OS for Database systems
Virtual Memory and IPC

Seehwan Yoo

Dankook University

Disclaimer: Some slides are borrowed from UC. Berkeley's 2014 OS and system programming

목차

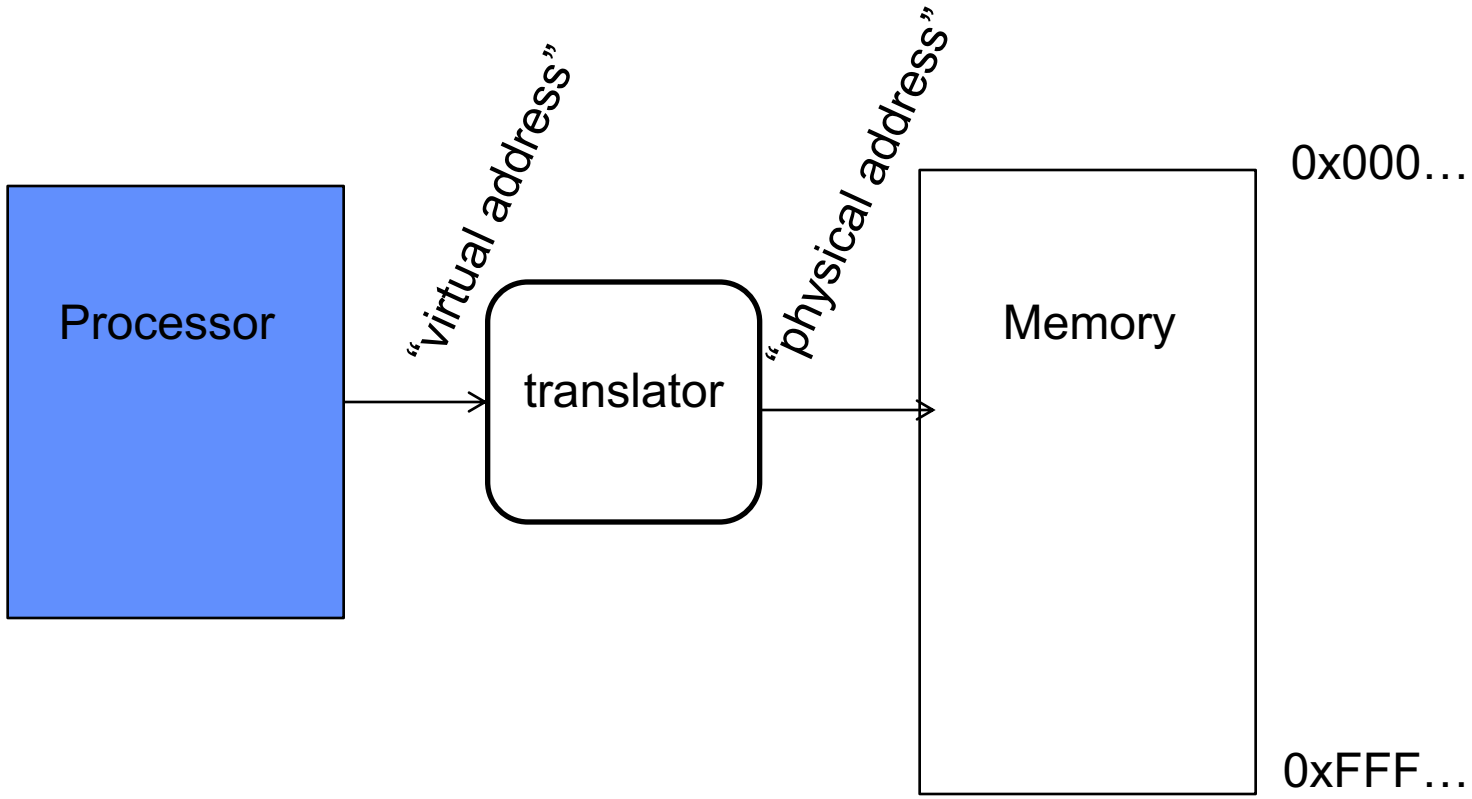
- 강의 개요
- OS 이론 강의 (8/29~9/2)
  - OS 개론, 프로세스. + mini 실습
  - 병렬성. + mini 실습
  - 메모리, IPC. + mini 실습
  - 파일시스템, 네트워킹. + mini 실습
  - 클라우드 컴퓨팅 및 시스템 관리
- OS 실습 강의 (9/19~9/23)
  - 리눅스 개발 환경 구축 및 쉘 프로그래밍 (1d)
  - 다중쓰레드 자료구조 실습 프로젝트 (2d)
  - AWS 실습 및 WordPress 기반 DB 실습 프로젝트 (2d)

Contents

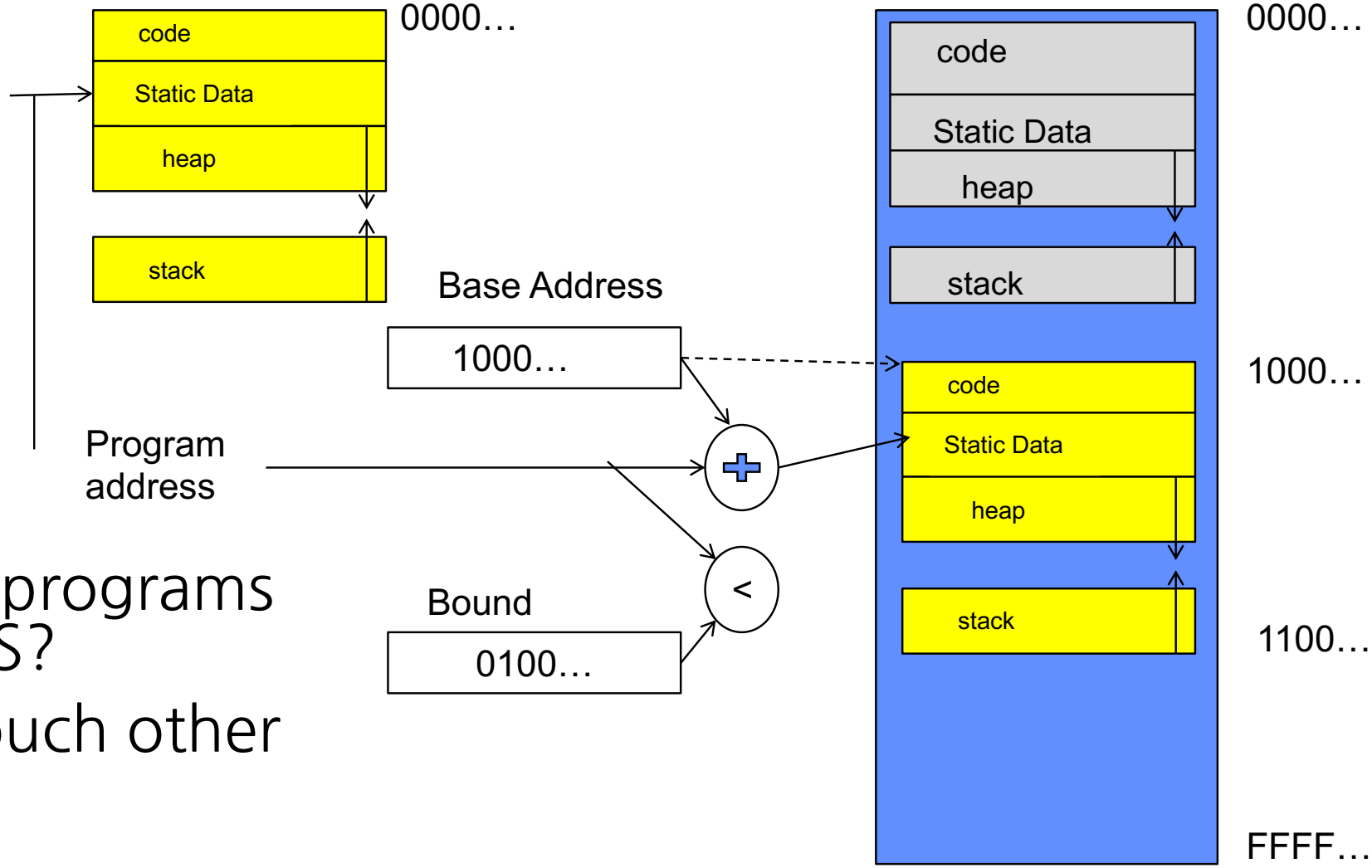
- Address space
- Segmentation & Base and Bound
- Paging and mapping
- Multi-level paging
- TLB, Caching
- Disk swapping
- IPC, signal, shm, msg_q

Key OS Concept: Address Space

- Program operates in an address space that is distinct from the physical memory space of the machine



A simple address translation: B&B



- Can the programs touch OS?
- Can it touch other pgms?

Virtual Memory

- Logical view of memory
  - Implementing per-process address space
  - Logical (memory) address != physical (memory) address
  - Independent memory for each process
- Benefits
  - Easy of programming
  - Better protection
  - Free from physical limitation

Implementing Virtualization

- Multiplexing
  - At one time, use this
  - At another time, use another
  - e.g.) Context switching
- Indirection
  - Logical to physical mapping
  - Some hardware support
  - e.g.) Virtual memory

~~Multiplexing~~ mapping memory

- Flat facts
  - Physical memory address is fixed
  - Processes have to share them
- Multiplexing memory is difficult
  - Too much!
  - Instead, mapping (indirection)
  - Controlled sharing (for communication)

Look into binary

- Objcode of gcd

```
Disassembly of section .text:

00000000 <main>:
   0:   27bdffd0      addiu   sp,sp,-48
   4:   afbf002c      sw      ra,44(sp)
   8:   afbe0028      sw      s8,40(sp)
   c:   03a0f021      move    s8,sp
  10:   24021298      li      v0,4760
  14:   afc20018      sw      v0,24(s8)
  18:   34029387      li      v0,0x9387
  1c:   afc2001c      sw      v0,28(s8)
  20:   8fc40018      lw      a0,24(s8)
  24:   8fc5001c      lw      a1,28(s8)
  28:   0c000000      jal     0 <main>
  2c:   00000000      nop
  30:   afc20020      sw      v0,32(s8)
  34:   03c0e821      move    sp,s8
  38:   8fbf002c      lw      ra,44(sp)
  3c:   8fbe0028      lw      s8,40(sp)
  40:   27bd0030      addiu   sp,sp,48
  44:   03e00008      jr      ra
  48:   00000000      nop

0000004c <gcd>:
  4c:   27bdffe0      addiu   sp,sp,-32
  50:   afbf001c      sw      ra,28(sp)
  54:   afbe0018      sw      s8,24(sp)
  58:   03a0f021      move    s8,sp
  5c:   afc40020      sw      a0,32(s8)
  60:   afc50024      sw      a1,36(s8)
  64:   8fc30020      lw      v1,32(s8)
  68:   8fc20024      lw      v0,36(s8)
  6c:   00000000      nop
  70:   14620004      bne     v1,v0,84 <gcd+0x38>
...
  d8:   8fc50020      lw      a1,32(s8)
  dc:   0c000000      jal     0 <main>
  e0:   00000000      nop
  e4:   03c0e821      move    sp,s8
  e8:   8fbf001c      lw      ra,28(sp)
  ec:   8fbe0018      lw      s8,24(sp)
  f0:   27bd0020      addiu   sp,sp,32
  f4:   03e00008      jr      ra
  f8:   00000000      nop
  fc:   00000000      nop
```

Source code of gcd

- Where is gcd() call in the binary?

```
void main()
{
        int a = 0x1298;
        int b = 0x09387;
        int res;

        res = gcd (a, b);
}

int gcd(int a, int b)
{
        if (a==b) return a;
        else if (a>b) return gcd(a-b, b);
        else return gcd(b-a, a);
}
```

Why calling main?

- GCD binary code

```
Disassembly of section .text:

00000000 <main>:
  0:   27bdffd0      addiu   sp,sp,-48
  4:   afbf002c      sw      ra,44(sp)
  8:   afbe0028      sw      s8,40(sp)
  c:   03a0f021      move    s8,sp
 10:   24021298      li      v0,4760
 14:   afc20018      sw      v0,24(s8)
 18:   34029387      li      v0,0x9387
 1c:   afc2001c      sw      v0,28(s8)
 20:   8fc40018      lw      a0,24(s8)
 24:   8fc5001c      lw      a1,28(s8)
 28:   0c000000      jal     0 <main>
 2c:   00000000      nop
 30:   afc20020      sw      v0,32(s8)
 34:   03c0e821      move    sp,s8
 38:   8fbf002c      lw      ra,44(sp)
 3c:   8fbe0028      lw      s8,40(sp)
 40:   27bd0030      addiu   sp,sp,48
 44:   03e00008      jr      ra
 48:   00000000      nop
```

```
0000004c <gcd>:
  4c:   27bdffe0      addiu   sp,sp,-32
  50:   afbf001c      sw      ra,28(sp)
  54:   afbe0018      sw      s8,24(sp)
  58:   03a0f021      move    s8,sp
  5c:   afc40020      sw      a0,32(s8)
  60:   afc50024      sw      a1,36(s8)
  64:   8fc30020      lw      v1,32(s8)
  68:   8fc20024      lw      v0,36(s8)
  6c:   00000000      nop
  70:   14620004      bne     v1,v0,84 <gcd+0x38>
...
  d8:   8fc50020      lw      a1,32(s8)
  dc:   0c000000      jal     0 <main>
  e0:   00000000      nop
  e4:   03c0e821      move    sp,s8
  e8:   8fbf001c      lw      ra,28(sp)
  ec:   8fbe0018      lw      s8,24(sp)
  f0:   27bd0020      addiu   sp,sp,32
  f4:   03e00008      jr      ra
  f8:   00000000      nop
  fc:   00000000      nop
```

Linking & loading

- At compile time, we cannot know target address of a function
- Compile
  - Language translation from high-level PL. to low-level PL. (executable on CPU)
- Linking
  - Resolving address of unknown symbol used in the program
- Loading
  - Locating the code/data from permanent storage to memory (designated address)

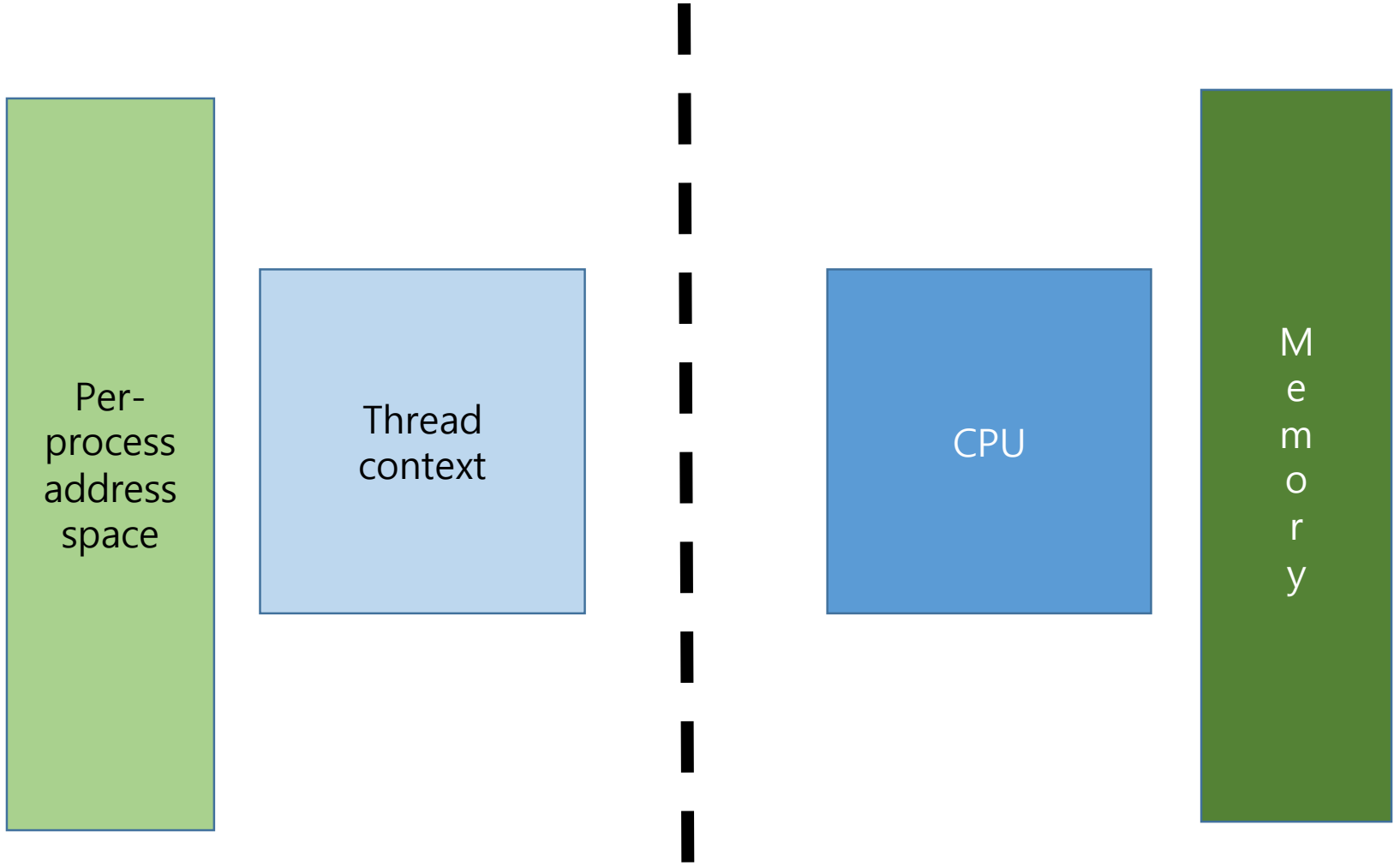
After linking,

- 00400610 <main>:
- 400610: 27bdffd0 addiu sp,sp,-48
- 400614: afbf002c sw ra,44(sp)
- 400618: afbe0028 sw s8,40(sp)
- 40061c: 03a0f025 move s8,sp
- 400620: 24021298 li v0,4760
- 400624: afc20018 sw v0,24(s8)
- 400628: 34029387 li v0,0x9387
- 40062c: afc2001c sw v0,28(s8)
- 400630: 8fc40018 lw a0,24(s8)
- 400634: 8fc5001c lw a1,28(s8)
- **400638: 0c100197 jal 40065c <gcd>**
- 40063c: 00000000 nop
- 400640: afc20020 sw v0,32(s8)
- 400644: 03c0e825 move sp,s8
- 400648: 8fbf002c lw ra,44(sp)
- 40064c: 8fbe0028 lw s8,40(sp)
- 400650: 27bd0030 addiu sp,sp,48
- 400654: 03e00008 jr ra
- 400658: 00000000 nop

- **0040065c <gcd>:**
- 40065c: 27bdffe0 addiu sp,sp,-32
- 400660: afbf001c sw ra,28(sp)

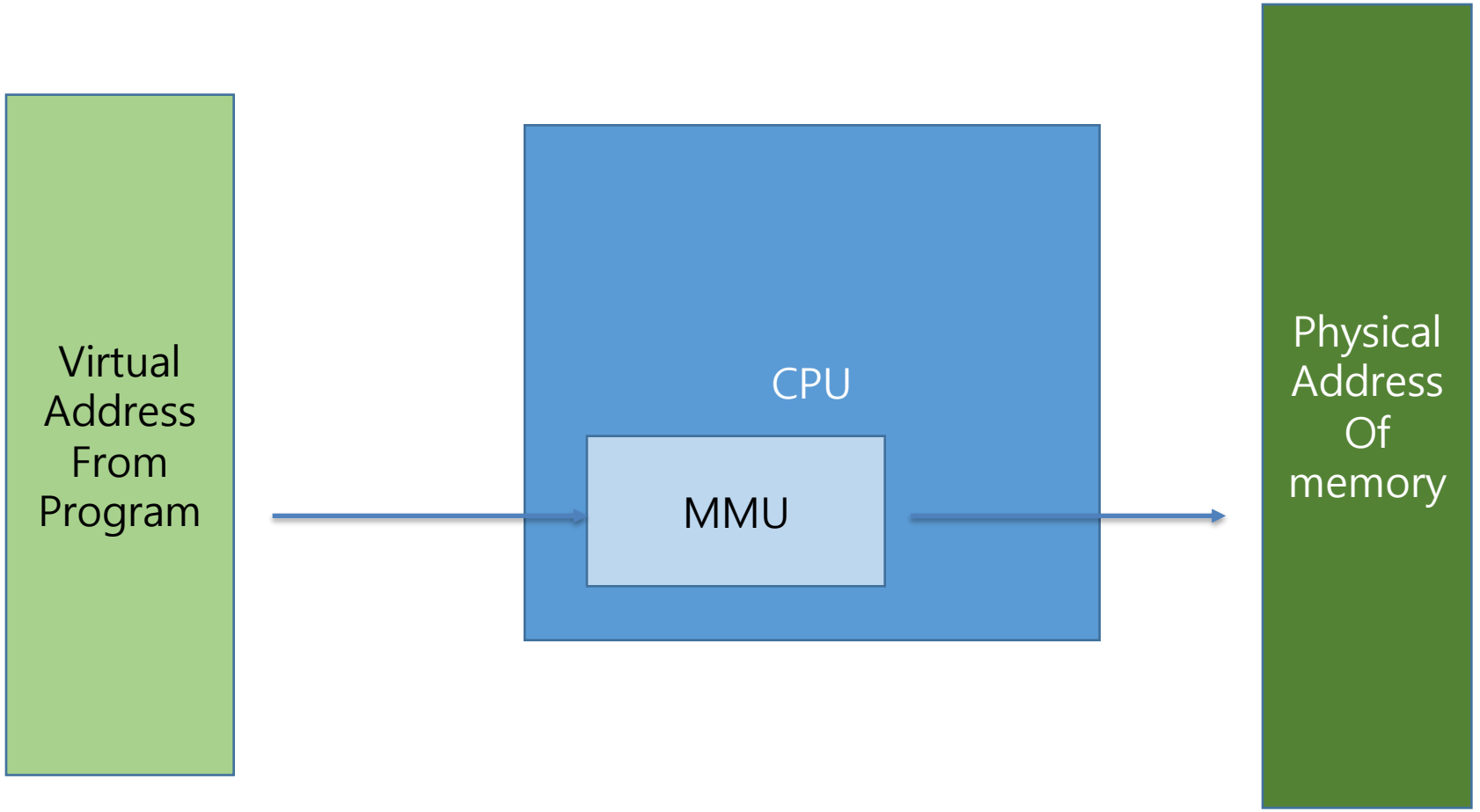
- 4006b0: 00402025 move a0,v0
- 4006b4: 8fc50024 lw a1,36(s8)
- **4006b8: 0c100197 jal 40065c <gcd>**
- 4006bc: 00000000 nop
- 4006c0: 10000008 b 4006e4 <gcd+0x88>
- 4006c4: 00000000 nop
- 4006c8: 8fc30024 lw v1,36(s8)
- 4006cc: 8fc20020 lw v0,32(s8)
- 4006d0: 00621023 subu v0,v1,v0
- 4006d4: 00402025 move a0,v0
- 4006d8: 8fc50020 lw a1,32(s8)
- **4006dc: 0c100197 jal 40065c <gcd>**
- 4006e0: 00000000 nop
- 4006e4: 03c0e825 move sp,s8
- 4006e8: 8fbf001c lw ra,28(sp)
- 4006ec: 8fbe0018 lw s8,24(sp)
- 4006f0: 27bd0020 addiu sp,sp,32
- 4006f4: 03e00008 jr ra

Logical view from CPU side



MMU: A big picture

- VA-PA translation hardware inside CPU



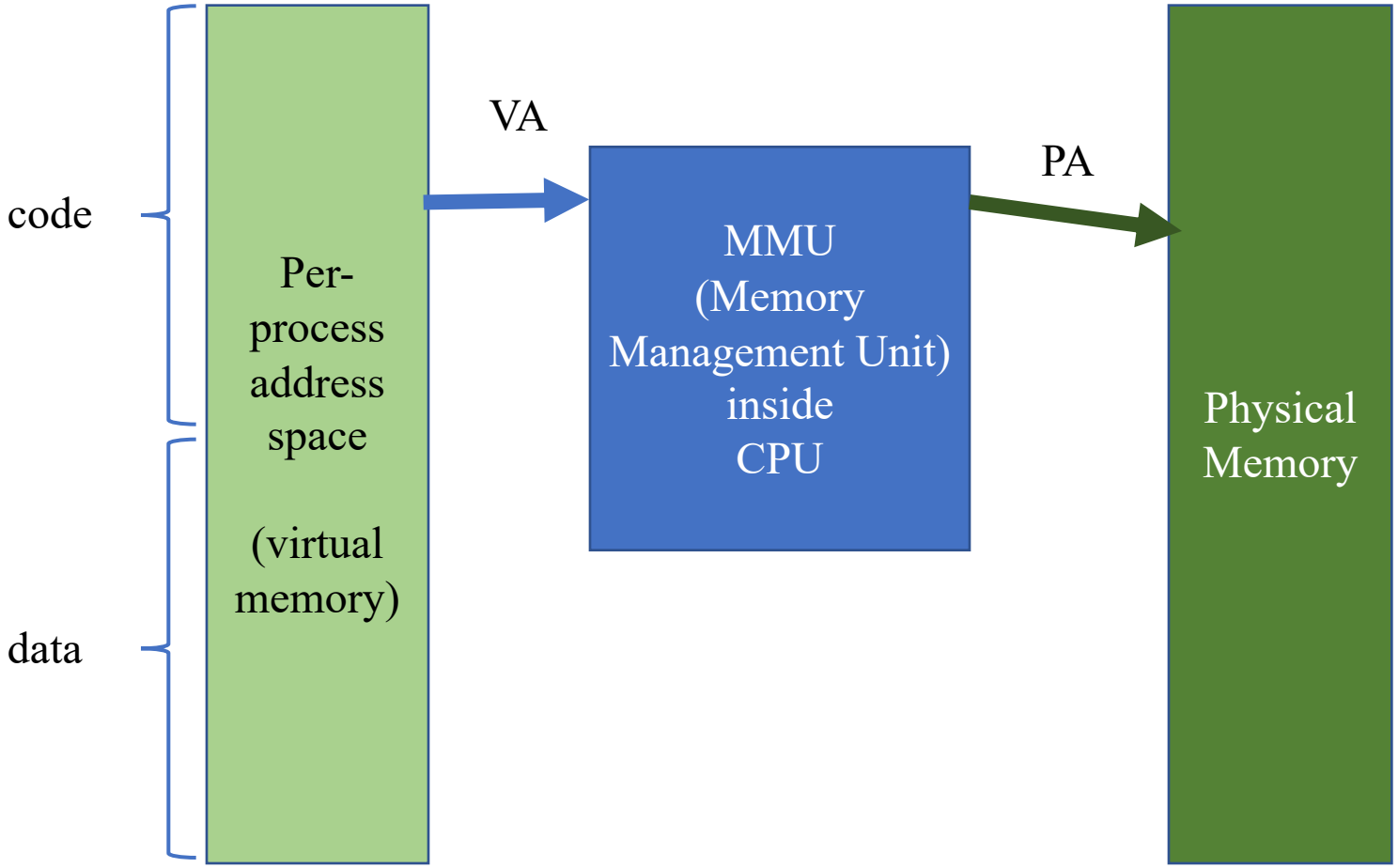
How OS is involved

- OS sets up MMU
  - For different process,
    different address space
    different mapping (VA-PA)
- Mapping table
  - VA-PA mapping
  - Will be addressed in paging

Memory manager with H/W support

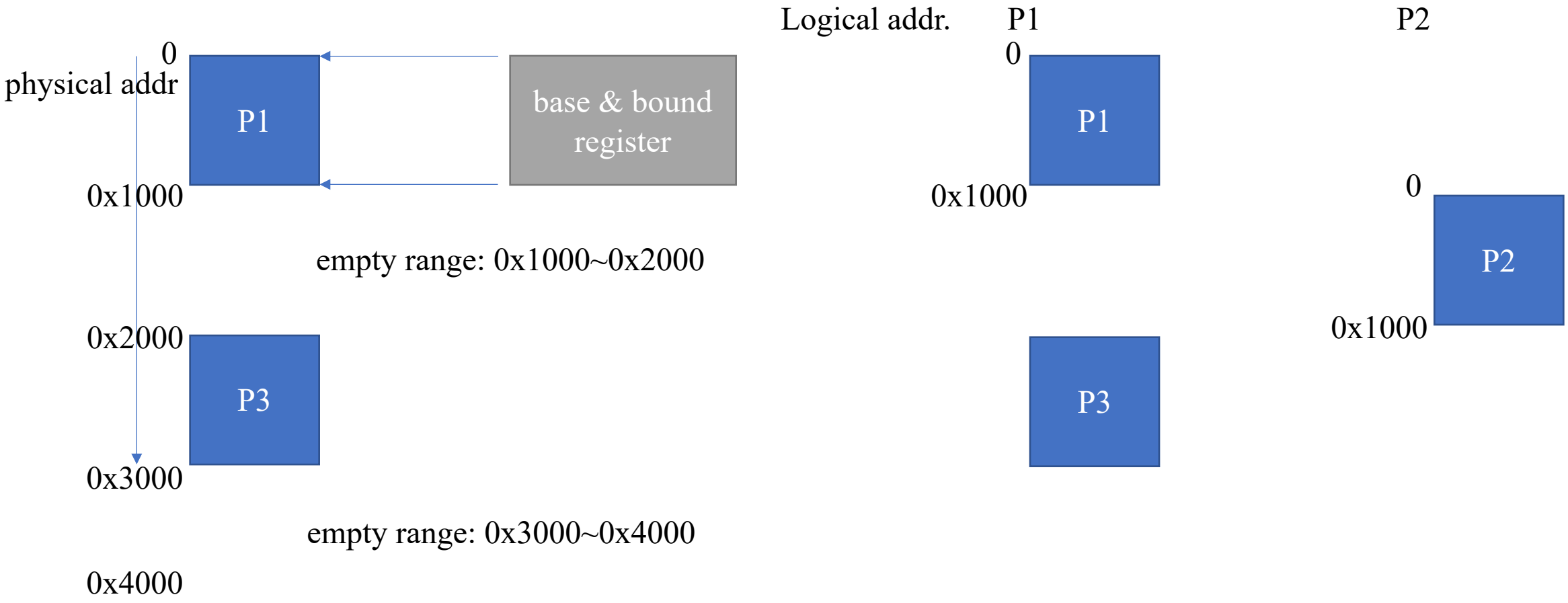
- Architectural support for OS
  - Base/bound registers for CRAY
    - OS sets up base & bound register
    - When context switches, base/bound registers are re-programmed (re-loaded)
  - Segmentation of Intel CPUs
    - Segments: part of program (code, data, stack)
    - Segment registers: point the starting address of segment
    - OS manages segments table

A slightly more detailed picture



Base & Bound Segmentation

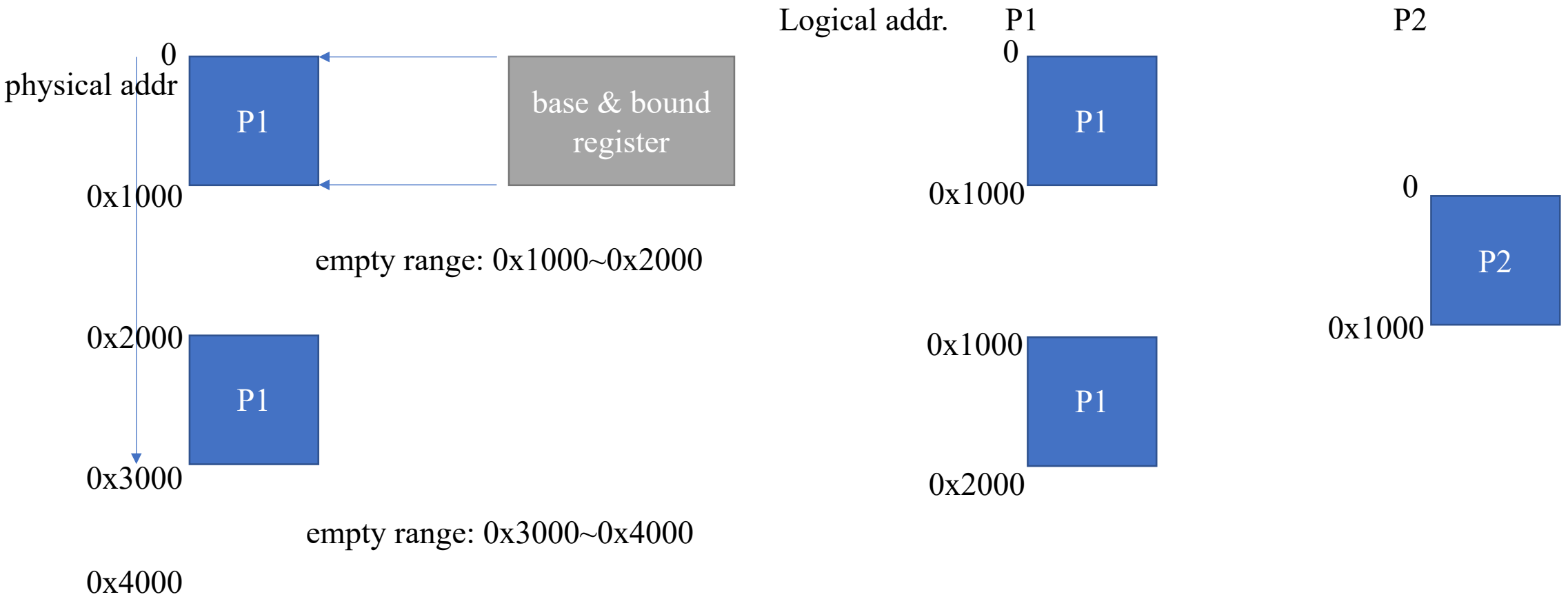
Limitation of B&B
→ only allows contiguous memory region



Fragmentation

total empty region size: 0x2000 = 8KB, however we cannot allocate P4, which has 5KB memory region

Limitation of B&B
→ only allows contiguous memory region

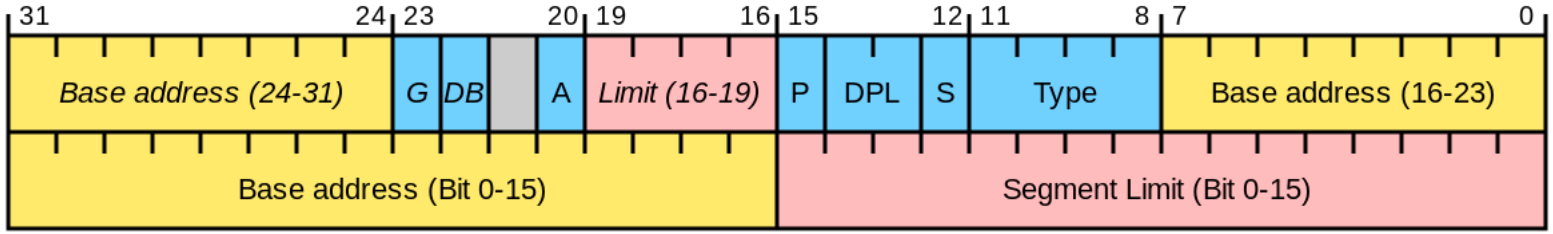


Fragmentation

total empty region size: 0x2000 = 8KB, however we cannot allocate P4, which has 5KB memory region

Segmentation

- Extend idea from Base & Bound (B&B)
- Divide address space into logical regions
  - Code segment (CS), Data segment (DS), Stack segment (SS), Extra segment (ES), for general use
  - FS, GS: canary-based stack protector, thread-local storage
- For Intel specific implementation,
  - Segment descriptor: base & bound of a segment is stored in memory
  - All segments descriptors are in the 'descriptor table'
  - Now, you have to know, which segment is used



VA-to-PA mapping with segmentation

- Prepare segment descriptors
- Enable segmentation by programming *segment register* (segment selector)
  - CS = index of code segment descriptor (CS is a segment selector)
  - DS = index of data segment descriptor
- Memory access
  - All code access
    → VA = (CS_i:base + addr)
    → Linear address
  - All data access
    → VA = (DS_i:base + addr)
    → Linear address

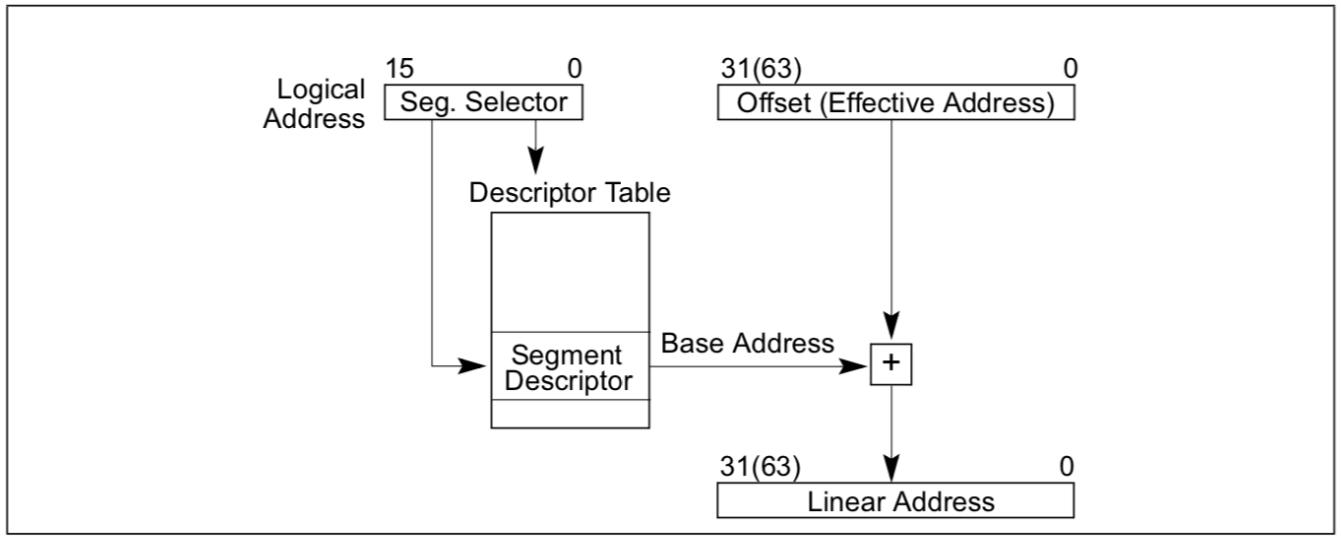


Figure 3-5. Logical Address to Linear Address Translation

Paging and Address translation architecture

Paging

- Small, equal sized mapping between physical memory region into address space
  - Usually, 4KB sized mapping
- Page
  - Fragment of address space, each of which has the small, equal sized region
  - Page number: index to all pages
  - VA: (page #) << 12 | (VA & 0xFFF)
- Page frame
  - Fragment of physical memory
  - Each of which has the small, equal sized region
  - Pfn (page frame number): index to all page frames
  - PA: (pfn << 12) | (PA & 0xFFF)

Mapping from VA-to-PA using paging

- *Page table* has a mapping between (page number) and (page frame number)
- VA = (page number) : (page offset)
- PA → Page table(page number) : (page offset)
  → (page frame number) : (page offset)
- Page table entry

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Address of page directory[1]																				Ignored							PCD	PWT	Ignored			CR3
Bits 31:22 of address of 4MB page frame										Reserved (must be 0)					Bits 39:32 of address[2]				PAT	Ignored			G	1	D	A	PCD	PWT	U/S	R/W	1	PDE: 4MB page
Address of page table																				Ignored				0	Ign	A	PCD	PWT	U/S	R/W	1	PDE: page table
Ignored																															0	PDE: not present
Address of 4KB page frame																				Ignored				G	PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page
Ignored																															0	PTE: not present

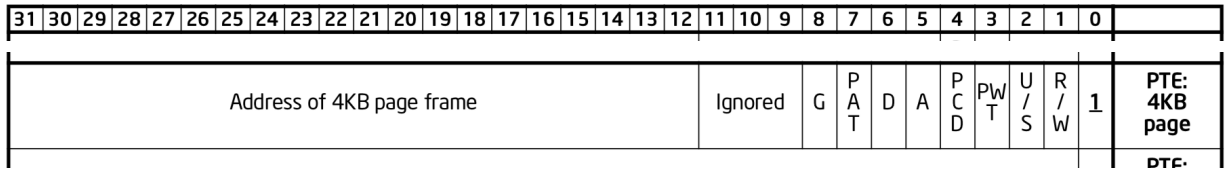
Figure 4-4. Formats of CR3 and Paging-Structure Entries with 32-Bit Paging

Page	pfn
0	0x1234
1	0x1235
2	0x1236
…	
0x100	0x1237
0xFFF	0xFFF
…	…
0x80000	0x0
…	…
0xFFFFF	0x4000

Mapping from VA-to-PA using paging

- *Page table* has a mapping between (page number) and (page frame number)
- VA = (page number) : (page offset)
- PA → Page table(page number) : (page offset)
  → (page frame number) : (page offset)
- Page number is used as index into the page table
  - We don't need to store them!
- Page table entry
  - 4-byte size: high bits have pfn

Page	pfn
0	0x1234
1	0x1235
2	0x1236
…	
0x100	0x1237
0xFFF	0xFFF
…	…
0x80000	0x0
…	…
0xFFFFF	0x4000



Address Translation: how to

- Needs a translation table (page table)
  - Remembers VA-to-PA mapping
  - Use some bits in VA as index
  - Translation table is in memory
- Needs architectural support
  - MMU is introduced
  - TTBR (translation table base register) points to the page table
  - ISA changed (H/W, S/W change required)
- Needs a free page list
  - OS keeps track of free pages in the entire memory

Page table looks like this

- PA[31:12] + some more bits for the page
- Translation table holds many entries of the following descriptors

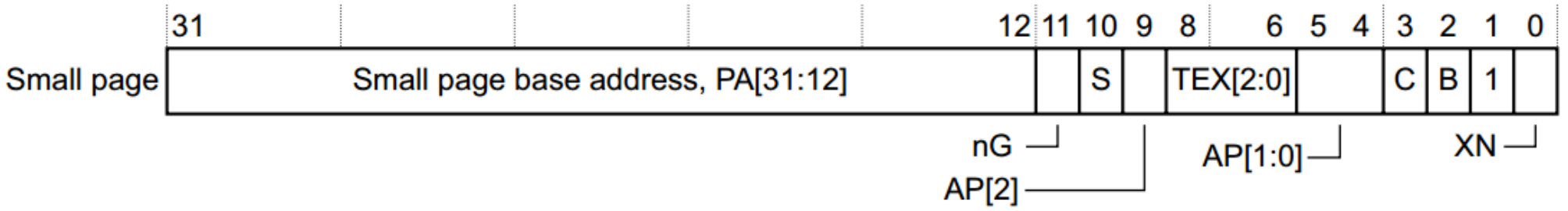
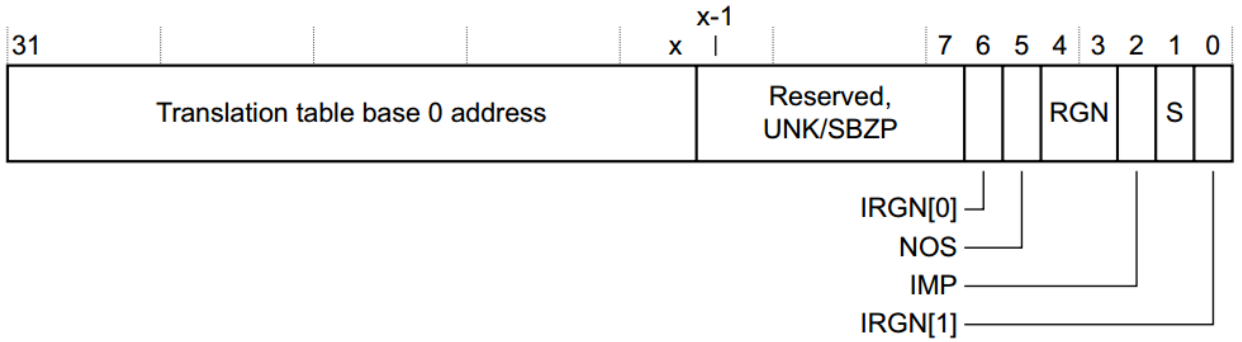


Figure B3-5 Short-descriptor second-level descriptor formats

ARM architecture reference manual ARMv7-A/R,
http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0333h/I1029222.html

TTBR: translation table base register

- OS sets up the page table, before it is used
- OS writes TTBR register to change page table
  - When context switching occurs



B4.1.154 TTBR0, Translation Table Base Register 0, VMSA

The TTBR0 characteristics are:

Purpose

TTBR0 holds the base address of translation table 0, and information about the memory it occupies. This is one of the translation tables for the stage 1 translation of memory accesses from modes other than Hyp mode.

This register is part of the Virtual memory control registers functional group.

How large is page table?

- Each entry is 4 bytes-sized (32 bit)
- Each entry maps 4KB for small pages
  - How many entries for 32bit address space?
  - $2^{32}/2^{12} = 2^{20}$ ~1M entries
- 1M entries x 4 bytes for each
  - 4MB per process (1024 page frames are required)
  - Page table has to be contiguous
- Too large!
- Many of them are not used, but takes space!
- Many entries are repeated for multiple processes!
  - Kernel region (1G/2G/3G) is the same for all of the processes

About the page table size

- Page table's single entry size: 4 byte (32 bits)
- How many mapping exists in a process?
  - Entire address space: 4GB for 32 bit machine (2^{32})
  - Each mapping has 4KB (2^{12}) address space, physical page frames
  - $2^{32} / 2^{12} = 2^{20} = 1M$
- Page table size = (# of entries) * (entry size) = 1M * 4B = 4MB.
  - Too large for many-process systems
  - 100 user processes = 400 MB of page tables
    → needs further reduction!

Page table size reduction

- Three observations
  - Observation one: Not all address space require mapping
    - There are numerous small programs, only utilizes several kilobytes of memory
  - Observation two: many programs share some memory (code / data)
    - Kernel, libc, shared library, JVM
  - Observation three: larger mapping reduces the mapping table size
    - What if 4MB (2^{22}) mapping is used?
    - Entry size: 4B
    - (# of entries in a page table): 2^{32} / 2^{22} = 2^{10}
    - Page table size: 4KB!, same with the size of page frame!

Two-level paging

- Make a coarse grained mapping for some regions
  - For empty (not utilized) mapping
  - For (large) shared mapping
- How about the small mappings?
  - Add another level indirection
  - Coarse-grained mapping points to the next level page table
- *Page directory*
  - Coarse-grained (4MB) mapping table, size?
  - # of entries: 4GB(=2^{32}) / 4MB (2^{22}) = 2^{10} = 1024 entries
- *Page table*
  - 4KB-size small mapping table, size?
  - # of entries: 4MB(=2^{22}) / 4KB (2^{12}) = 2^{10} = 1024 entries

Two-level paging

- Make levels
- First level
  - Each entry represents coarse grained mapping
- Second level
  - Each entry represents fine-grained mapping

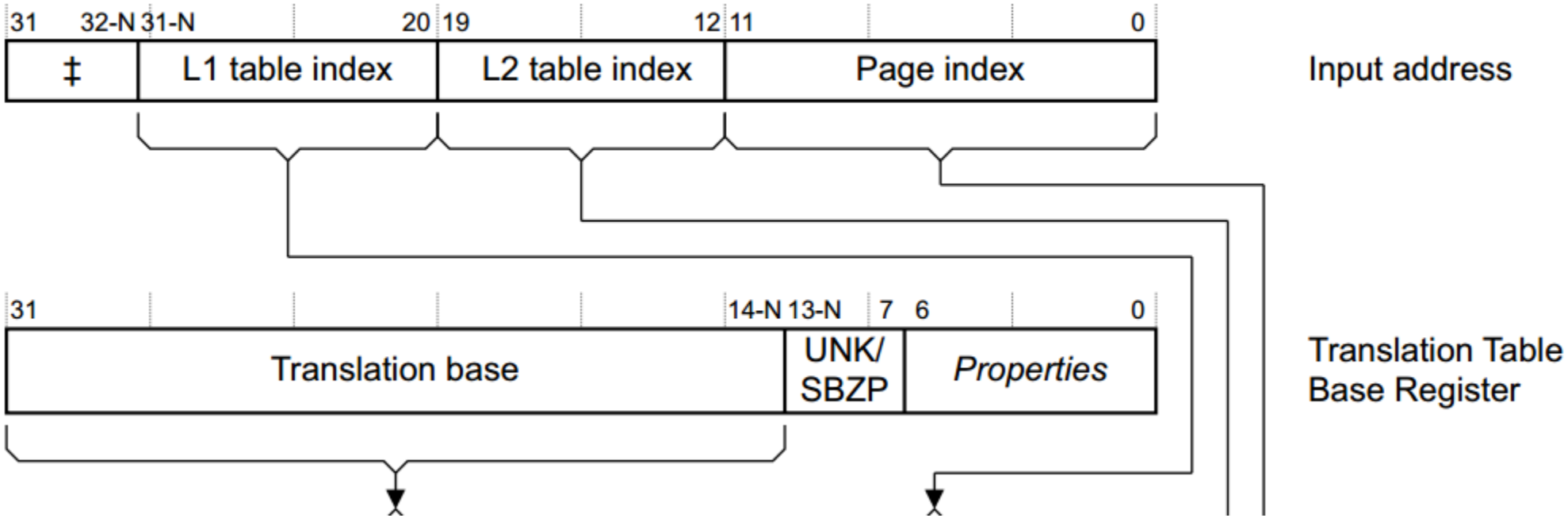
Small page A 4KB memory region, described by the combination of:

- a first-level translation table entry that indicates a second-level Page table address
- a second-level descriptor that indicates a Small page.

See *Translation flow for a Small page* on page B3-1337.

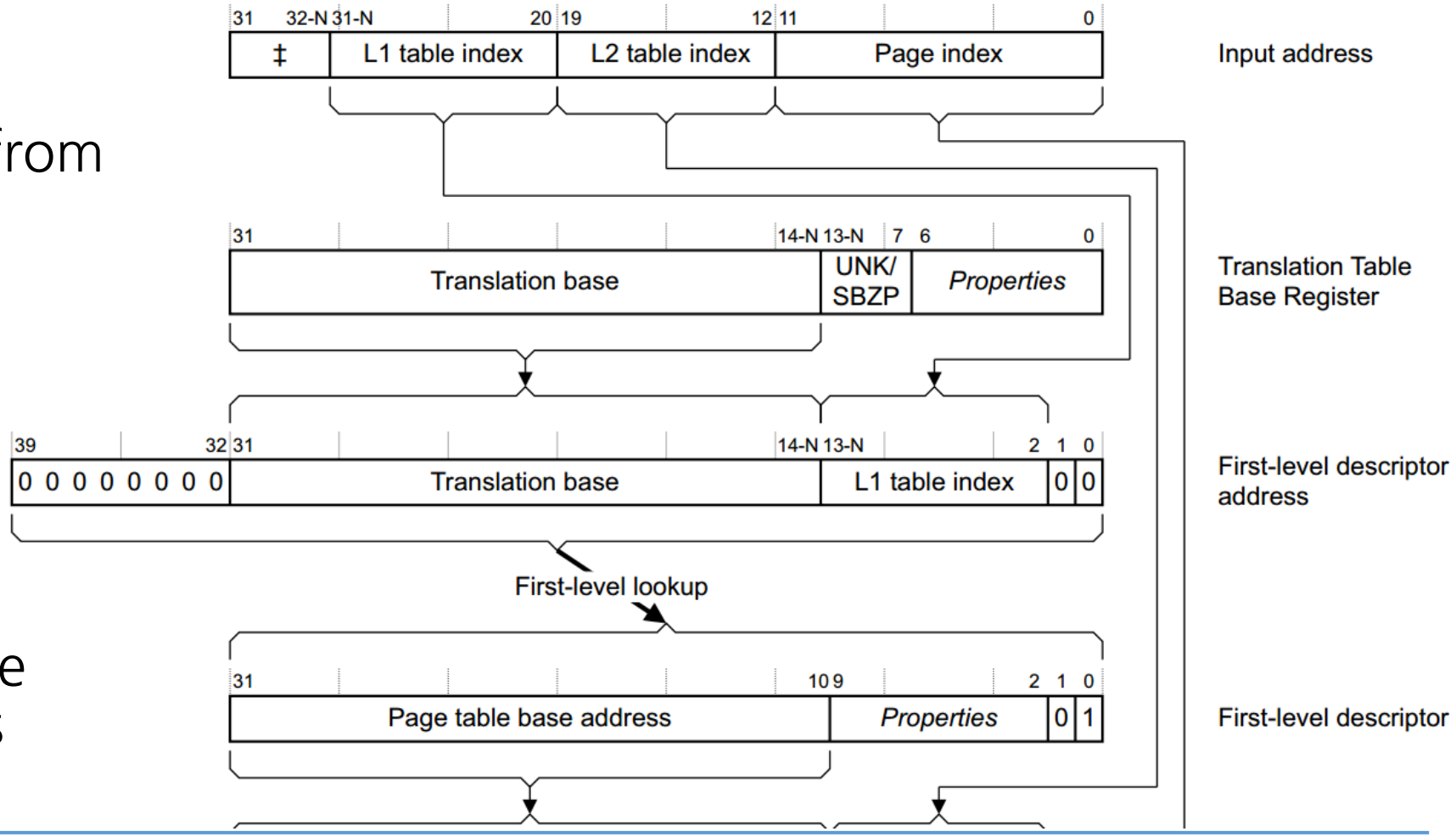
Two-level paging: step 1

- Read TTBR (finds the page table)



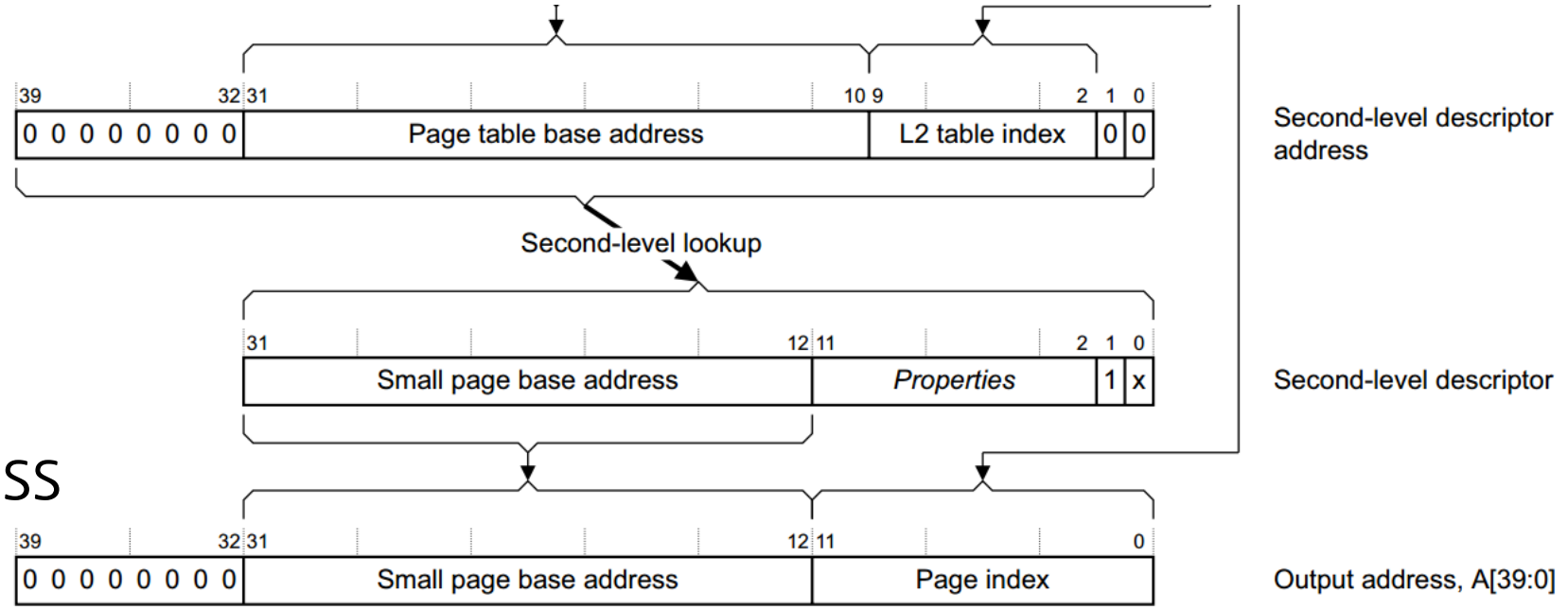
Two-level paging: step 2

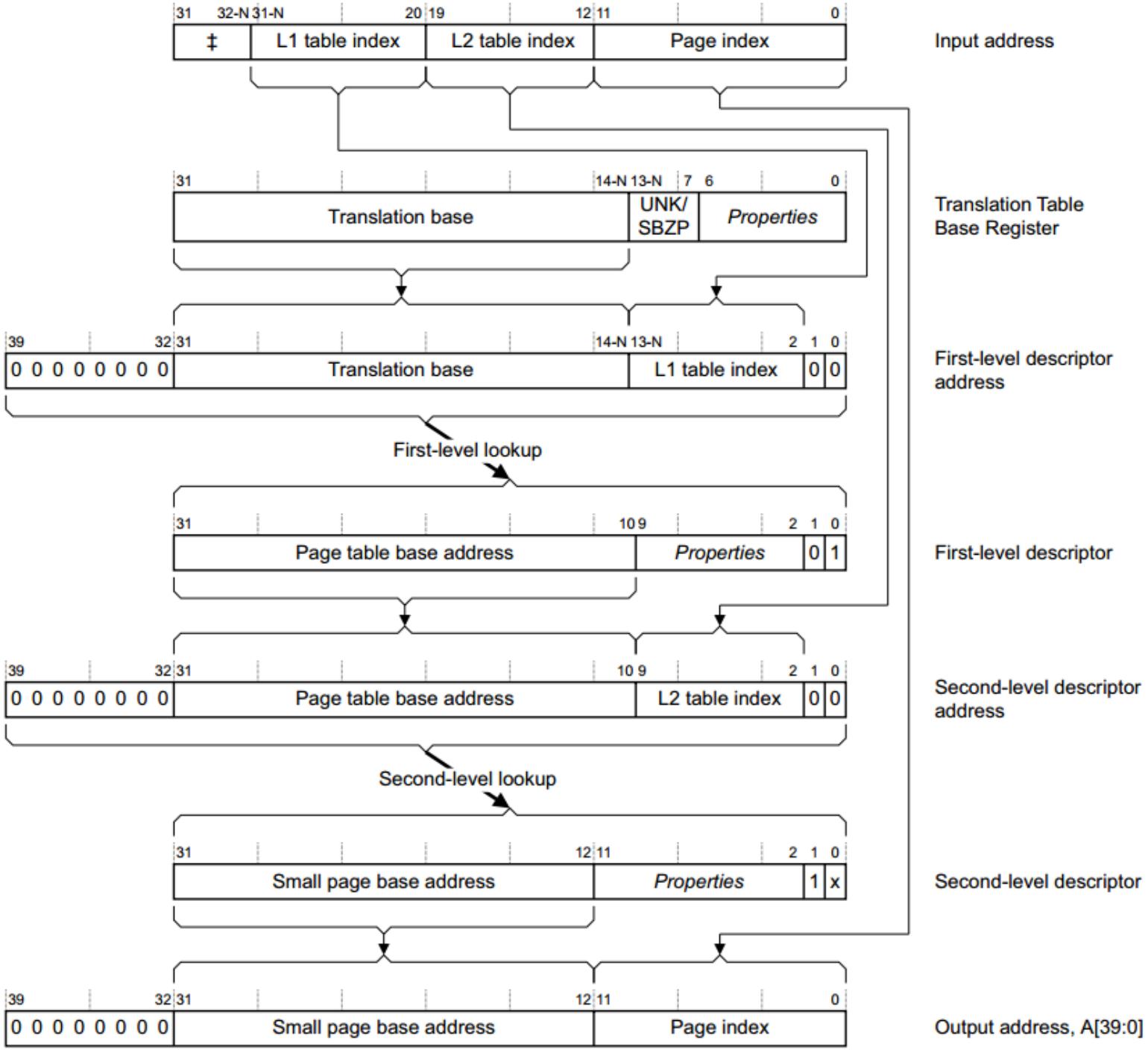
- First-level lookup
  - From TTBR addr,
  - Get L1 table index from VA
  - Look at Pagetable [L1_index(VA)]
  - Pagetable addr + L1_index(VA)*4
  - → second-level table address + some bits



Two-level paging step 3

- From second-level page table, L2_index(VA)
- Look at L2_pagetable [L2_index(VA)]
- addr + L2_index(VA)*4
- → page frame address + some bits




31 32-N 31-N 20 19 12 11 0
‡
L1 table index
L2 table index
Page index
Input address
31 14-N 13-N 7 6 0
Translation base
UNK/ SBZP
Properties
Translation Table Base Register
39 32 31 14-N 13-N 2 1 0
0 0 0 0 0 0 0 0
Translation base
L1 table index
0 0
First-level descriptor address
First-level lookup
31 10 9 2 1 0
Page table base address
Properties
0 1
First-level descriptor
39 32 31 10 9 2 1 0
0 0 0 0 0 0 0 0
Page table base address
L2 table index
0 0
Second-level descriptor address
Second-level lookup
31 12 11 2 1 0
Small page base address
Properties
1 x
Second-level descriptor
39 32 31 12 11 0
0 0 0 0 0 0 0 0
Small page base address
Page index
Output address, A[39:0]

IA-32 paging structure

- CR3 is a special register pointing to the page directory

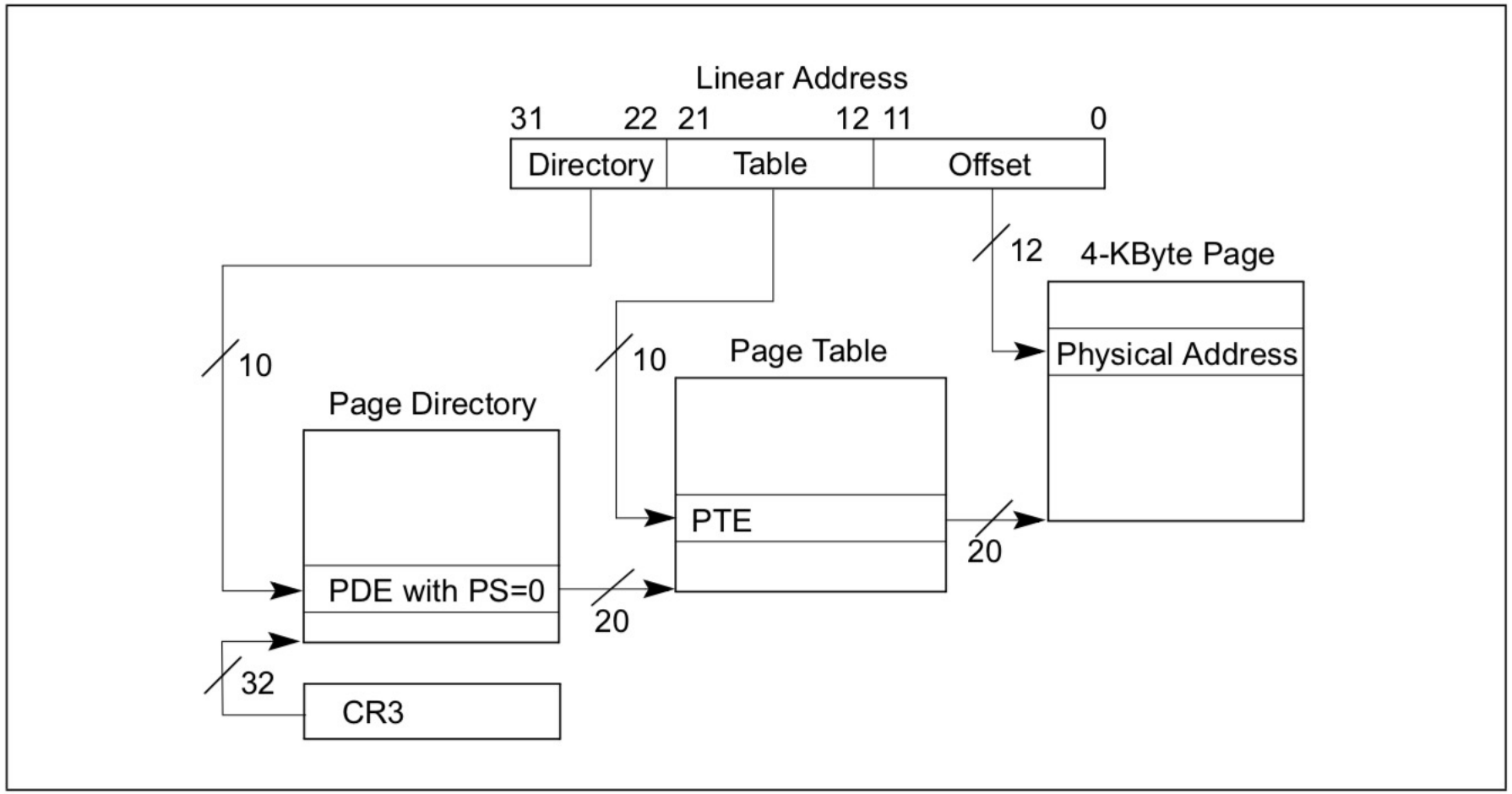


Figure 4-2. Linear-Address Translation to a 4-KByte Page using 32-Bit Paging

Some more: demand paging

- Page frames: at the beginning, not all the page frames are used
  - In fact, only a few page frames are required
  - Use only the frames that is actually touched
    - Allocate page frames on-demand
- Initially, page table is empty
  - When first accessed, OS finds a free frame, maps it onto the requested virtual address
    - And you have to re-execute the instruction
  - Page fault
    - Occurs when the page table is invalid

Demand paging: Filling in the page table, on-demand

- Page table entry is empty/invalid → CPU cannot find proper physical page frame
  - Cannot make progress
- Page fault is an exception, CPU execution enters OS
  - Pre-defined exception handler is invoked
    - Page fault handler
- Page fault handler finds a new free page frame
  - Map the page onto the address space
  - Filling in the page table
  - Do the instruction again

Pros and Cons of two level paging

- Pros
  - Small page tables
    - Better allocation
  - Reduce wasted (null) entries
  - Easy sharing
- Cons
  - Slow translation
    - Additional memory access for single translation

Mitigate slow two-level translation

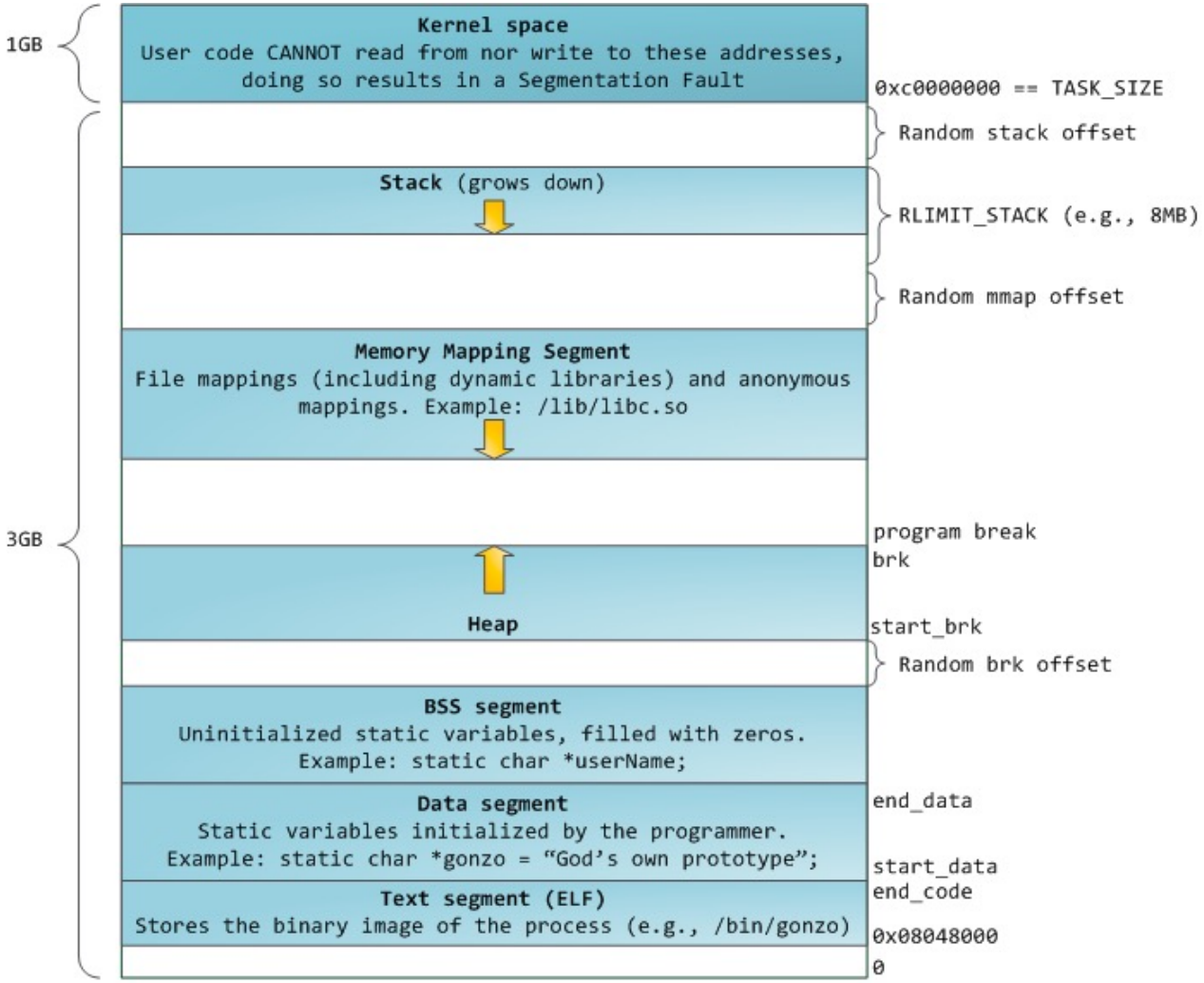
- Introduce TLB (Translation Lookaside buffer)
  - Caches VA-PA mapping
- Critical disadvantage of TLB?
  - Context switching
- TLB lockdown
  - For kernel region
- ASID
  - Tagged TLB

An example

- Let's assume that
- PMem[Addr/4] = Addr+0x1:0000
  - Mem is 4 bytes array
  - TTBR value is 0
  - What's the value of VA(0x1234:5678)?

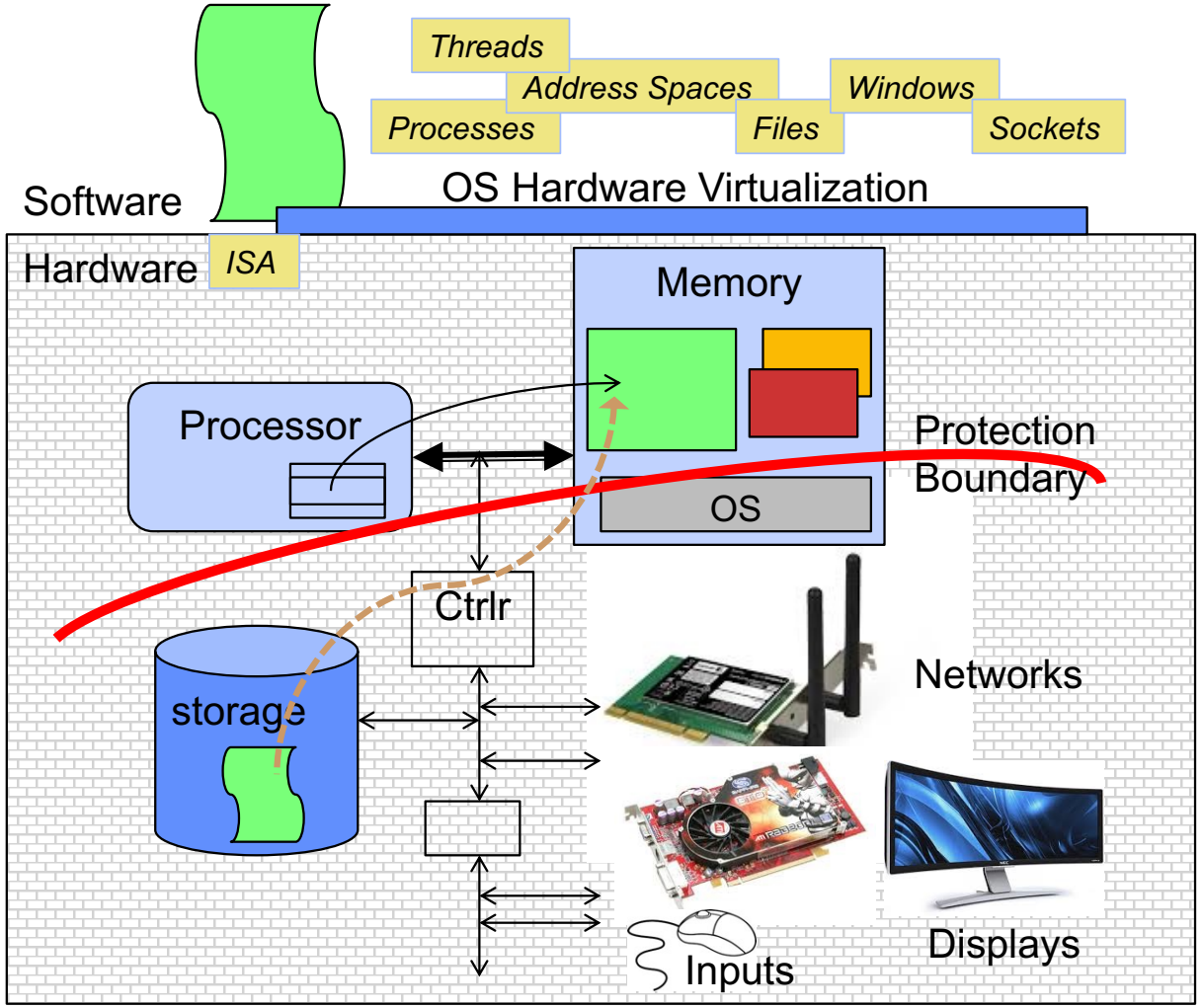
Shared mapping among processes

- Kernel region (code, data)
  - Address space is reserved!
  - All user processes share the region
- Shared library region (code, data)
  - Should be thread-safe
  - Can be re-entrant
- We can make a shared page table
  (shared 4MB mapping)



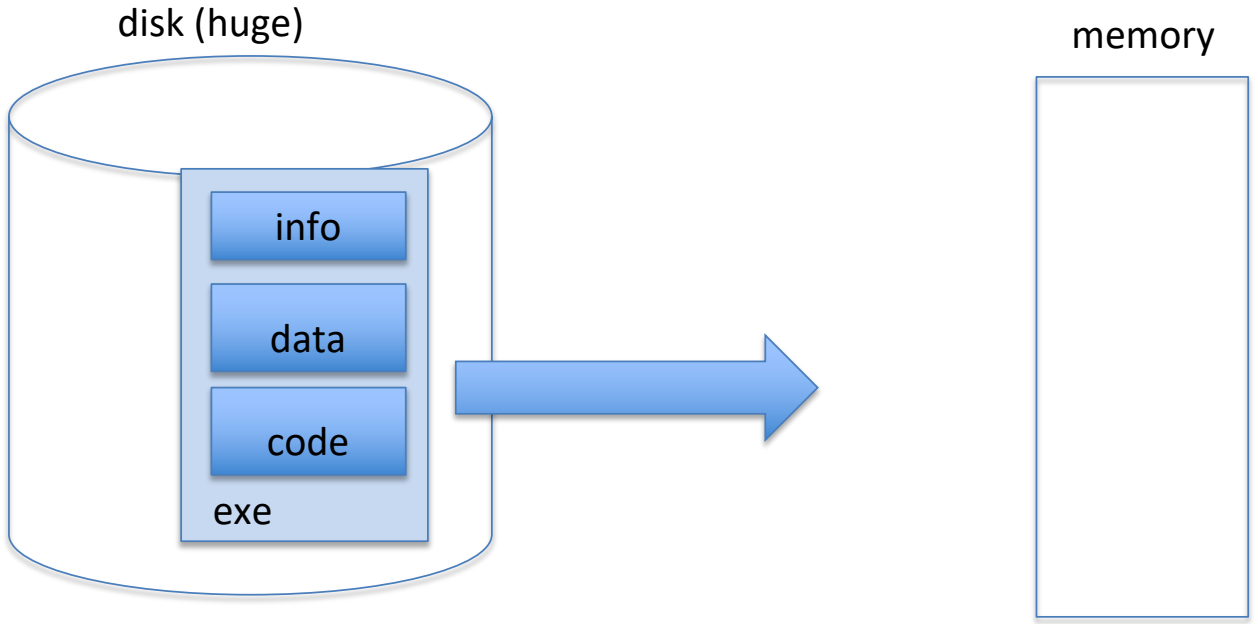
Disk Swapping

Recall: the most basic OS function



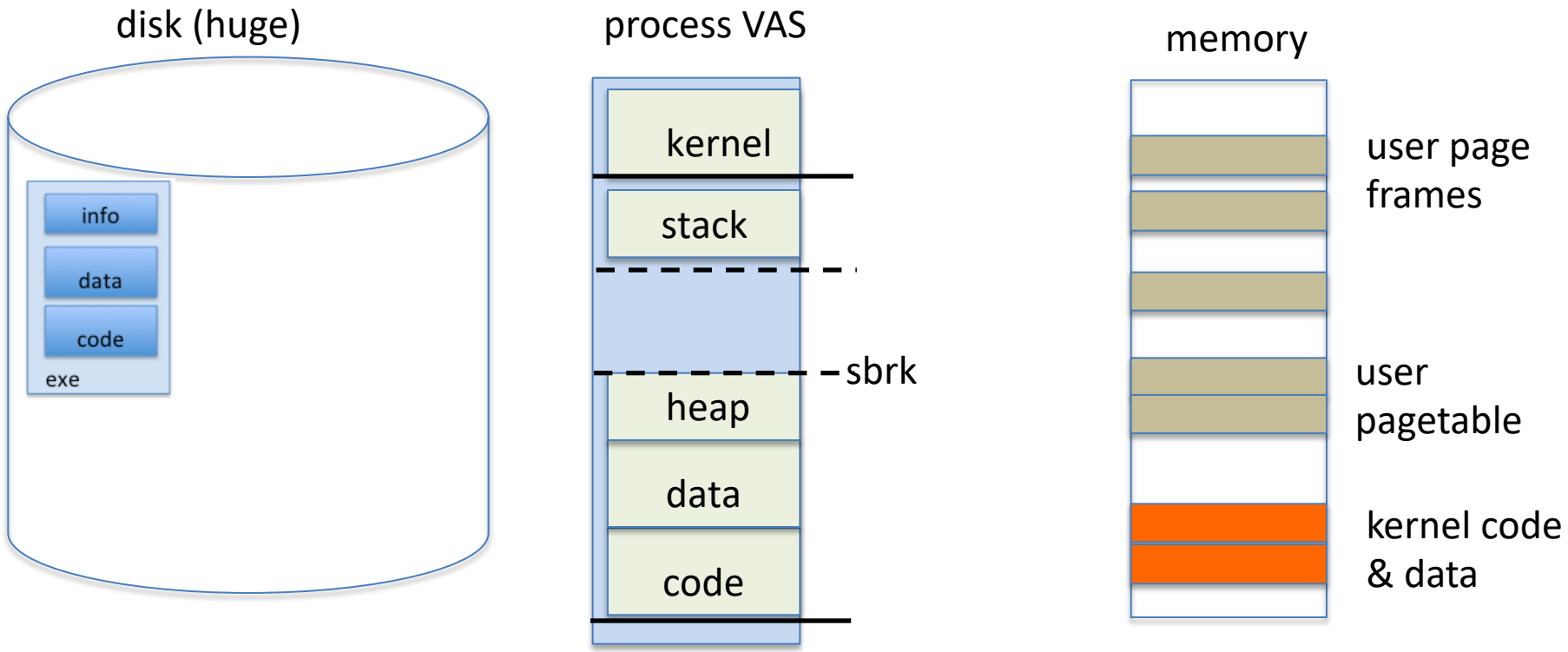
Loading an executable into memory

- .exe
  - lives on disk in the file system
  - contains contents of code & data segments, relocation entries and symbols
  - OS loads it into memory, initializes registers (and initial stack pointer)
  - program sets up stack and heap upon initialization: CRT0



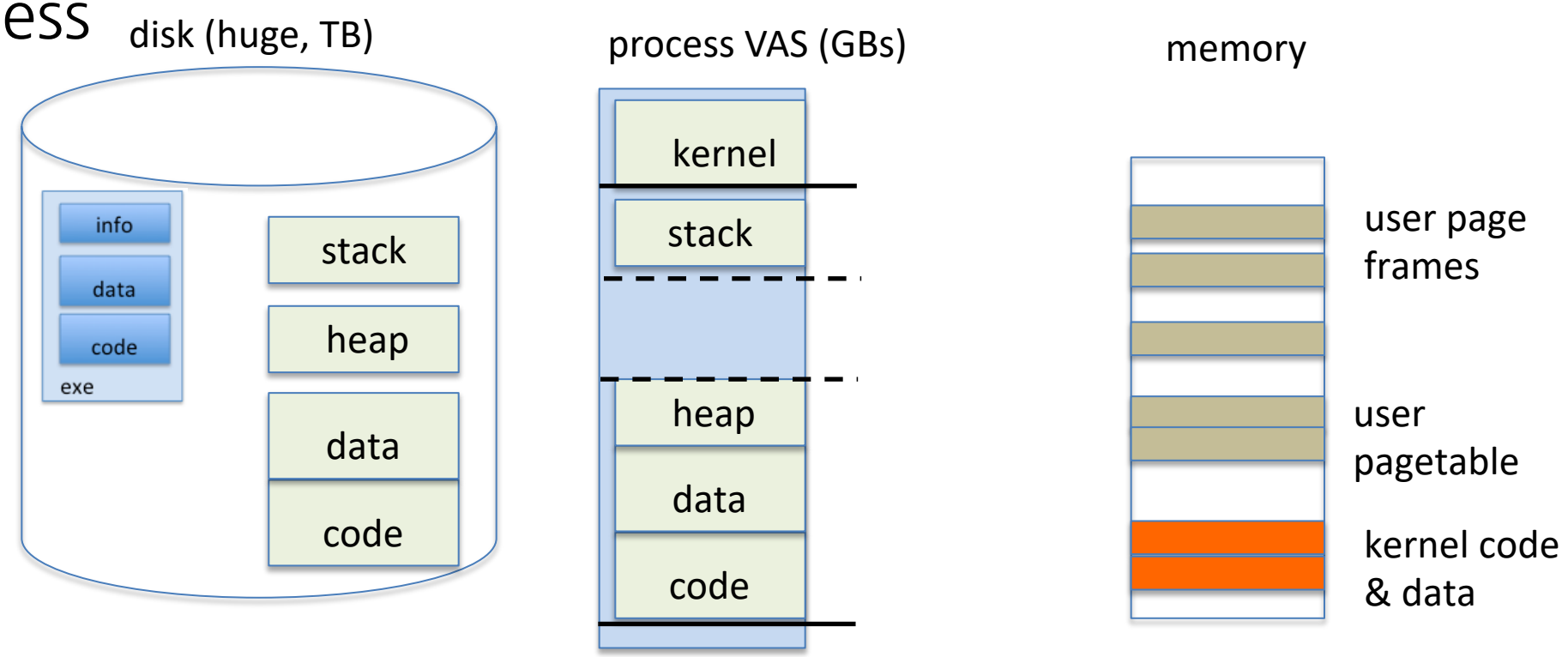
Create Virtual Address Space of the Process

- Utilized pages in the VAS are backed by a page block on disk
  - called the backing store
  - typically in an optimized block store, but can think of it like a file



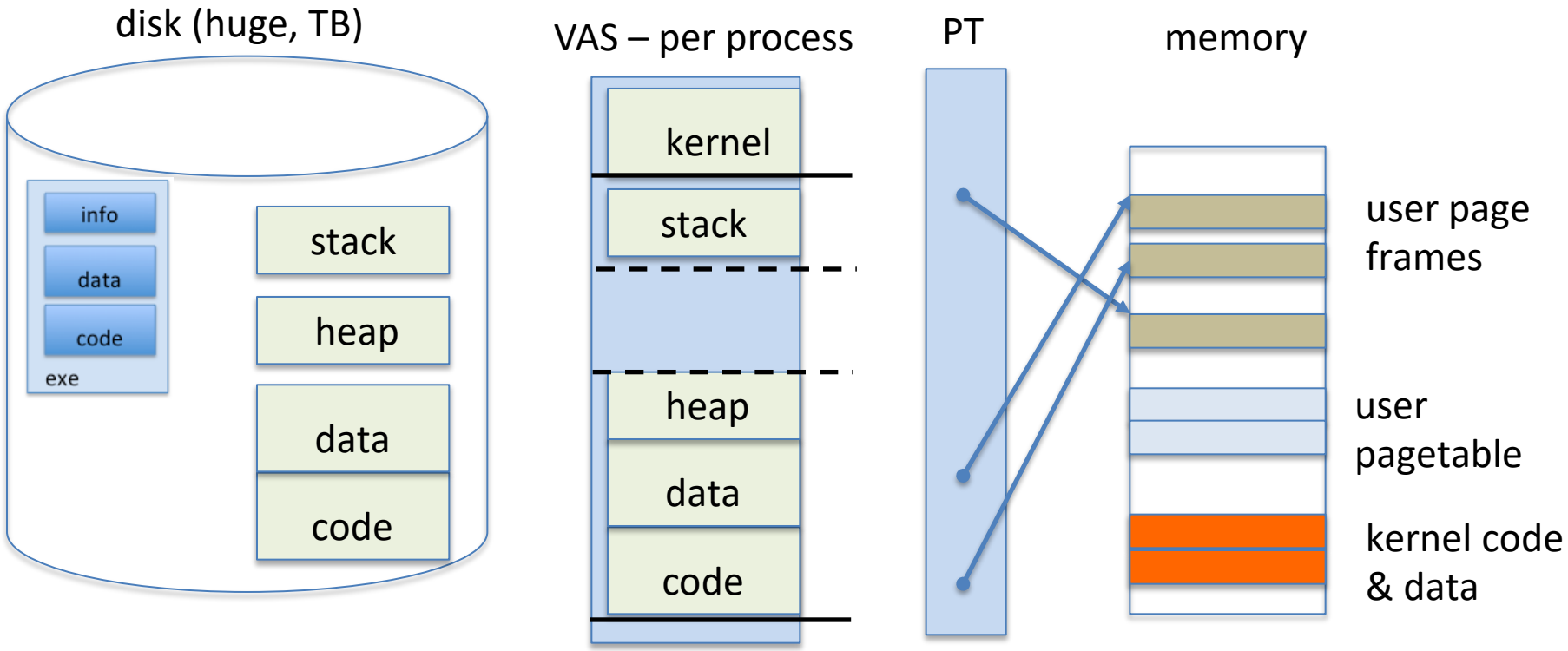
Create Virtual Address Space of the Process

- User Page table maps entire VAS
- All the utilized regions are backed on disk
  - swapped into and out of memory as needed
- For *every* process



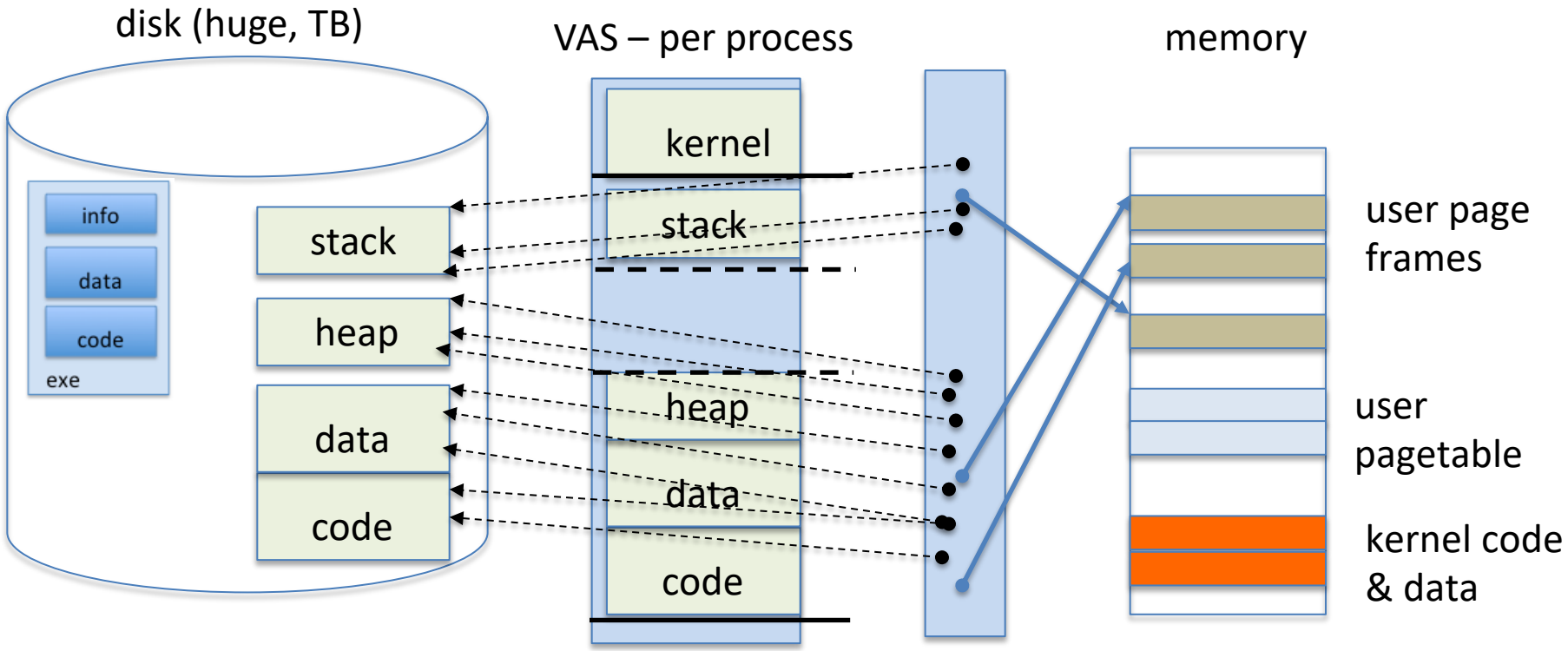
Create Virtual Address Space of the Process

- User Page table maps entire VAS
  - resident pages to the frame in memory they occupy
  - the portion of it that the HW needs to access must be resident in memory



Provide Backing Store for VAS

- User Page table maps entire VAS
- Resident pages mapped to memory frames
- For all other pages, OS must record where to find them on disk

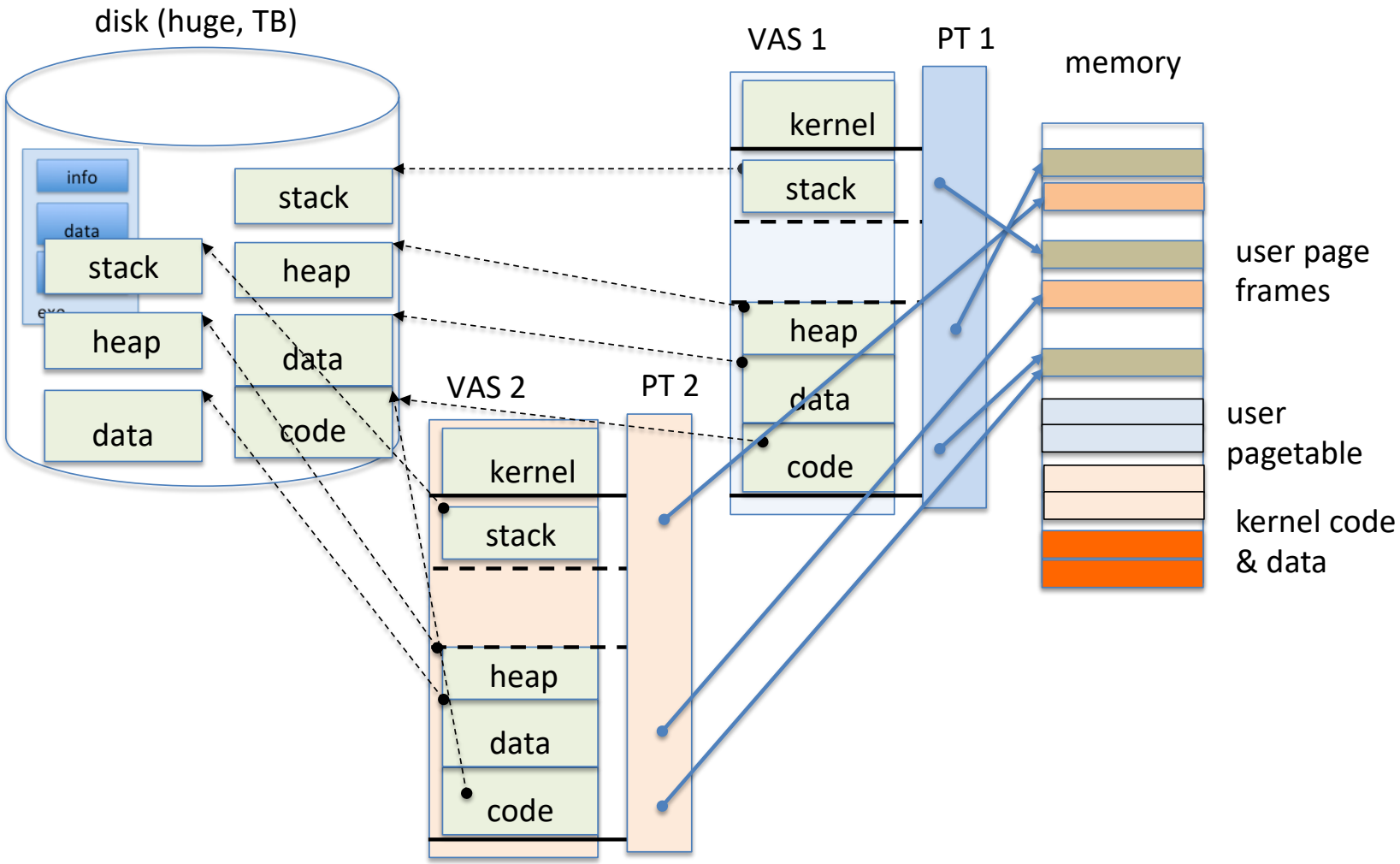


What data structure is required to map non-resident pages to disk?

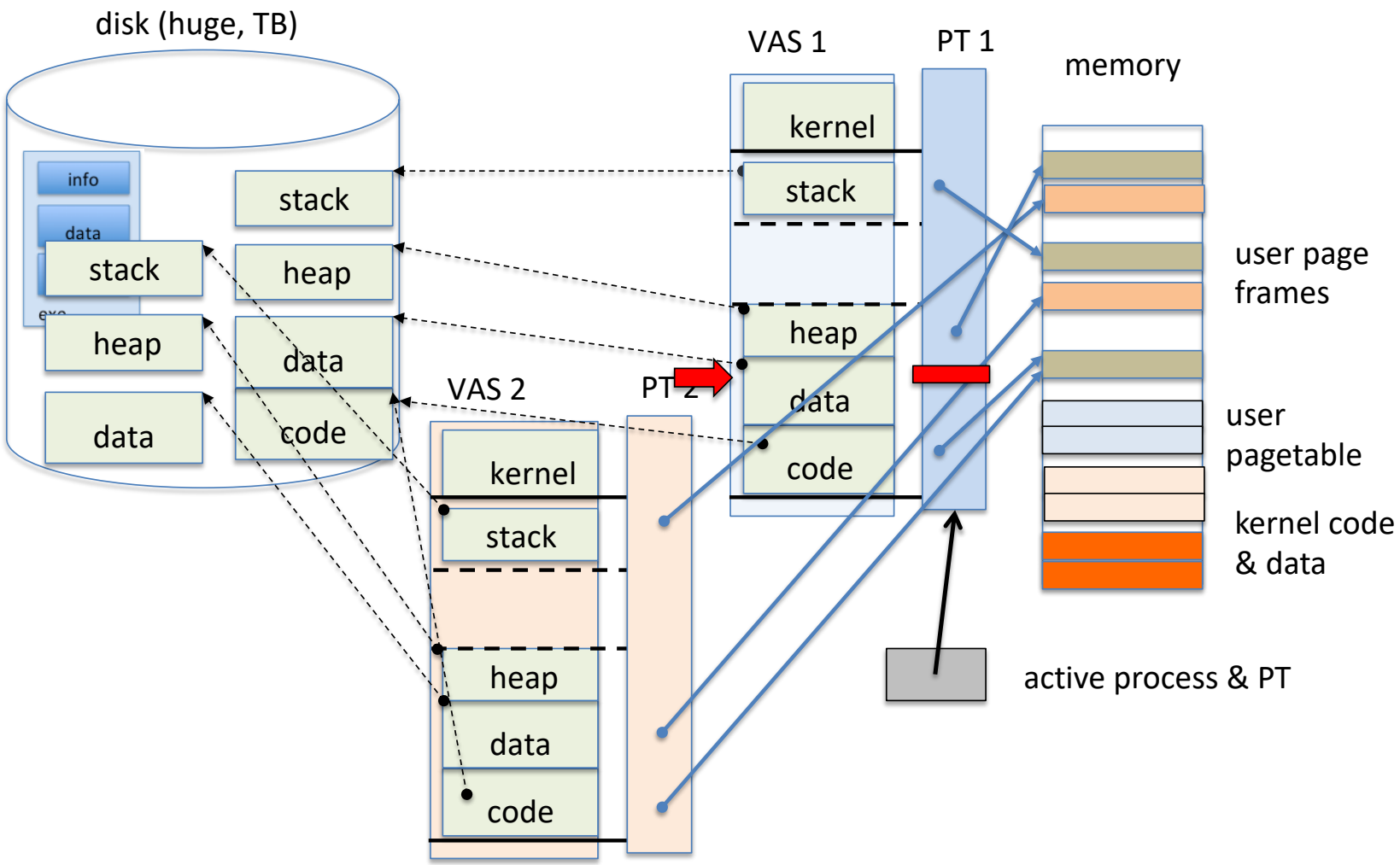
- FindBlock(PID, page#) => disk_block

- Like the PT, but purely software
- Where to store it?
- Usually want backing store for resident pages too.
- Could use hash table (like Inverted PT)

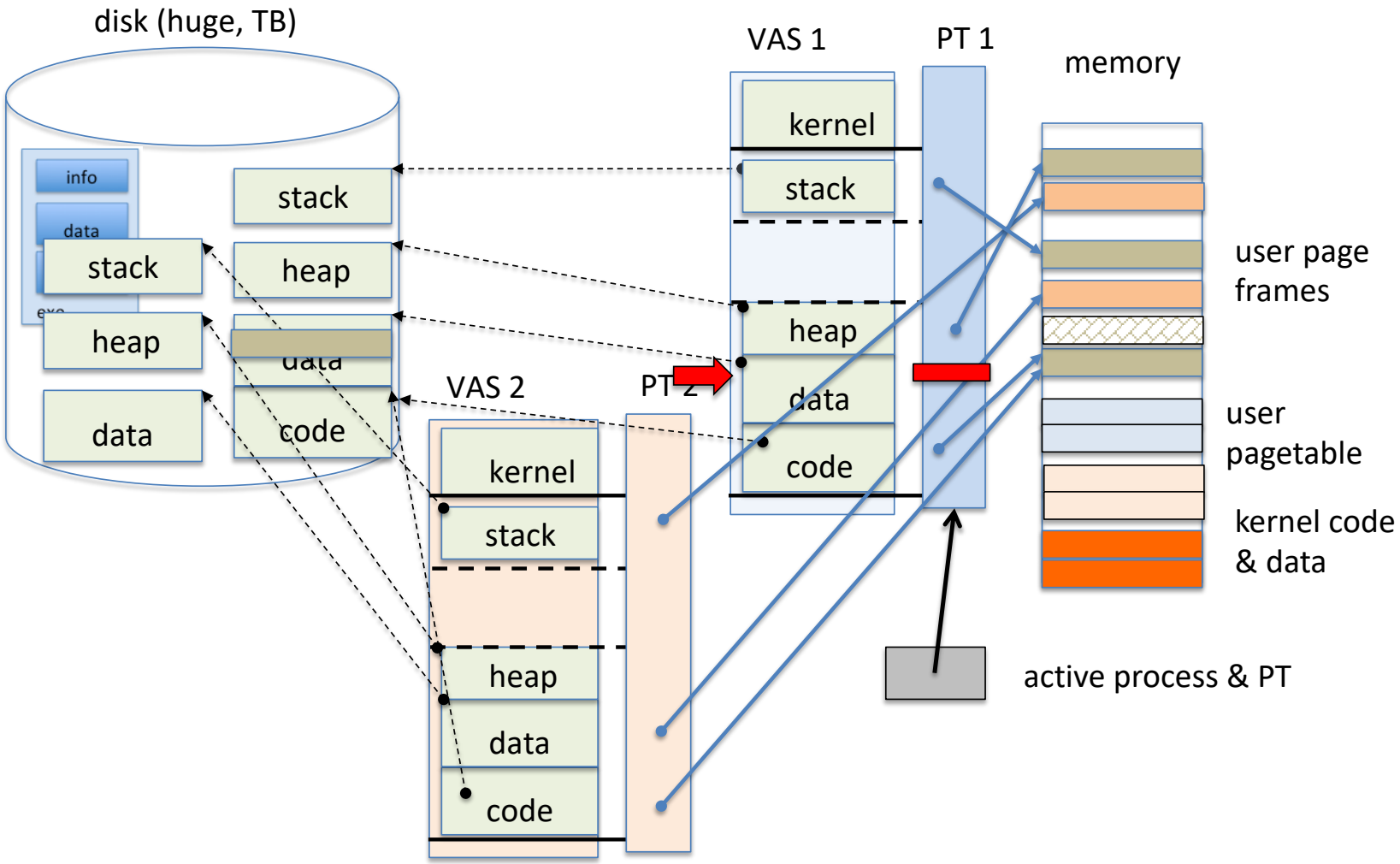
Provide Backing Store for VAS



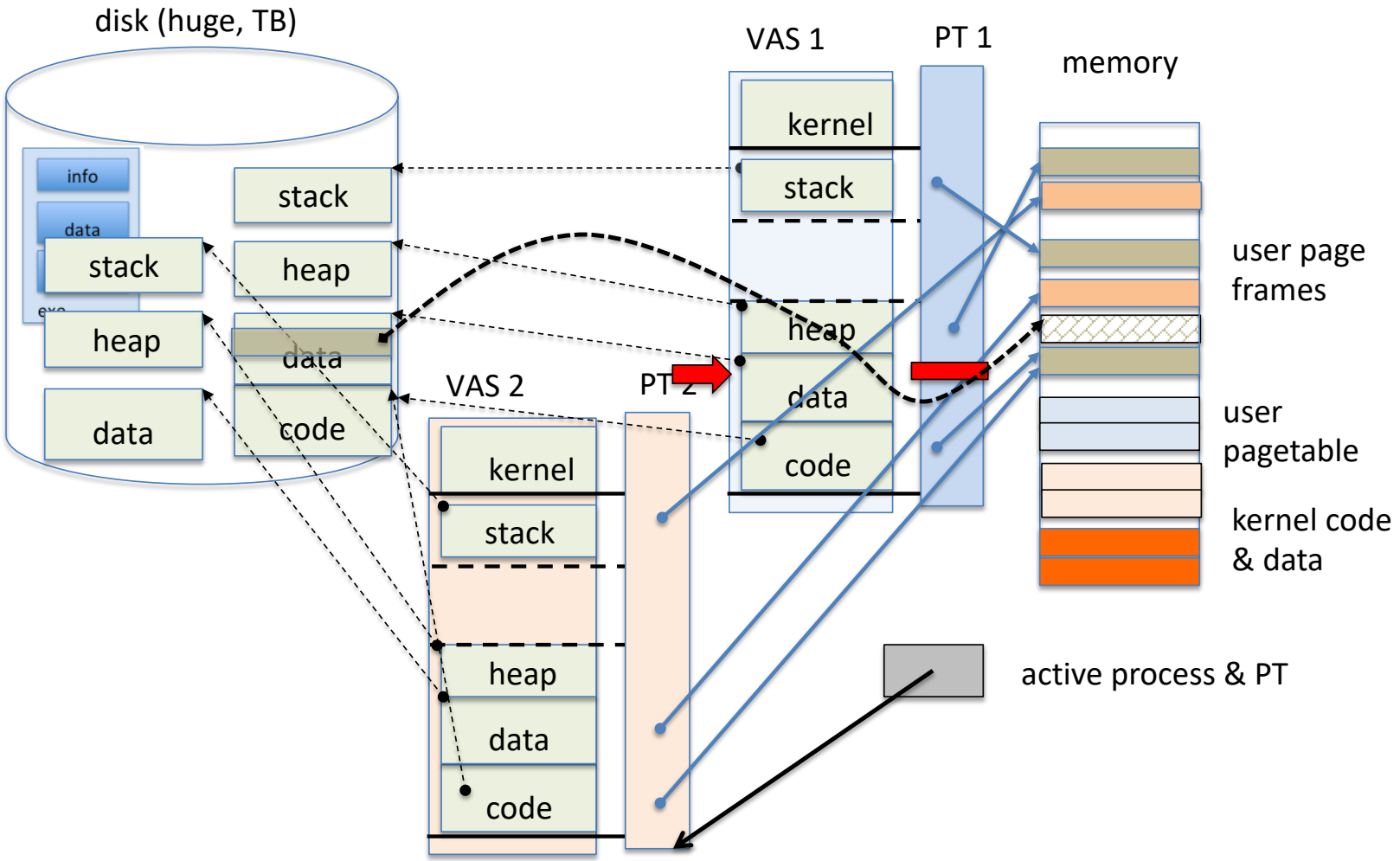
On page Fault …



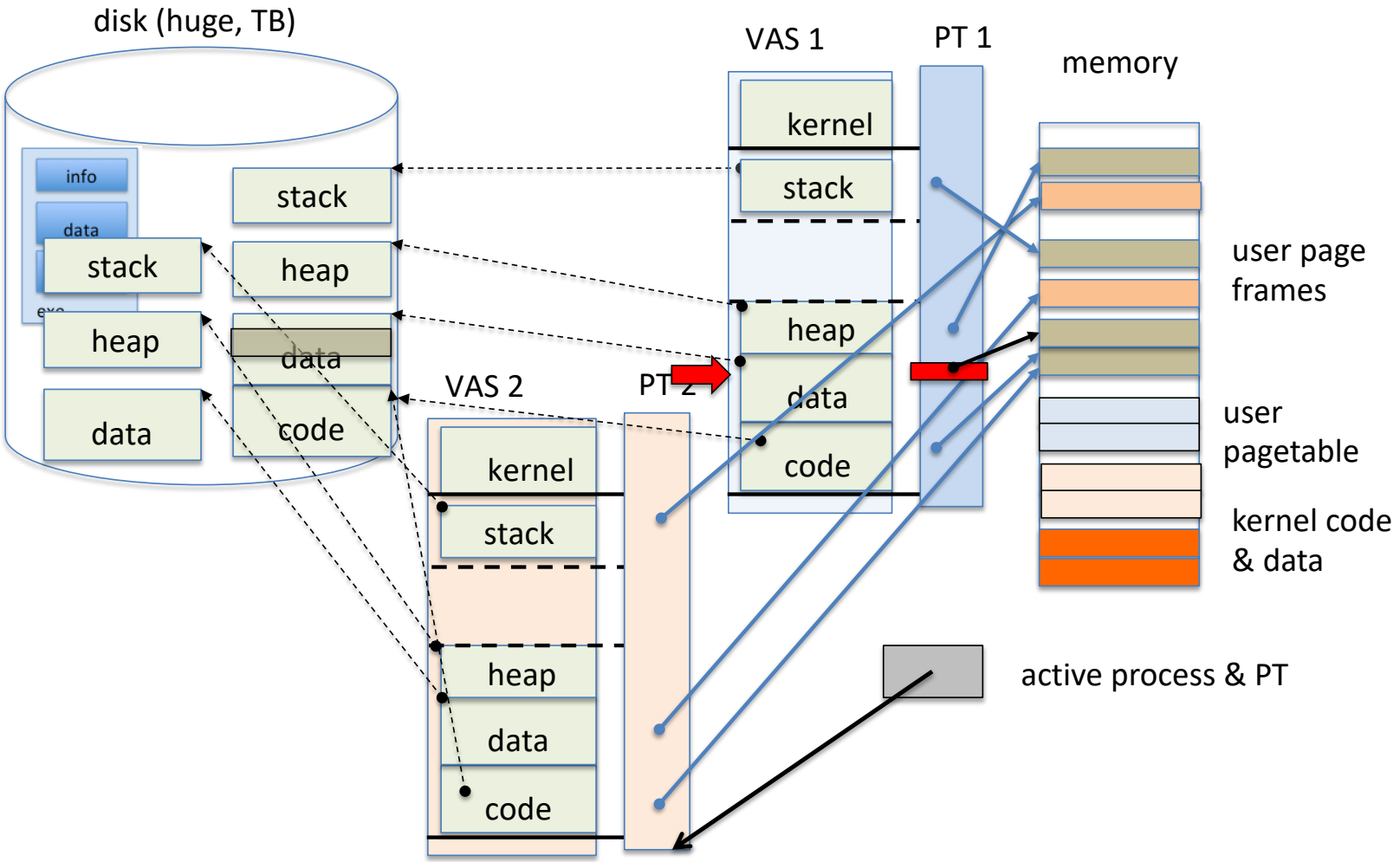
On page Fault … find & start load



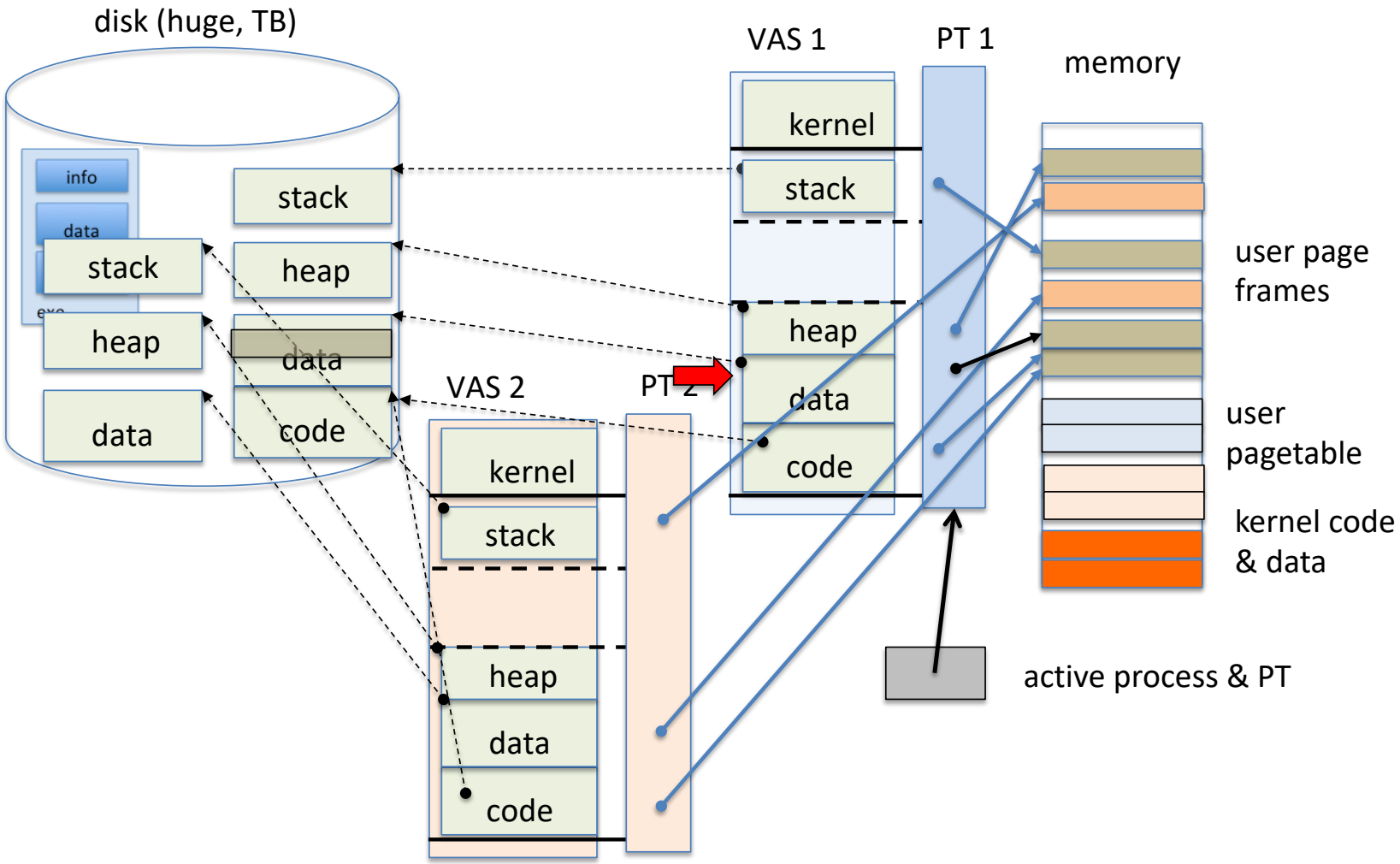
On page Fault … schedule other P or T



On page Fault … update PTE



Eventually reschedule faulting thread



Where does the OS get the frame?

- Keeps a free list
- Unix runs a “reaper” if memory gets too full
- As a last resort, evict a dirty page first

When OS begins,

- Initialize physical memory
  - Free-page frames list
- When a process begins, address space is given
  - Along with the page table
  - Initially empty mapping
- When a process accesses memory,
  - Page fault occurs when invalid mapping is accessed
  - OS makes new mapping
    - Finds a new page frame
    - Fills in the page table with the new page frame
  - Re-access the memory by re-executing the fault instruction

When we run out of physical memory,

- No more free memory list!
  - Yet, we need more free pages!
  - Swap a in-memory page with backing store's page
    - Q1) Whom to evict? - Similar policy with CPU cache
    - Q2) Where to store? - needs a mapping table

Whom to evict from memory to backing store?

- Least recent used one (LRU)
  - Complex algorithm
  - Approximation to this, second-chance algorithm
  - Fully implemented in software
    - Still, it is too complex
- Better idea - working set
  - Memory set that are used together
  - Evict all, swap-in all
  - How to know the working set?
- Dirty bit in page table
  - Clean pages do not need write-back (always can fetch from the file system)

Side note: Thrashing

- Consider a case that a process P1
  - request to memory M1 evicts M2 (was in physical memory)
  - M1 will come into memory (for execution)
  - Another (next) request to memory M2 evicts M1
  - Sometimes, it is too late!
    - P1 swap-out M1, M2, M1, M2, ….
      - Not much progress
      - Would affect the entire processes
- Thrashing
  - CPU utilization is low, consume most of the time for swapping
  - When free memory < working set
  - Before it is too late, swap early
    - Monitoring the memory usage, do swapping actively (swapper)
  - Working set can be swapped-out altogether

Page fault handling

- PID, page # → block mapping info is essential data for recovery
- To swap-in, you have to know the where you stored your data

- Page fault handling
  - Check pg#, identify entry in the page table
  - If invalid, identify data location in the backing store
  - Find a free page frame, copy data from backing store into new page frame
  - Maps the page into the address space by filling in page table
  - Restart instruction

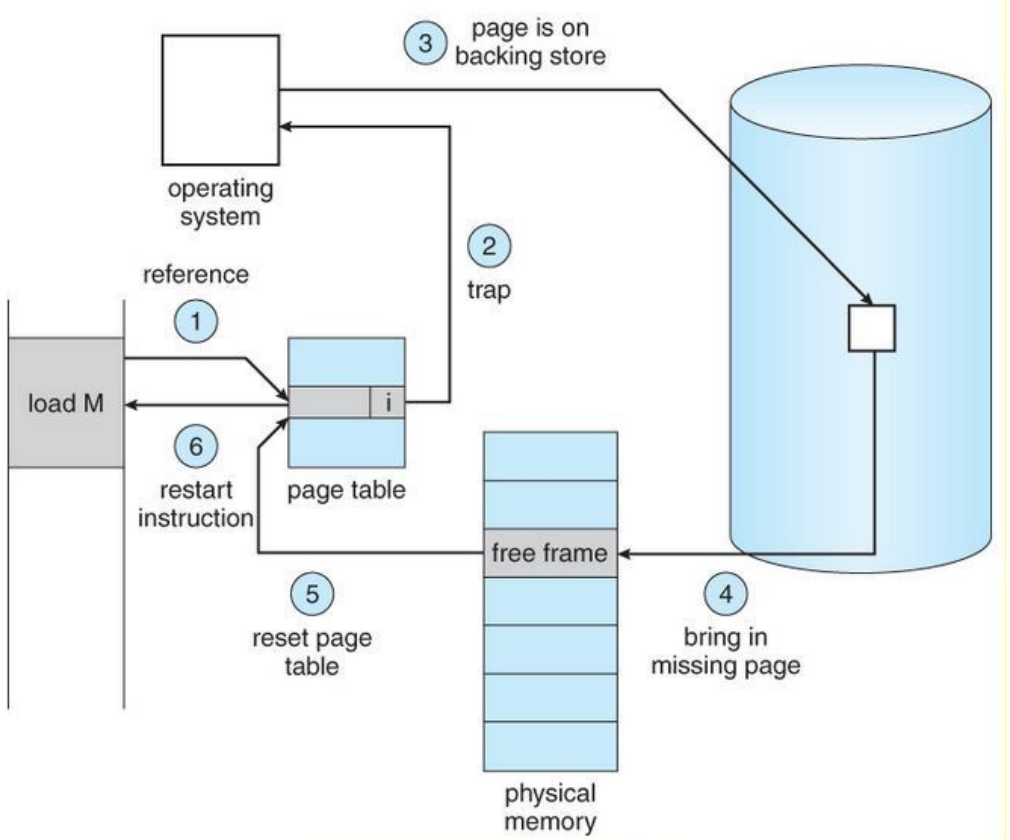


Figure 9.6 - Steps in handling a page fault

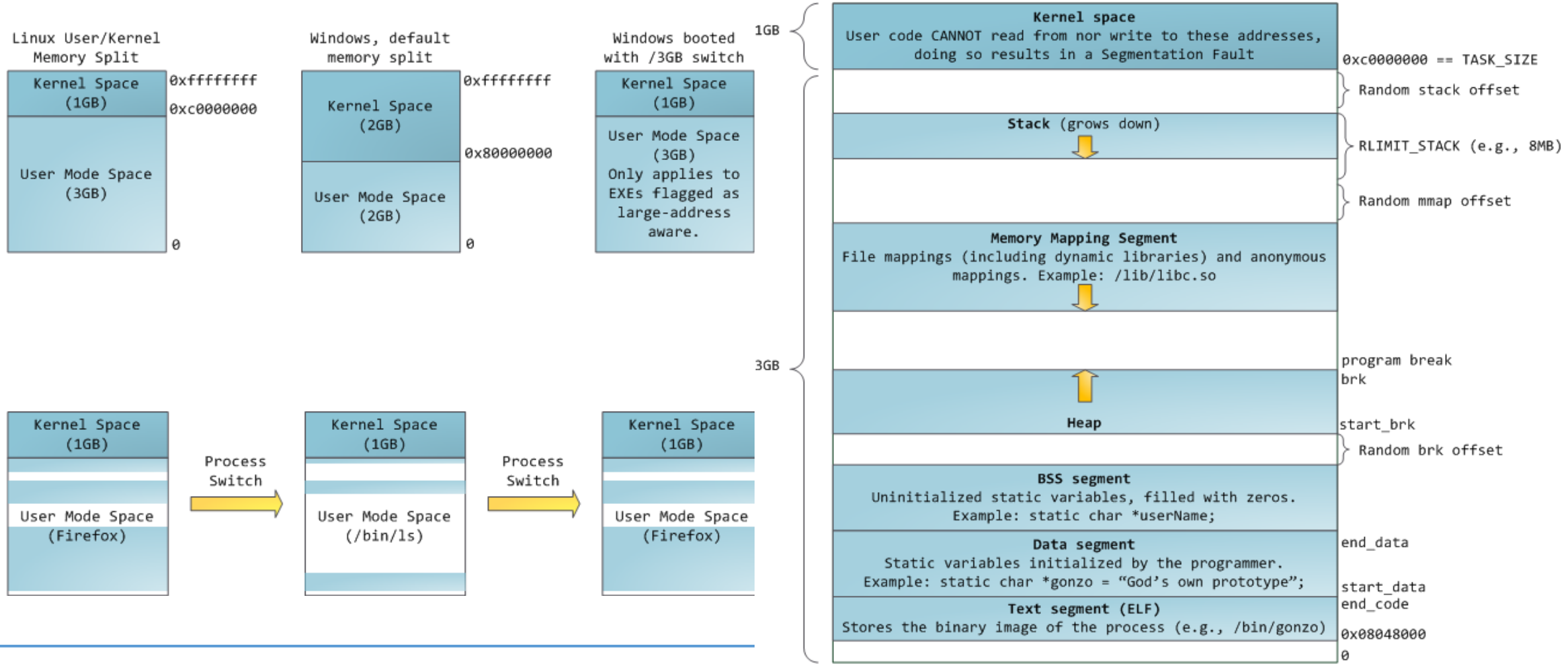
Pic. from OS Concepts

For the case of swap-out (replacement)

- Identify the page to swap out
- Pick a block in the backing store
  - You will have to remember it
- Copy data back to the backing store from the memory
- Keep the information
  - So that you can find in the case of future reference

Virtual memory address space layouts

- Internally,
  - Address space has pre-defined layout; cannot use 4G space for user only



Memory allocation at different levels

- User memory, put the flag (begin, end)
  - If you allocate more, end = end + size
  - malloc() simply returns the address that you can use
  - Kernel takes care of the physical memory, address space layout, page fault, etc.
- Kernel memory
  - Types of physical memory region
    - Kernel code, data, User code, data, rest of it
    - DMA / device region, etc. (buffer cache, page cache)
  - Logical separation of kernel virtual memory region (page allocation)
    - Kernel also have memory layout, heap, stack, etc.
    - Zone allocator
  - Page level allocation
    - Buddy algorithm
  - Fine-grained kernel memory allocation
    - Power-of-2 allocator (needs some data structure)

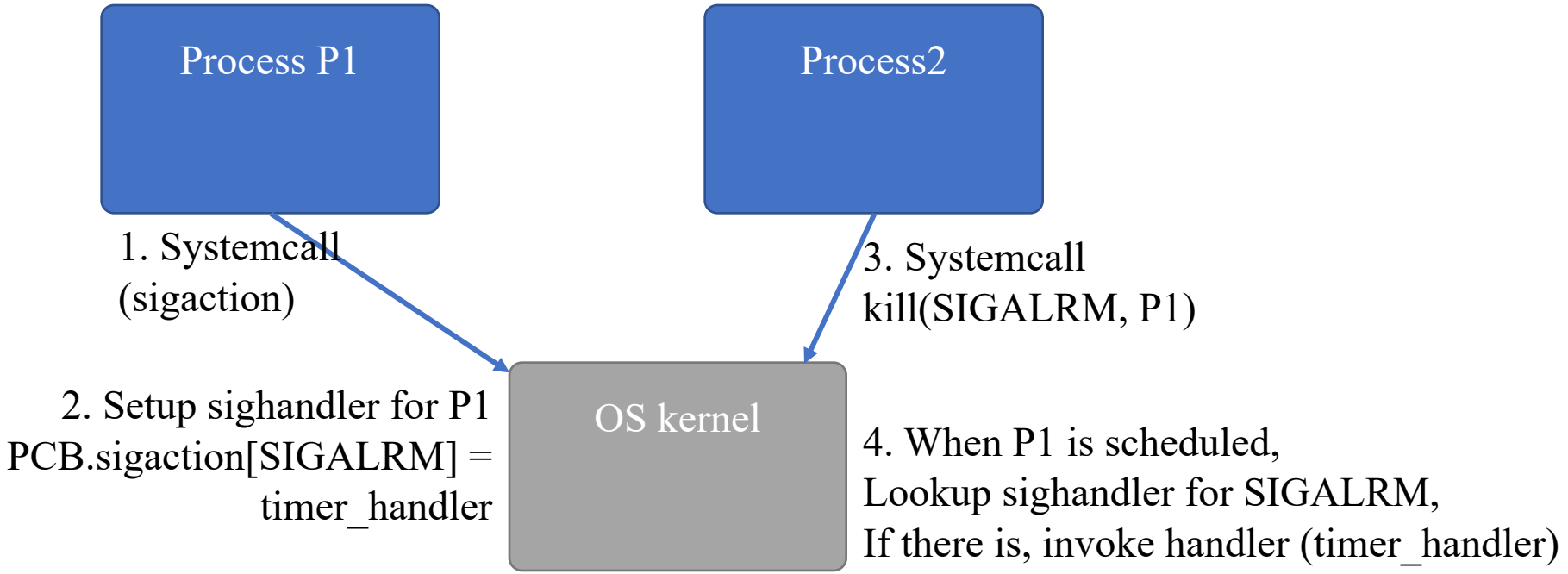
VM and its friends

VM with sharing

- Inter-process communication
- VM mapping between VA-PA
- What if VA1 → PA, and VA2→PA?
  - Different VA, points to the same PA
  - Same copy for different address
- Mapping across process boundary
  - What if P1.VA1 → PA, and P2.VA2→PA ?
  - Mapping across processes!
  - As a means of IPC
- Mapping for OS kernel memory
  - Shared among all the processes

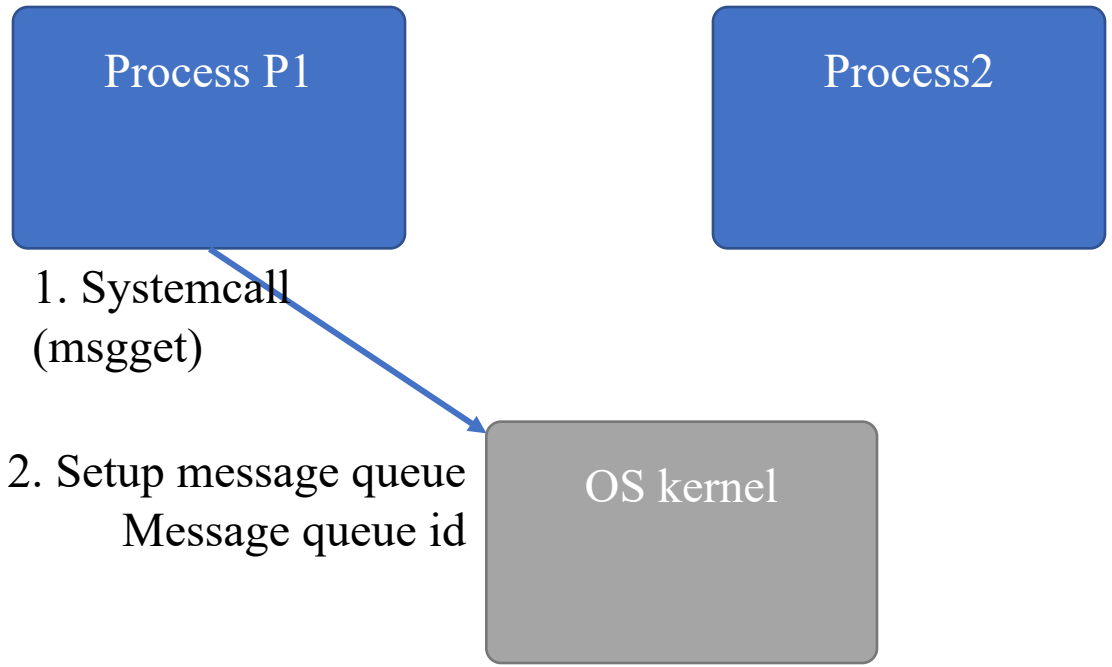
IPC facilities without shared memory

- Signal (signal setup/use)
- Message queue
- pipe



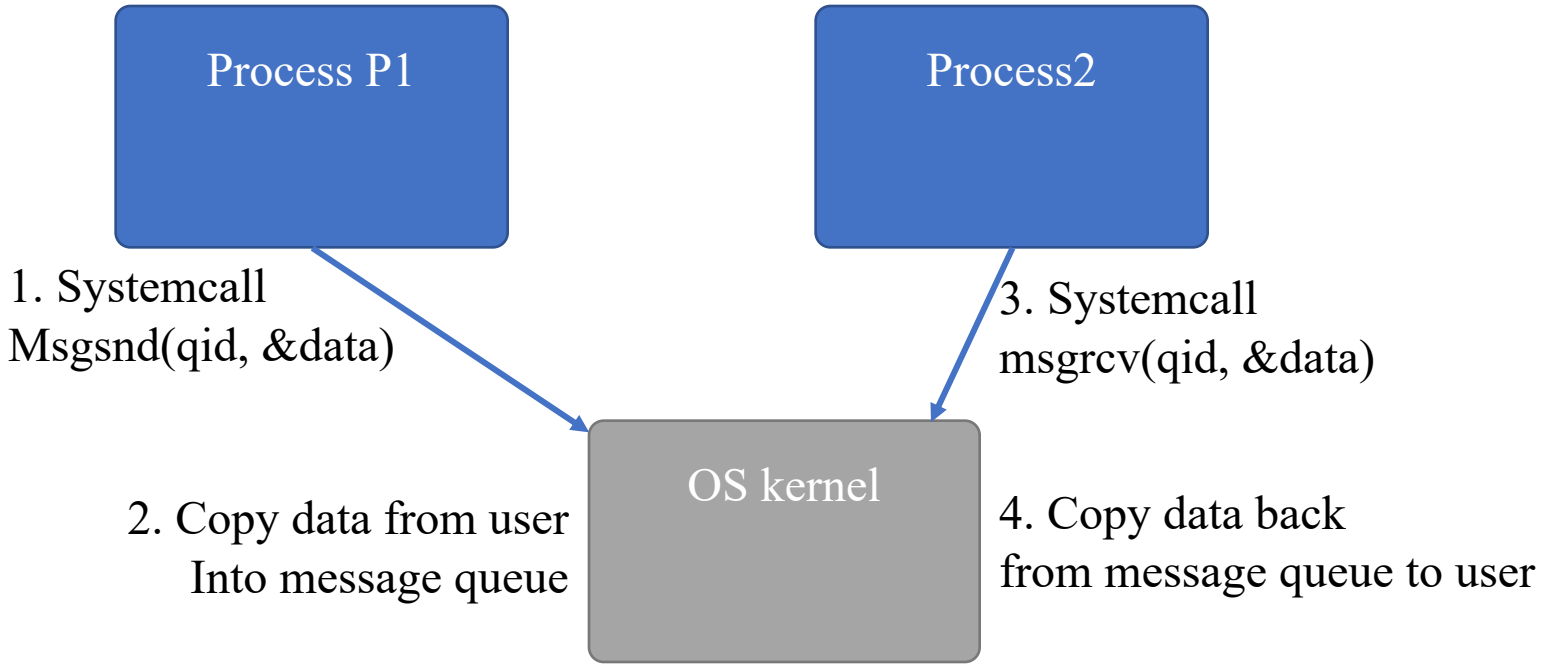
IPC facilities without shared memory

- Signal
- Message queue (initialization)
- pipe



IPC facilities without shared memory

- Signal
- Message queue (snd / rcv)
- pipe



Implication of data copy (esp. when data size is large)

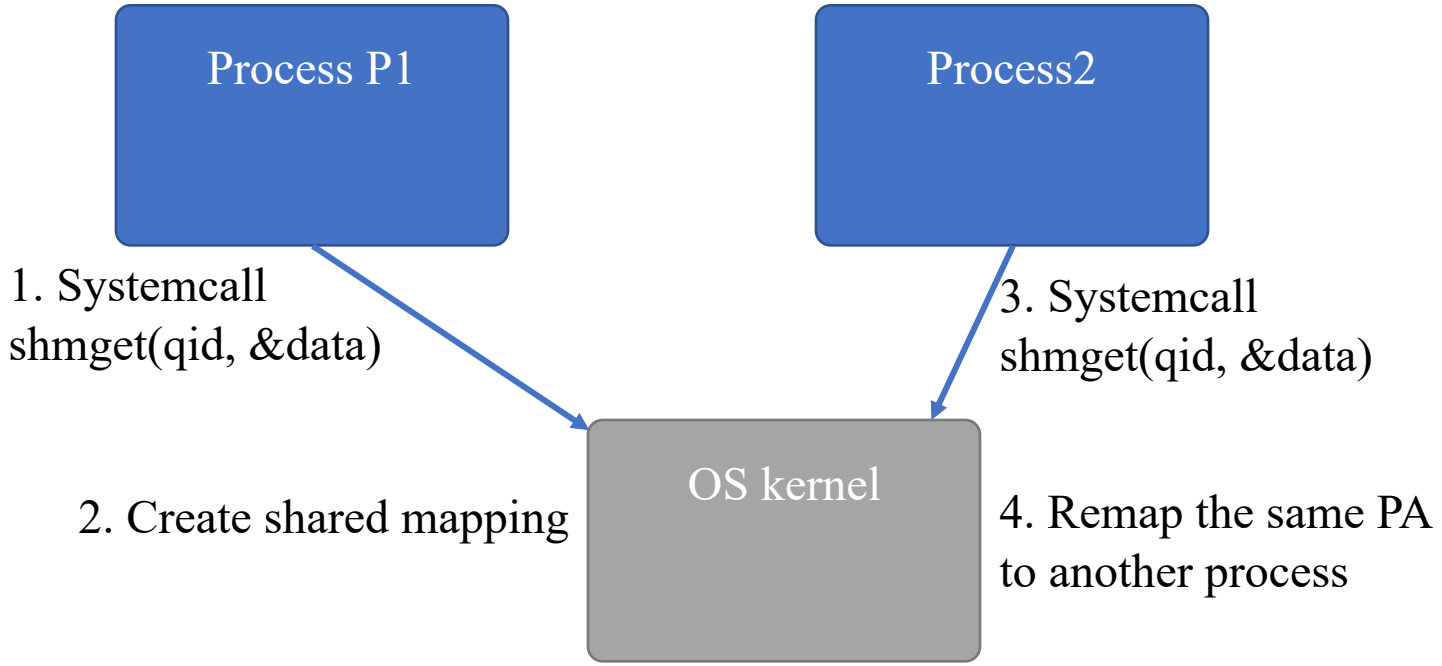
- Consume CPU cycles
  - Size could be large
  - Consume energy
- Byte-sized
  - Word size, Cacheline alignment mismatch
  - Per-byte size check
- Pollute cache
  - Useful data can be evicted from cache
  - Only efficient for a single (designated) process
- Large cache misses
  - Slow execution

Some mitigations from cache pollution, and misses

- Copy multiple bytes per operation
  - Word size consideration
  - Cacheline alignment
  - Avoid per-byte size check
    - Enhance copy algorithm
- Page table entry has Cache-able property
  - When cleared, the address regions are not cached
  - No cache pollution
  - OMG. This makes copy operation much slow!

A better way, re-map the page

- Shared memory
- No copy



A consideration

- Both processes can read/write data
  - Synchronization
  - That's your responsibility
  - Semaphore (semget, semop)
- Can be cached!
  - Cache with VA?
  - Cache with PA?
  - Virtual cache vs. physical cache?

Fork() and virtual memory

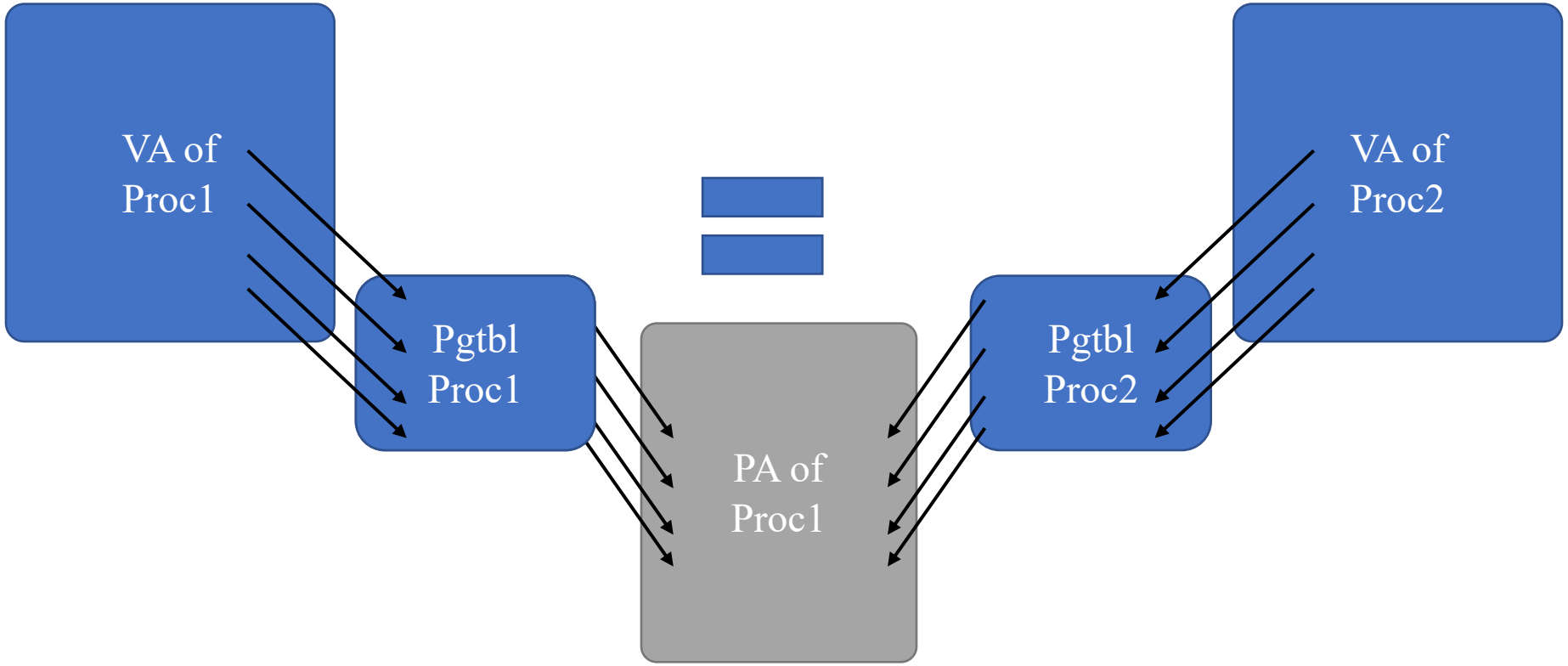
- Fork syscall creates a new process
  - Actually, copies from existing one (parent)
  - Threads of control
  - Address space (VA, same with the parent)
- Guess the page table at the time of fork?
  - Note that address space is given per-process
  - There are per-process page tables
  - One process has one page table

Fork() and virtual memory

- Fork syscall creates a new process
  - Actually, copies from existing one (parent)
  - Threads of control
  - Address space (VA, same with the parent)
- Guess the page table at the time of fork?
  - They (parent, child) are the same
  - Same mapping (PA-VA), regardless of the process boundary
    - Otherwise? To have the same data for the entire address space?
  - When they are detached?
    - When they are becoming different
    - When somebody write (sth) to memory

Copy-on-Write (CoW)

- Copy occurs when write is performed
  - Before? it maintains the same mapping

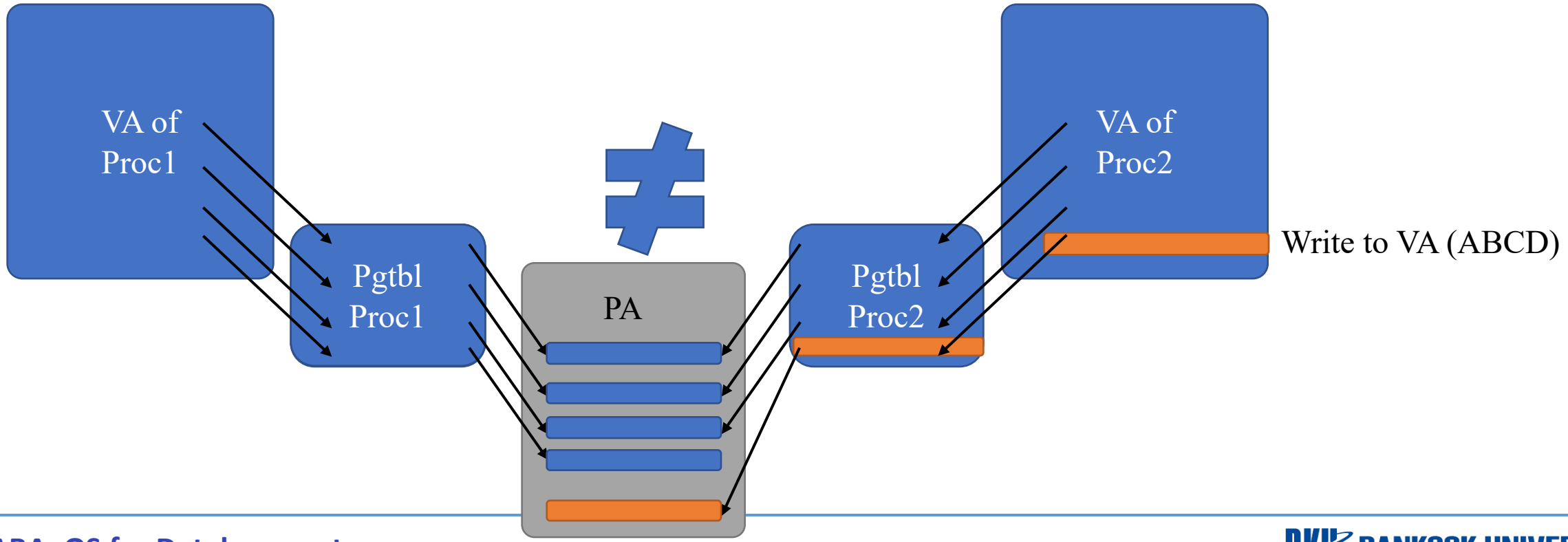


When they are detatched?

- Guess the page table at the time of fork?
  - They (parent, child) are the same
  - Same mapping (PA-VA), regardless of the process boundary
  - When they are detached?
    - When they are becoming different
    - When somebody write (sth) to memory
    - When somebody writes to memory → memory value becomes different
    - Two different processes should be detatched!
      → they cannot have the same mapping anymore!
    - They should have different mappings
    - The rest of the region (except for the modified bytes)?
      → should be copied
  - Copied when a process writes to memory!
    - Copy-on-write (CoW)

Detatched address space

- Mapping is 4K (small size mapping)
- Write 4 byte-sized integer value,
  then (4096-4) bytes have to be copied from original page frame



How can we know that a process writes to memory?

- Make the address space read-only
  - Setting up the page table
  - Normally, user data
- If a process begins to write, protection fault will occur
  - OS will catch the fault
  - Looks at the situation check whether the process in user mode, write data to read-only region (for available address)
  - Make a new mapping (find a page, fills in page table)
  - Copy the data from original page frame to new page frame
    - (further think about how)
  - Re-execute the instruction (write to new page frame)
  - Make the page read-write

Advantage of CoW

- Copy before? Copy on write?
  - Opportunistic approach
    - Use when it is actually needed
  - Human factor
    - Better to begin early
    - Give response to the user as early as possible
- vfork() syscall
  - Child borrows parent's resources before execve/exit is called
    - During the time, parent is suspended
  - Let child run as early as possible
    - Assuming that the child will call exec() or exit

MMIO vs. IO-mapped-IO

- Memory mapped I/O
  - I/O devices have memory map → I/O devices have internal VA-PA mapping
  - However, I/O devices registers are mapped to memory
- Do the I/O by accessing register → MM-I/O
  - Open/read/write/close
  - Read/write to memory → read/write to device register
- Device address space
  - Some address region (PA) is reserved for I/O devices
  - Virtual Memory allows OS to map I/O device region into process address space

MMap

- Memory mapped file
- File blocks can be mapped into memory
- Read from a file easily
- Write to the file easily

- Files can be device!
  - Read from device easily
  - Write to device easily
  - At the user side!
  - But, needs permission from OS

Shared library and DLLs

- Library shared across process boundary
- Use VM page sharing
- To use with multiple processes, some are statically allocated
  - Use the same address region for all processes
  - Where is printf ()? / homework: find out the location of printf() function
- Dynamic linking/loading library (DLL)
  - Load library dynamically
  - Changes the address of some symbols
  - Needs re-location
  - Why?

Your code is based upon static memory layout

- Assume a.out came from file_a.o, file_b.o, file_c.o
  - File_a has main(), calls foo()
  - File_b has foo(), calls bar()
  - File_c has bar()
- Location of foo()?
- Location of bar()?
- You can make your own library lib_a.o that has foo() and bar()
  - At link time, the location is fixed for foo() and bar()
  - lib_a.o can work with file_a.o
  - lib_a.o can work with another program file_z.o, where file_z.o has another main(), calls foo()

Position independent code (PIC)

- How about shared library?
  - Easy if it is static
  - All address is fixed
- Making a library position independent
  - Include symbol table for the library
  - So that it can be re-located at different address (memory location)
- DLL (dynamic linking library)
  - Library, that changes location at runtime
  - Different location for different process
  - ASLR (Address space layout randomization) effectively blocks attacks from some malicious shared library, that has pre-known vulnerable memory location

Mini-lab.