permission of access

Segmentation

- Overhead of managing
  - Large number of segments
  - Segments of variable size
  - Dynamically growing segments
- Creation of a segment
  - Virtual address (easy)
  - Memory location (difficult)
    - Fragmentation
    - Segments grows



Paged memory

- Alternative to segmentation
  - Fixed sized chunks
    - Page frames
  - Address translation is similar to segmentation
    - Segment table → page table
    - Table entry pomnts to page base
    - No need for bound
- Process still views memory as linear
  - Linear memory (virtula) accesses
    - Multiple accessed pages

Paged memory

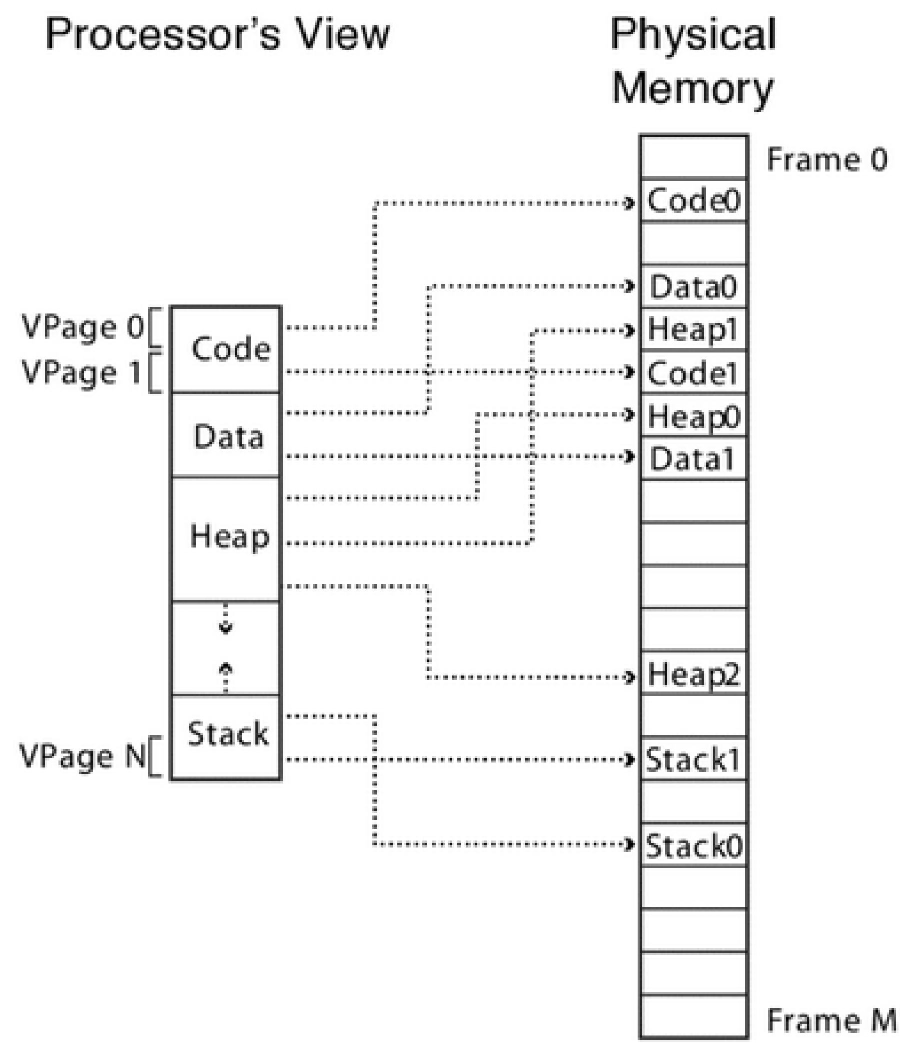
- Big advantage over segmentation
  - Free-space allocation is straighforward
  - Physical memory represetation
    - Bitmap (one bit per physical page)
- Memory sharing
  - Setting of pointer in page table

Optimizations

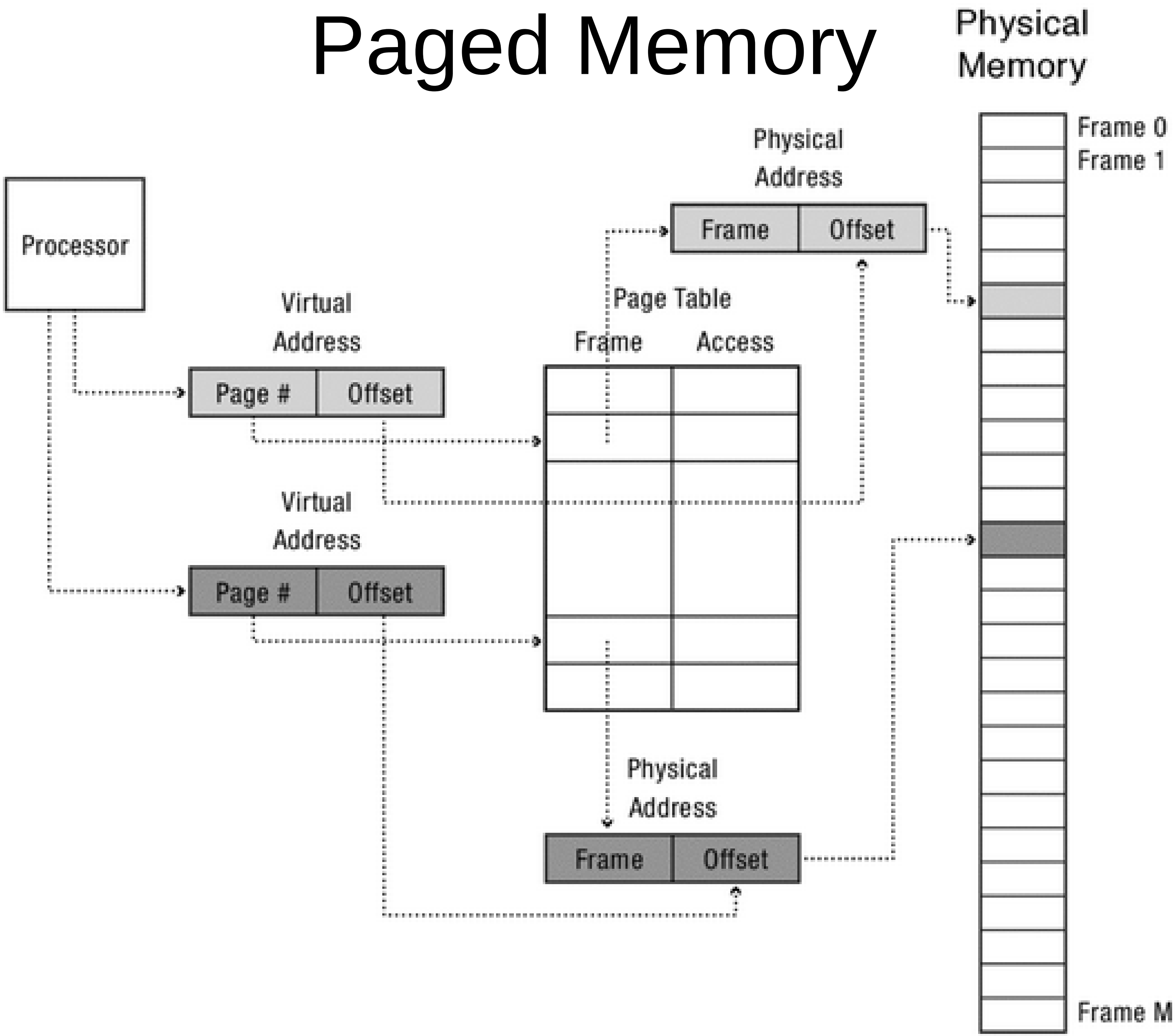
- Segmentation+paged memory
  - Copy on write
  - Zero on reference
  - Data breakpoints
    - Stop program when variable gets changed
- Paged memory
  - Program can execute without all code/data
    - Pages can be loaded while program is executing
  -

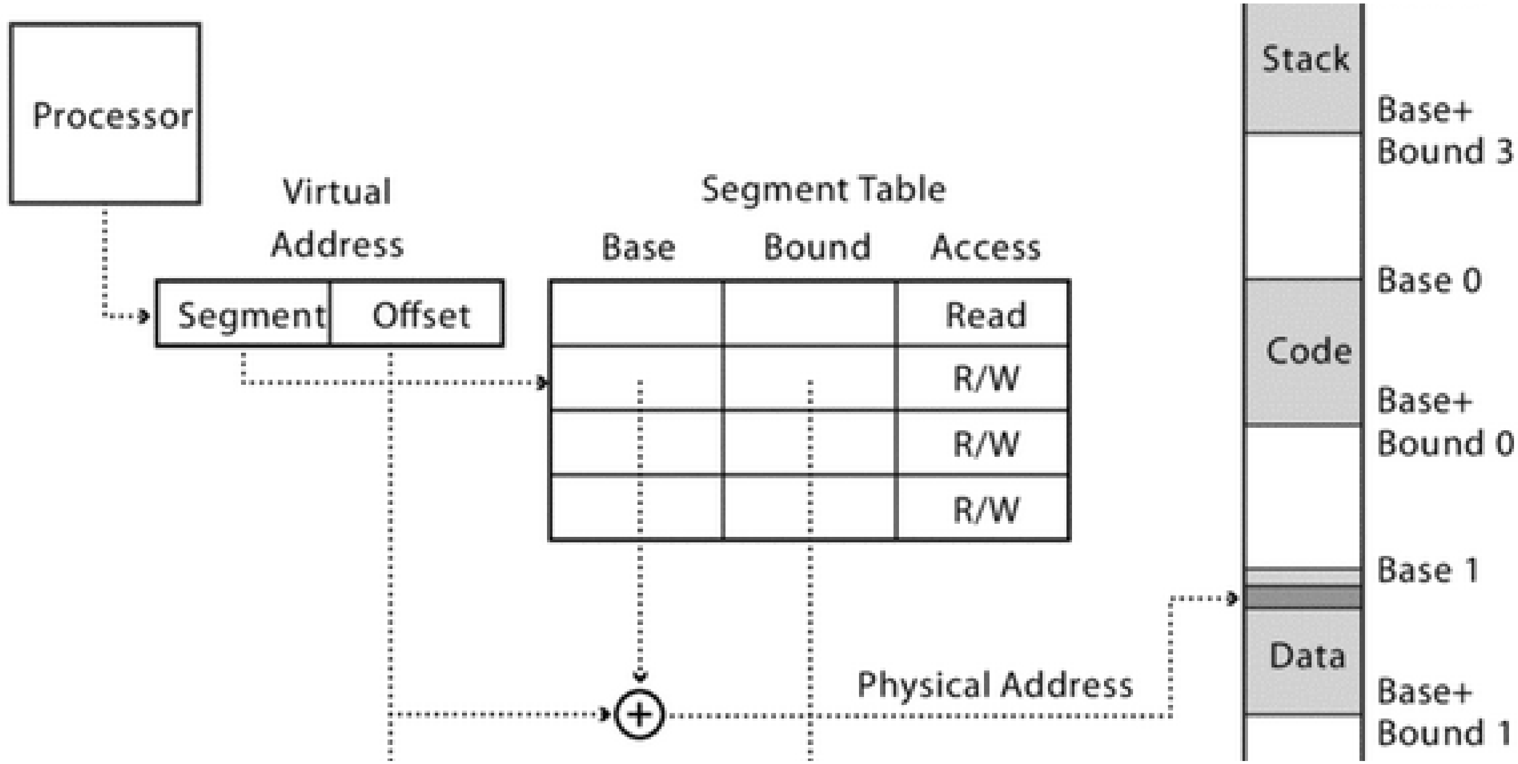
Paged memory

- Downsides
  - Virtual address management is more difficult
    - Stack and heap should be continuous
    - Compiler should guarantee this
  - How to make it work with multiple threads?

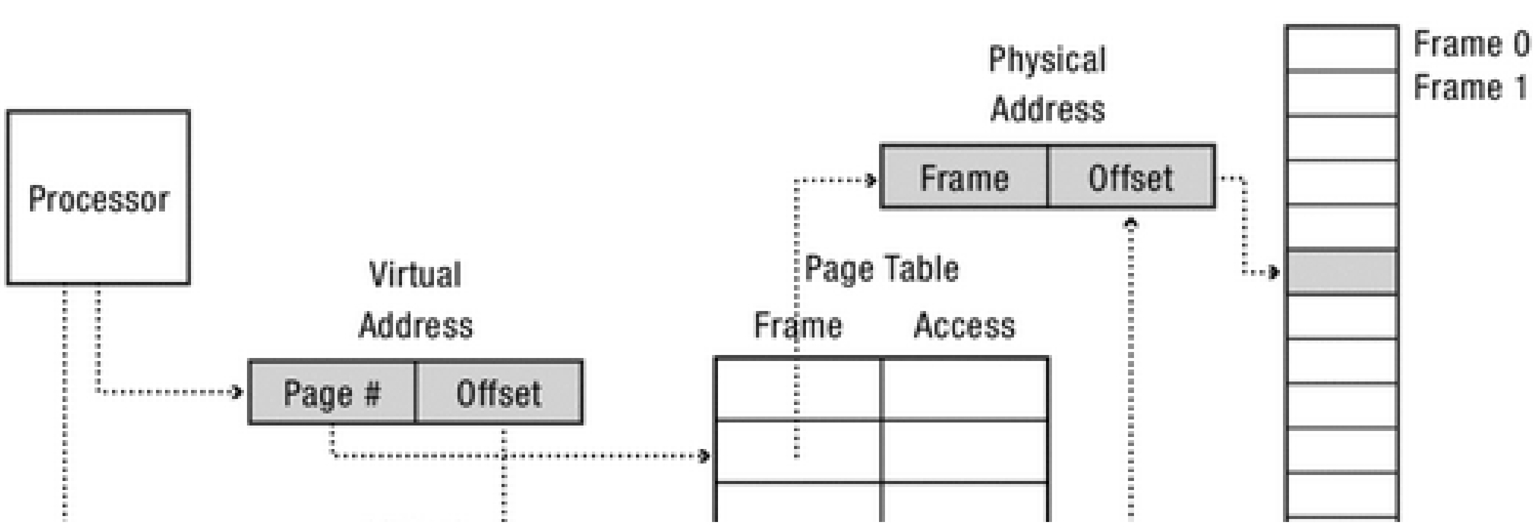


Paged Memory





- Segmentation
- Page



How to define virtual addresses?

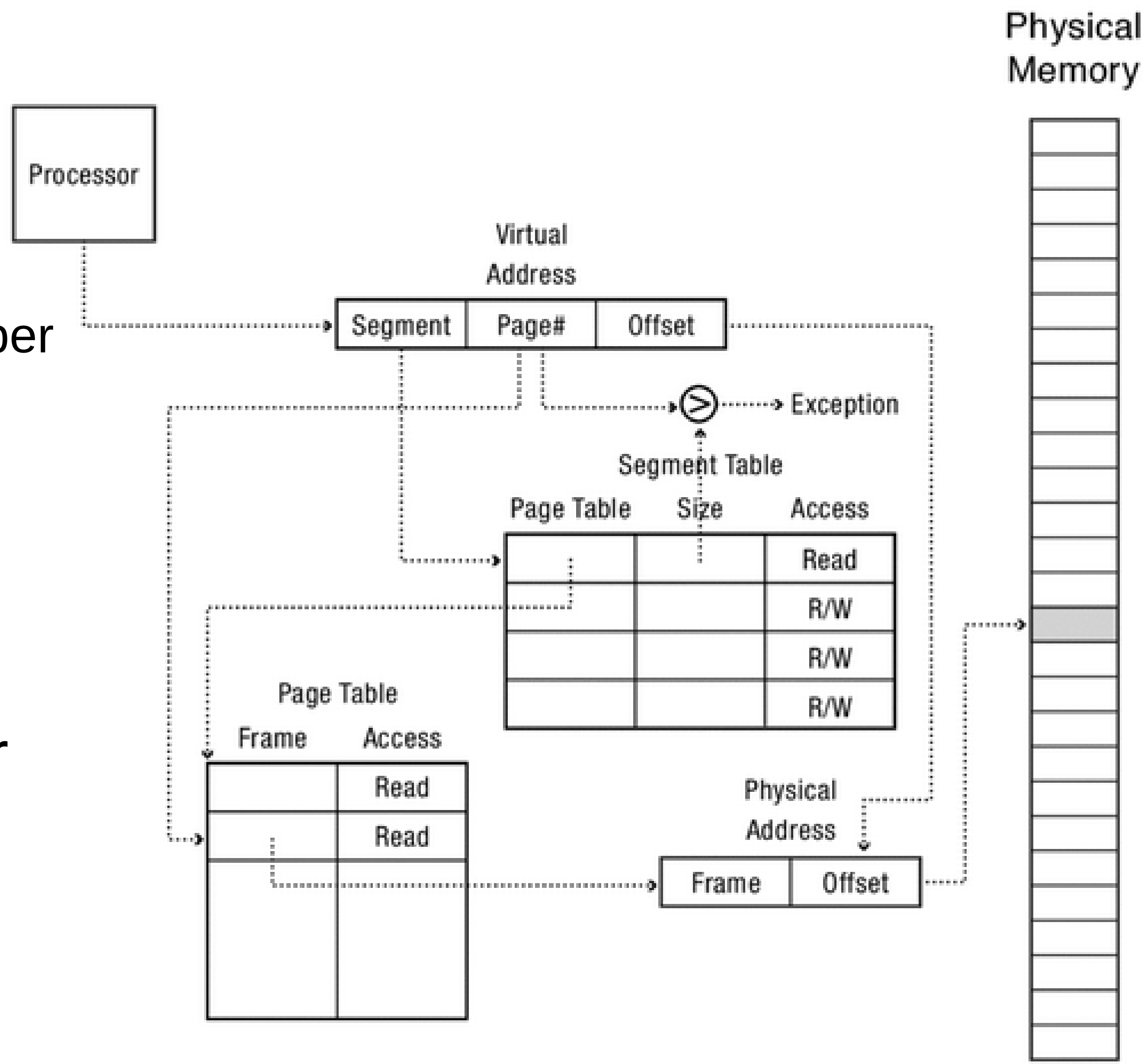
- Segmentation
  - Virtual address → segment# + offset
    - Max size of the segment → offset
    - Max memory per process → segment table size
    - Max real memory → base + offset

Where are the tables

- In memory
  - Special page/segment
- Multiple organizations
  - Page segmentation
  - Multi-level paging
  - Multi-level paged segmentation

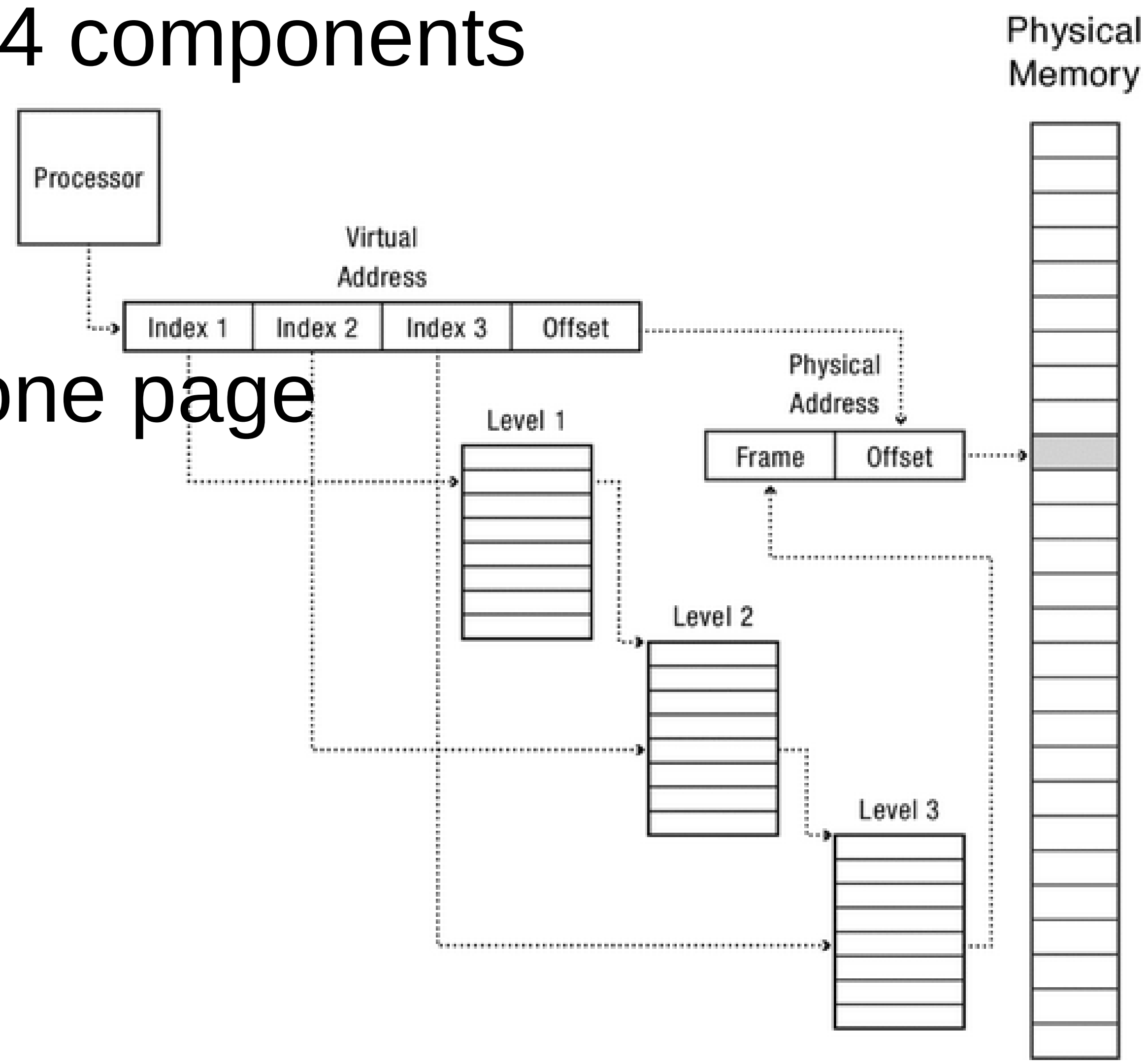
Page segmentation

- 32 bit address
- 4Kb pages
  - 10 bits
    - Segment number
  - 10 bits
    - Page number
  - 12 bits
    - Page offset
- 1 page table per segment
  - 4 bytes
  - 1K entries



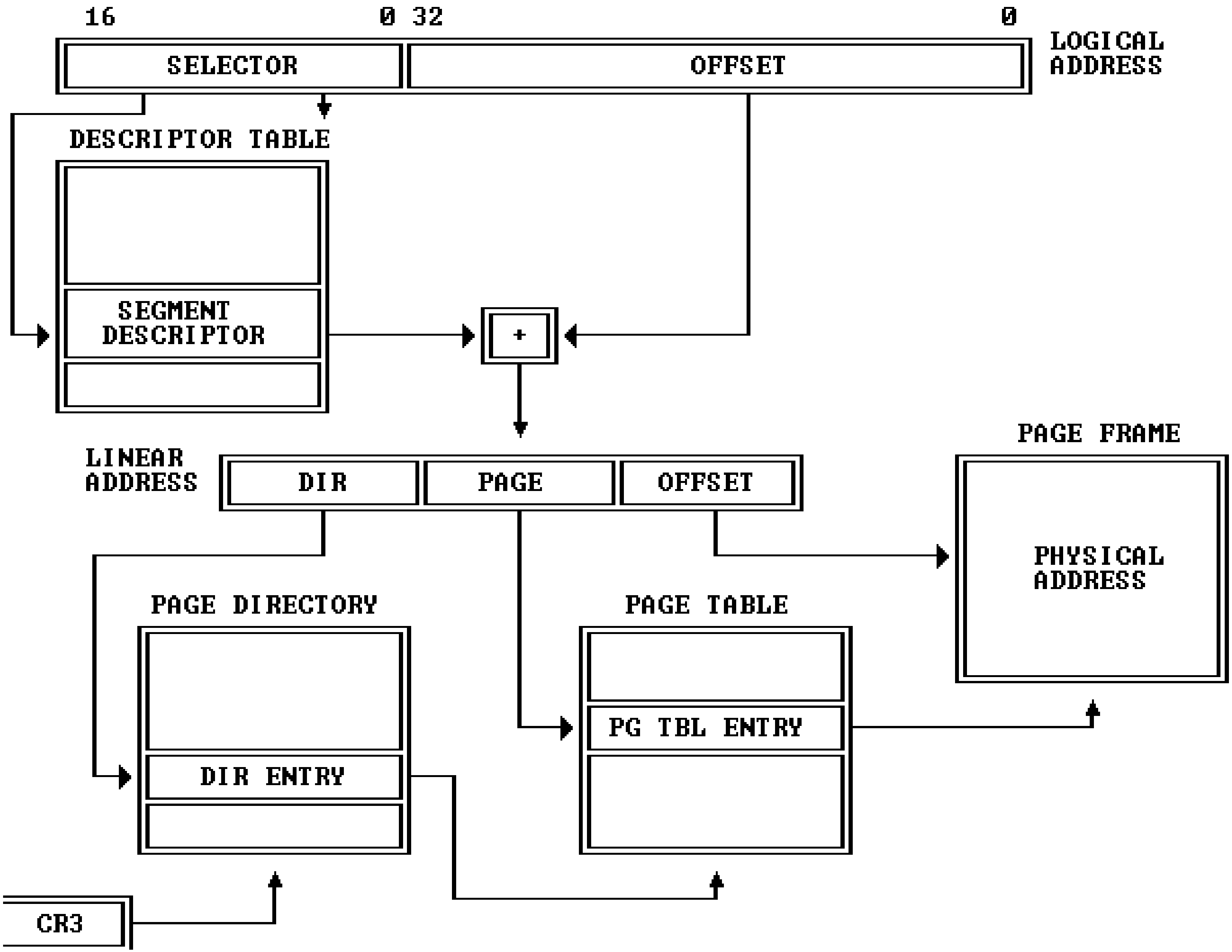
Multi-level paging

- Virtual address has 4 components
  - 3 indexes
  - Offset
- Each tabel fits into one page



Multi-level paged segmentation (X86)

igure 5-12. 80306 Addressing Machanism



32 bit - X86

- 2 level page table within a segment
  - 10 bits – page directory
  - 10 bit – second level page table
  - 12 bit – offset
- 4kb page frame
- Number of 2nd level page table
  - Depends on length of segment
  - Also has permissions

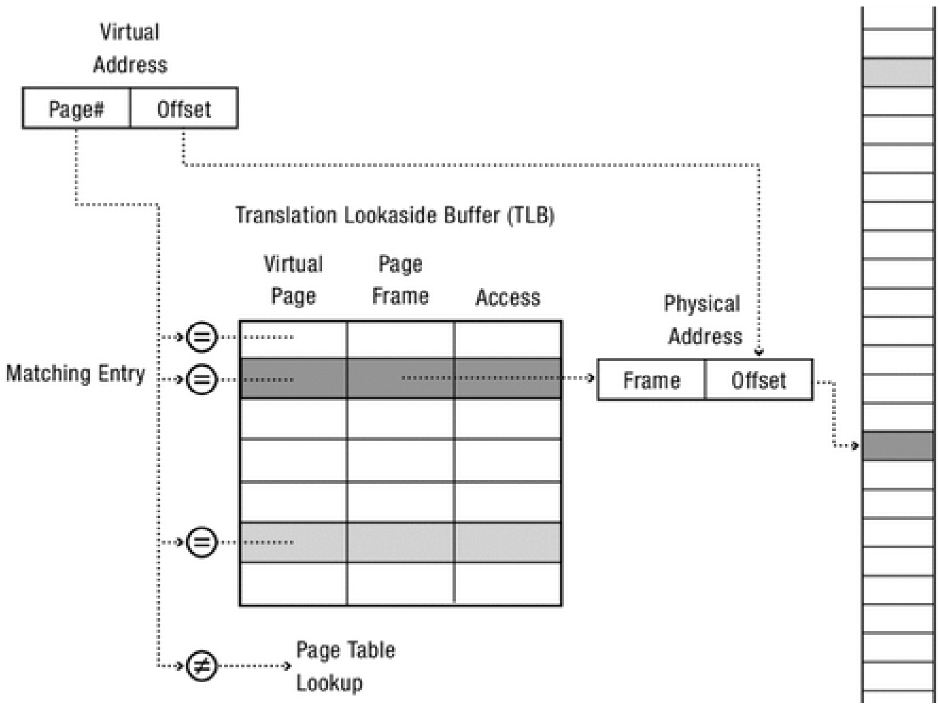
64 bit - X86

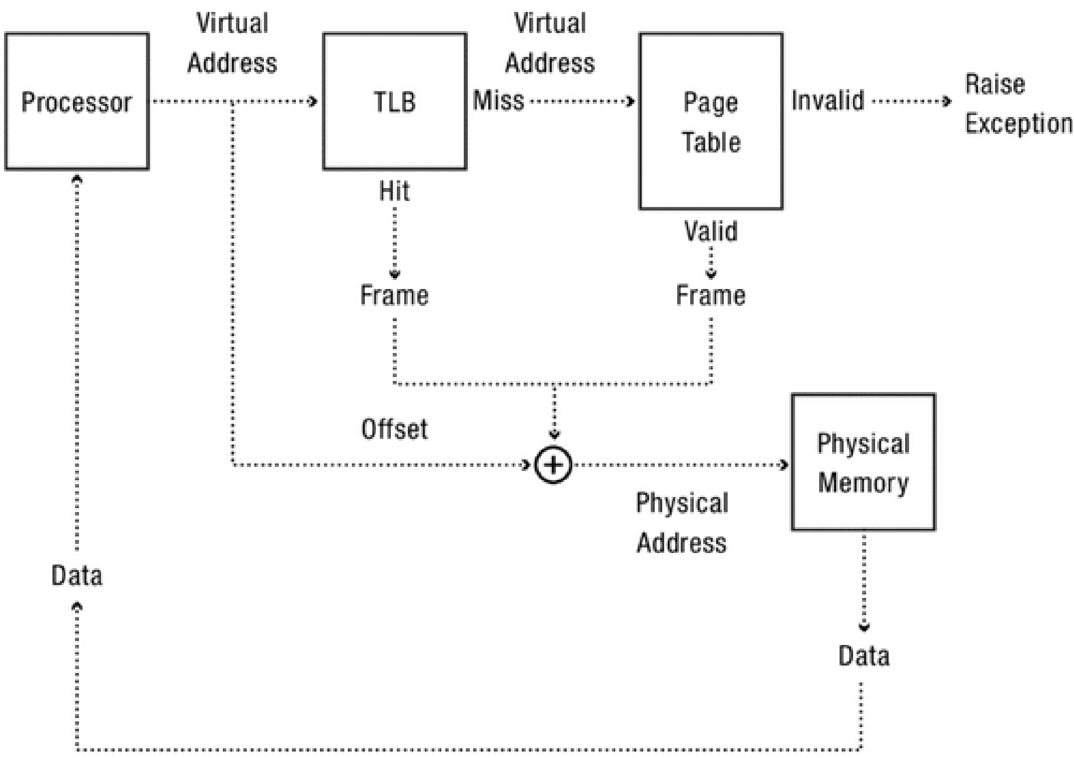
- 64 bits maximum address length
  - Optimization addresses only use 48 bit 128 terabytes
- 4 level page table within a segment
- Optimizations
  - Elimination of two levels of page tables
- 4kb page frame
  - 4th level pages → 2Mb
    - Can be overwriden if OS allocates 2Mb contiguos
  - 3rd level pages → 1Gb
    - Can be overwriden if OS allocates 1Gb contiguos
      - Good for server with two terabytes

Efficient Address translation

- Conversion from virtual to real address
  - Requires extra memory accesses
- Memory accesses are sequential
  - Repeated accesses to same page
  - Why not cache last translations?
- Translation look-aside buffers
  - Small HW table caching recent address translation

Translation look-aside buffers





TLB

- Extra HW
- Big savings in memory accesses
- Multiple levels of TLBs
- Requires extra care
  - Multiple processes
    - Different voirtual address space
    - Different permissions
  - Multiprocessors
    - Consistency of multiple TLB
      - one per CPU