Operating Systems Practice

Project #2 – (Large) Inverted Page Table

Seonggyun Oh

(sungkyun123@dgist.ac.kr)

Project #2

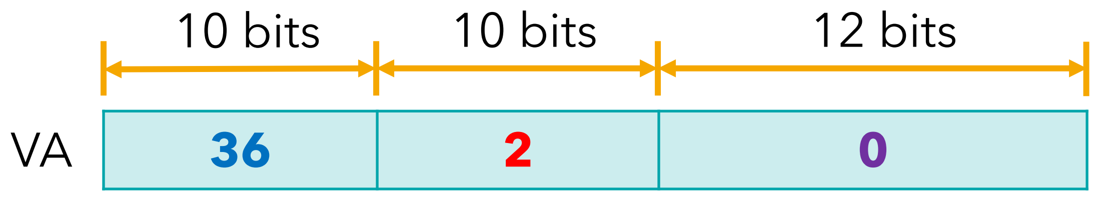
- Implement a large inverted page table in the xv6
  - Modify the default paging scheme (multi-level paging) to large inverted page table

- Objectives of this project
  - Understand How multi-level paging are performed in xv6 code
  - Implement a inverted page table

- Where to look and write code:
  - kalloc.c, vm.c (+etc)

Paging Structure in xv6: Multi-Level Paging

- 2-level paging
  - 1 page directory and 1 page table
  - Example: Write page to VA 0x9002000 (0b0000100100000000001000000000000)



Page directory address

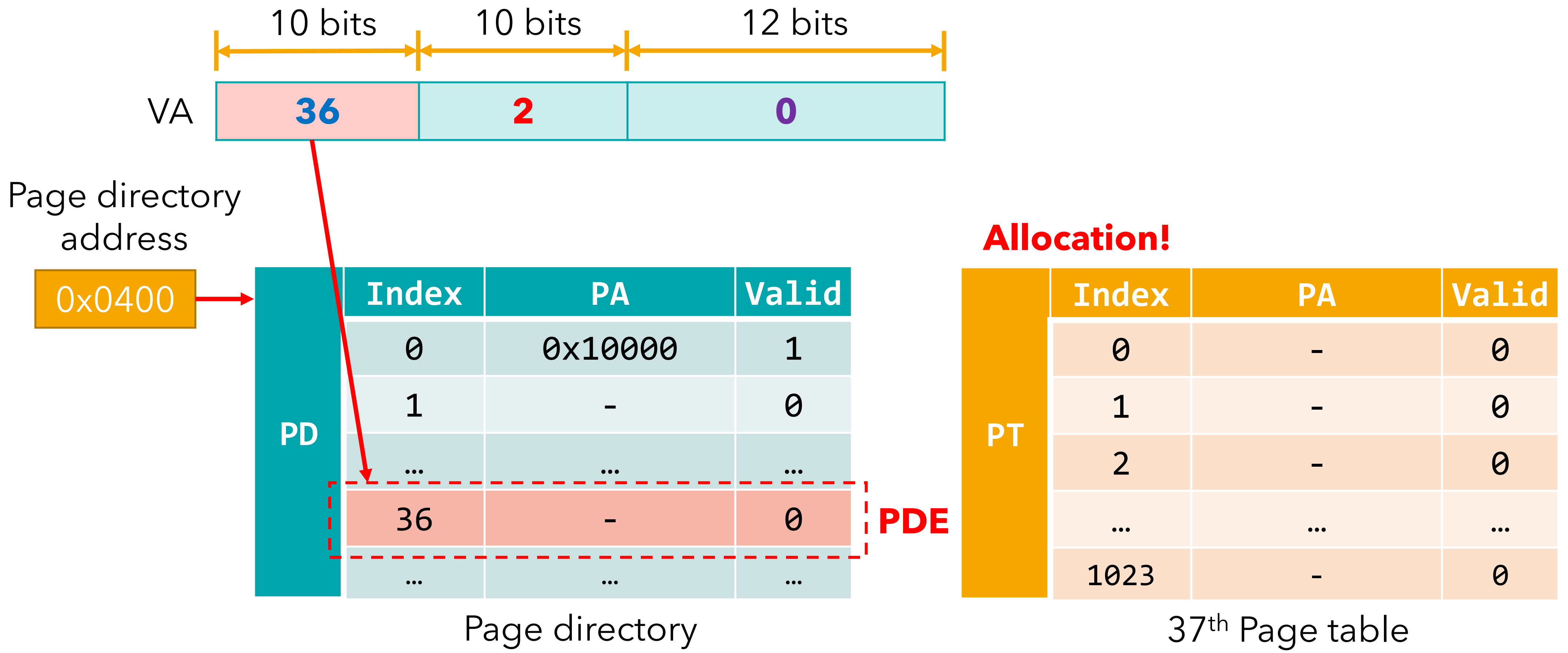
0x0400 → PD

PD	Index	PA	Valid
	0	0x10000	1
	1	-	0
	…	…	…
	36	-	0
	…	…	…

Page directory

Paging Structure in xv6: Multi-Level Paging

- 2-level paging
  - 1 page directory and 1 page table
  - Example: Write page to VA 0x9002000 (0b00001001000000000010000000000000)



Index	PA	Valid
0	0x10000	1
1	-	0
…	…	…
36	-	0
…	…	…

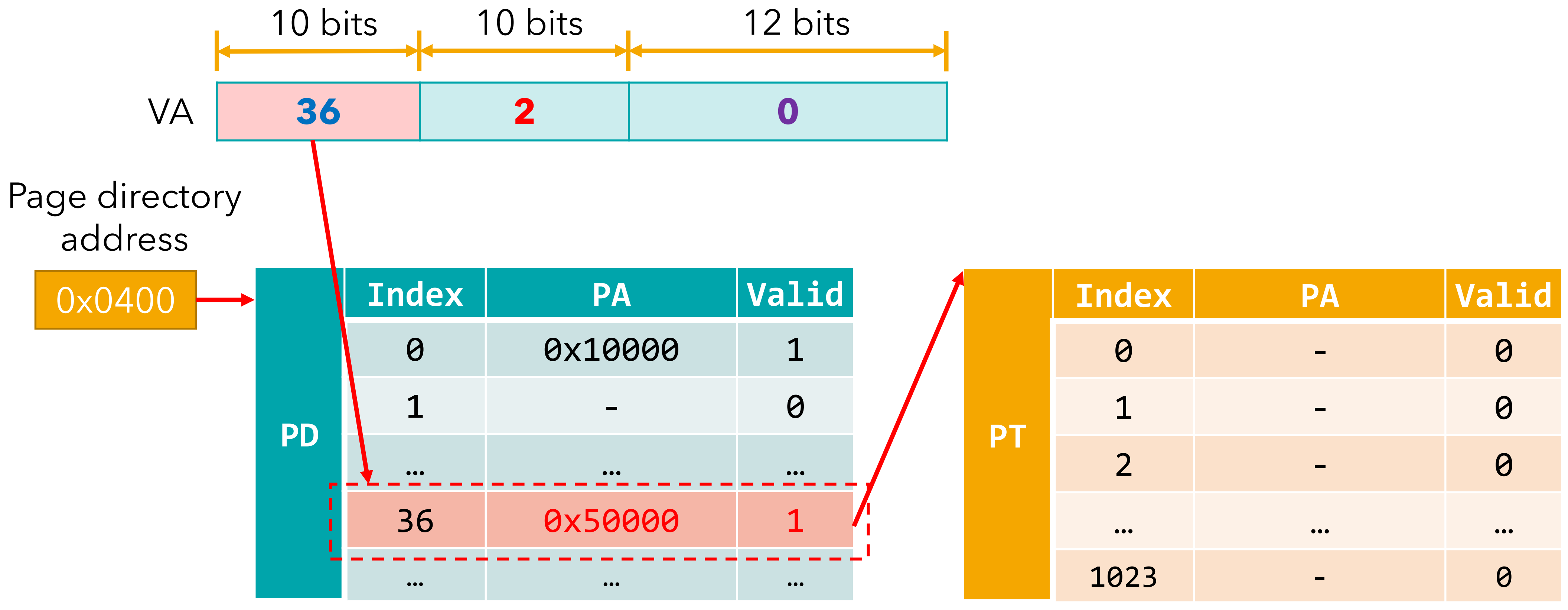
Page directory

Index	PA	Valid
0	-	0
1	-	0
2	-	0
…	…	…
1023	-	0

37th Page table

Paging Structure in xv6: Multi-Level Paging

- 2-level paging
  - 1 page directory and 1 page table
  - Example: Write page to VA 0x9002000 (0b00001001000000000010000000000000)



Index	PA	Valid
0	0x10000	1
1	-	0
…	…	…
36	0x50000	1
…	…	…

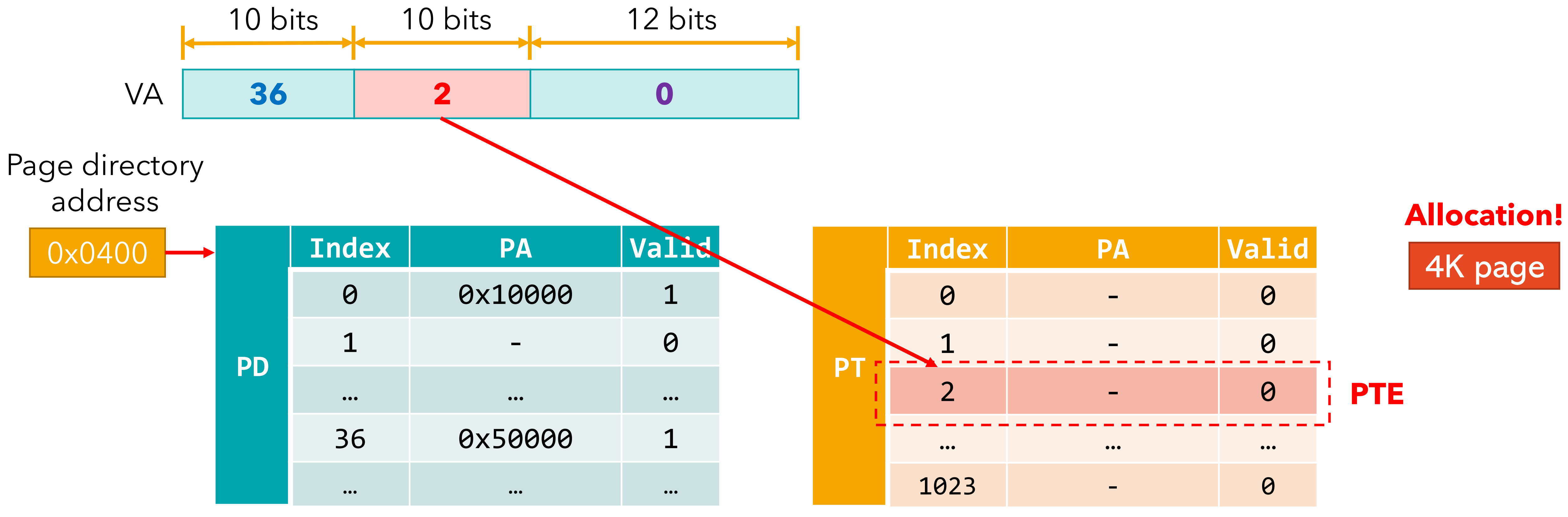
Page directory

Index	PA	Valid
0	-	0
1	-	0
2	-	0
…	…	…
1023	-	0

37th Page table

Paging Structure in xv6: Multi-Level Paging

- 2-level paging
  - 1 page directory and 1 page table
  - Example: Write page to VA 0x9002000 (0b00001001000000000010000000000000)



Index	PA	Valid
0	0x10000	1
1	-	0
…	…	…
36	0x50000	1
…	…	…

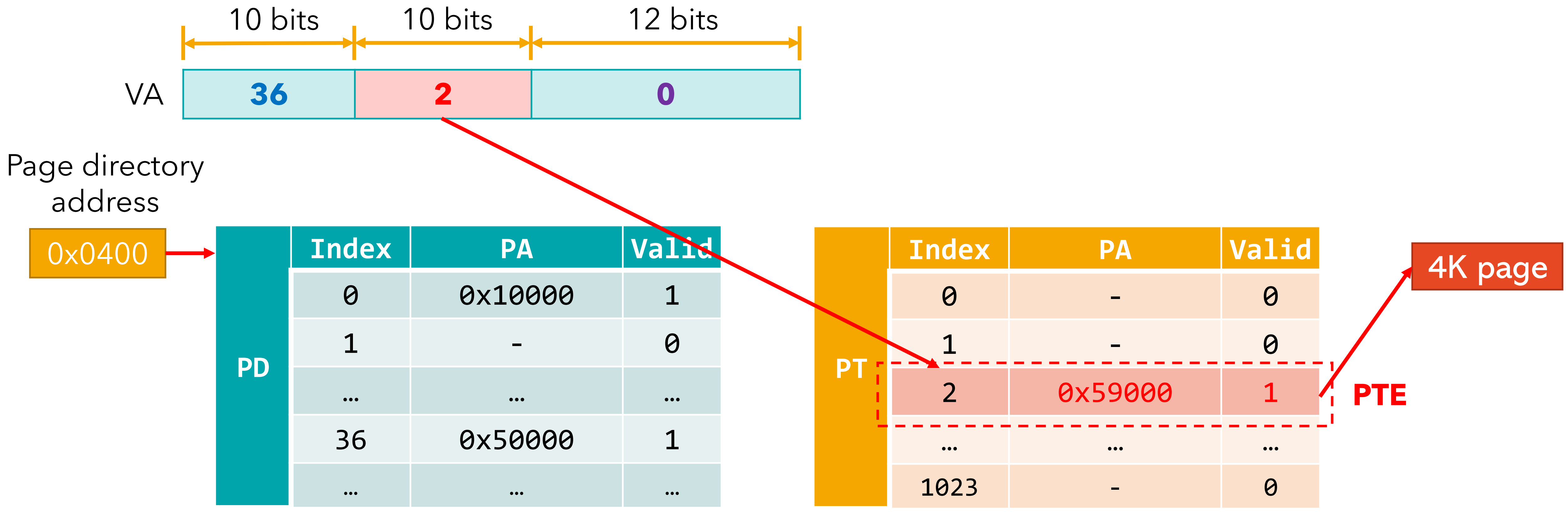
Page directory

Index	PA	Valid
0	-	0
1	-	0
2	-	0
…	…	…
1023	-	0

37th Page table

Paging Structure in xv6: Multi-Level Paging

- 2-level paging
  - 1 page directory and 1 page table
  - Example: Write page to VA 0x9002000 (0b00001001000000000010000000000000)



Index	PA	Valid
0	0x10000	1
1	-	0
…	…	…
36	0x50000	1
…	…	…

Page directory

Index	PA	Valid
0	-	0
1	-	0
2	0x59000	1
…	…	…
1023	-	0

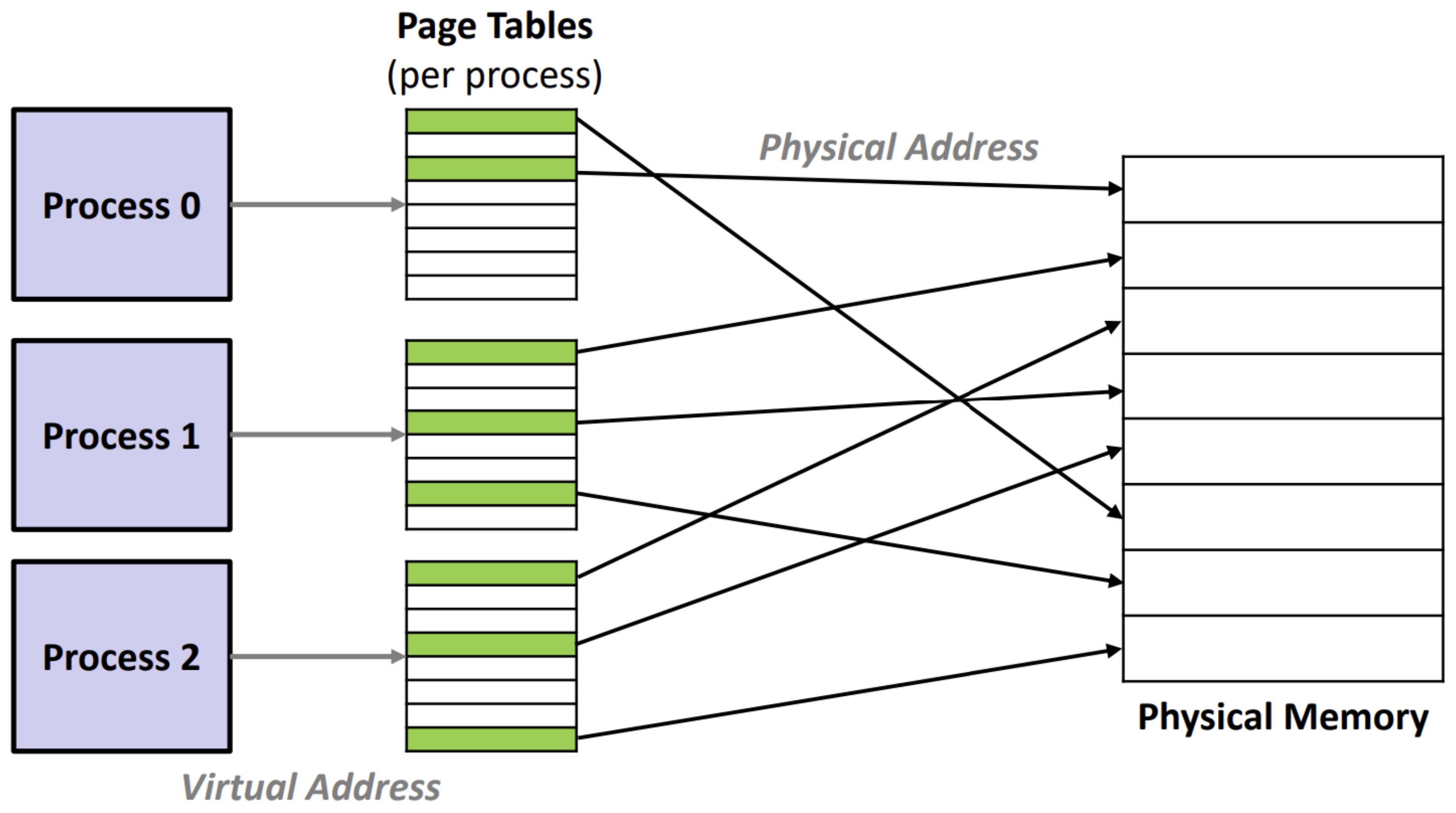
37th Page table

Pros and Cons of Multi-Level Paging

- Pros
  - Memory saving compared to linear page table
    - Only allocates page-table space in proportion to the amount of address space the process uses
- Cons
  - Time-space trade-off
    - On a TLB miss, it requires multiple memory references
      - it gets worse as the number of levels increases
  - Complexity
    - It should manage more than one table
  - Memory requirement
    - As the number of processes increases, the memory requirement for paging is non-negligible

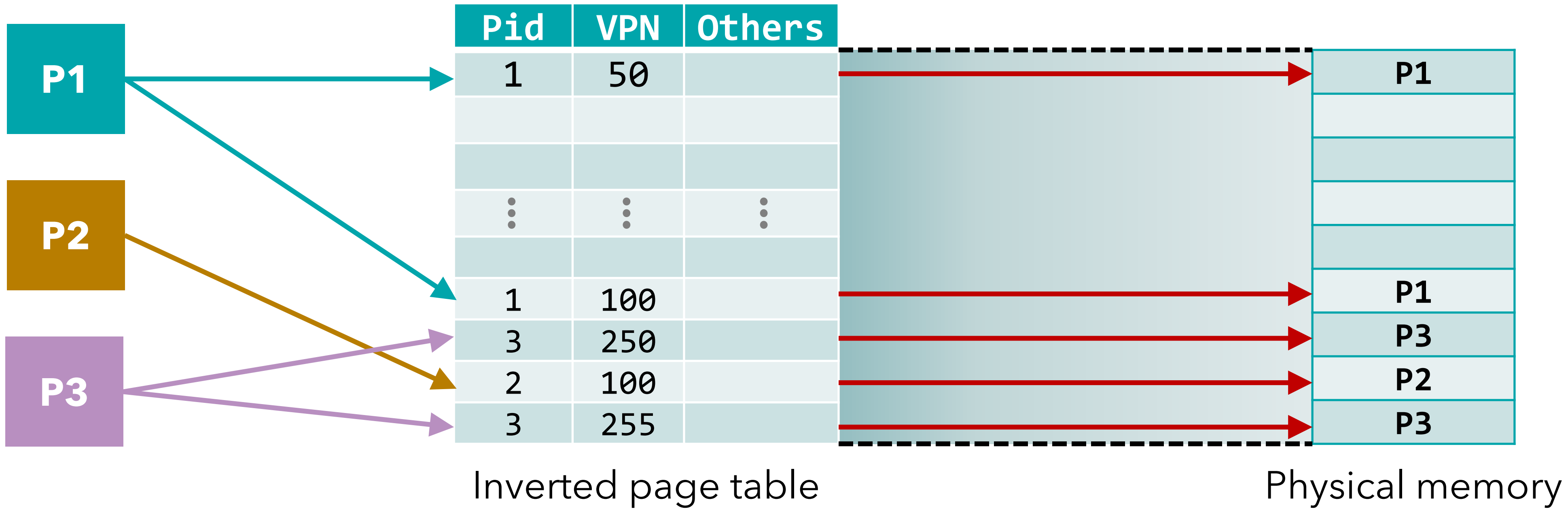
Pros and Cons of Multi-Level Paging

- In the paging, all the processes maintain their own page table
  - As the number of processes increases, the wasted memory space increases



Inverted Page Table: Overview

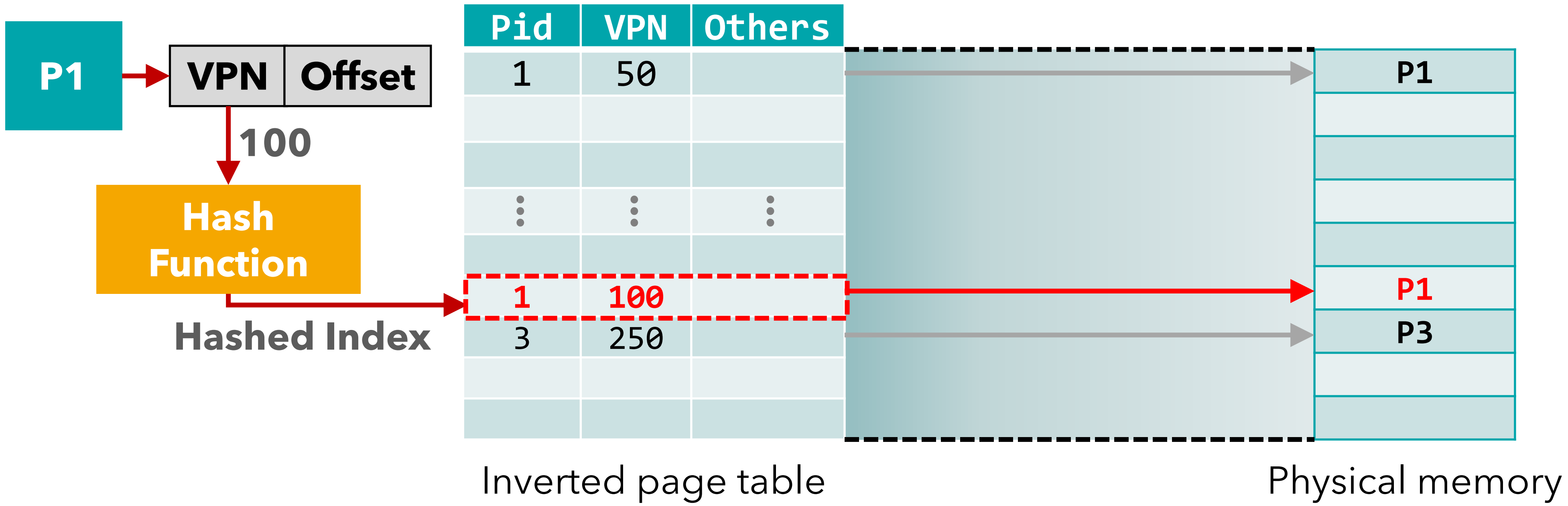
- Keeps a single page table that has a page table entry for each physical page of the system
    - Pros
        - Memory efficient – No need to maintain individual page tables for processes, thereby scaling with the size of physical memory



# of PTEs = # of Physical pages

Inverted Page Table: Mapping process

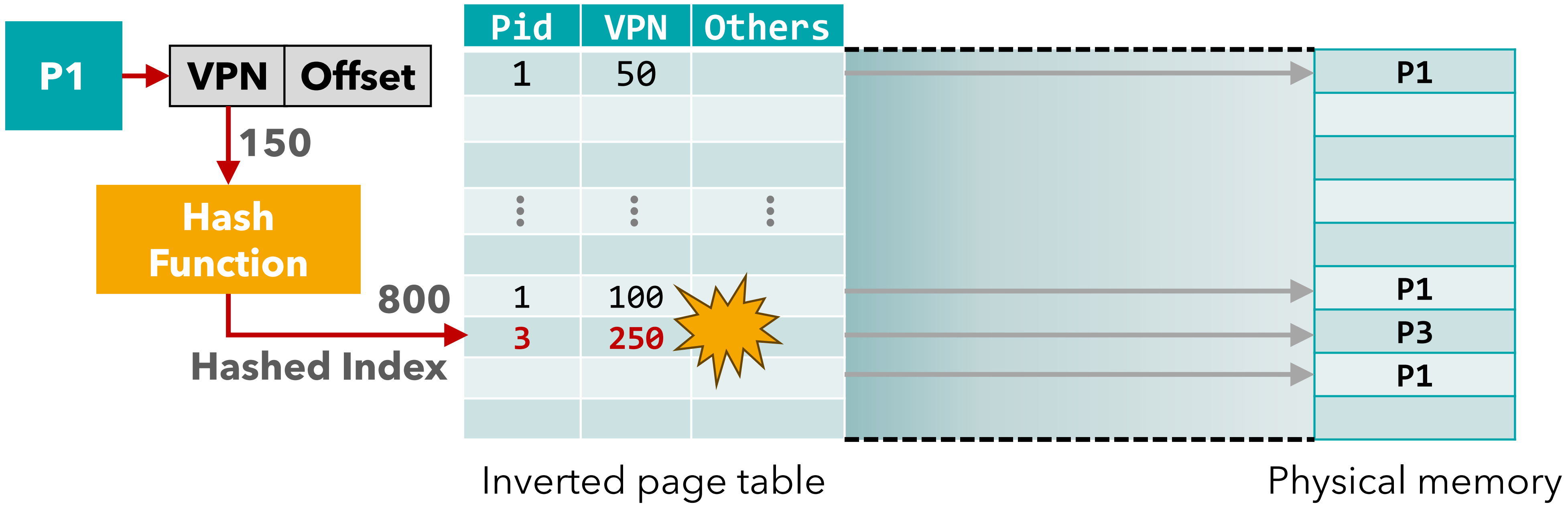
- A virtual address is hashed to a specific PTE in the table
- The index of the PTE is equal to the page frame number of the page it maps



Traversing inverted page table

Inverted Page Table: Mapping process

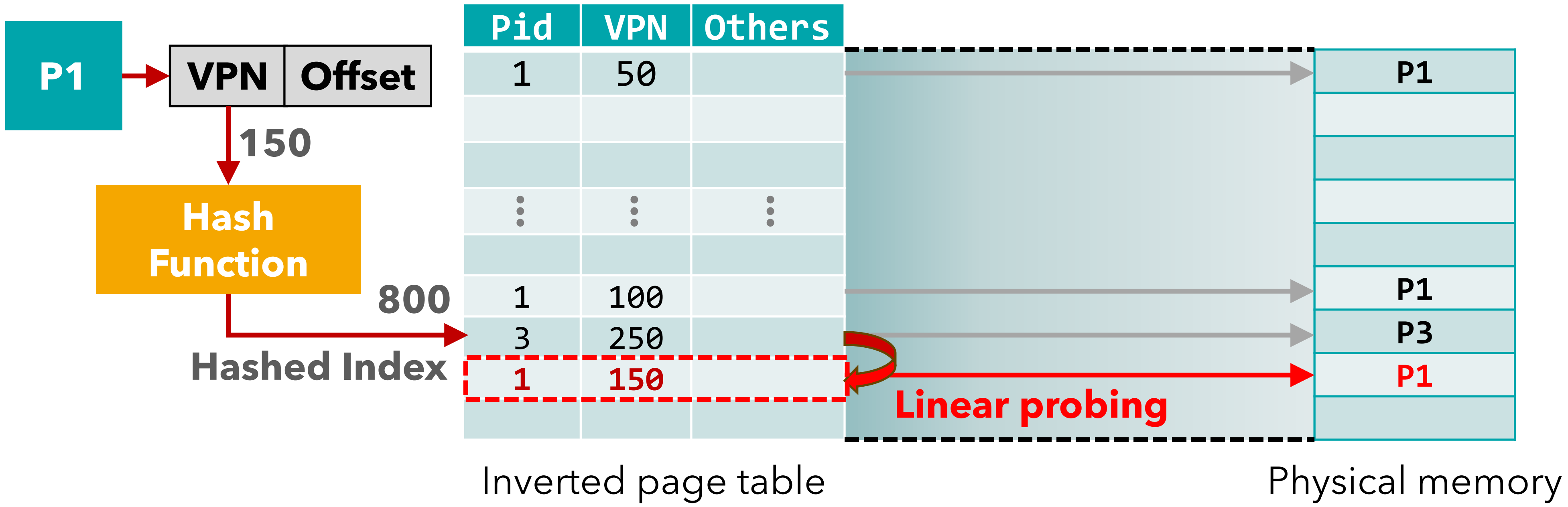
- A virtual address is hashed to a specific PTE in the table
  - **Hash collision** may occur (must handle!)
- The index of the PTE is equal to the page frame number of the page it maps



Traversing inverted page table

Inverted Page Table: Mapping Process

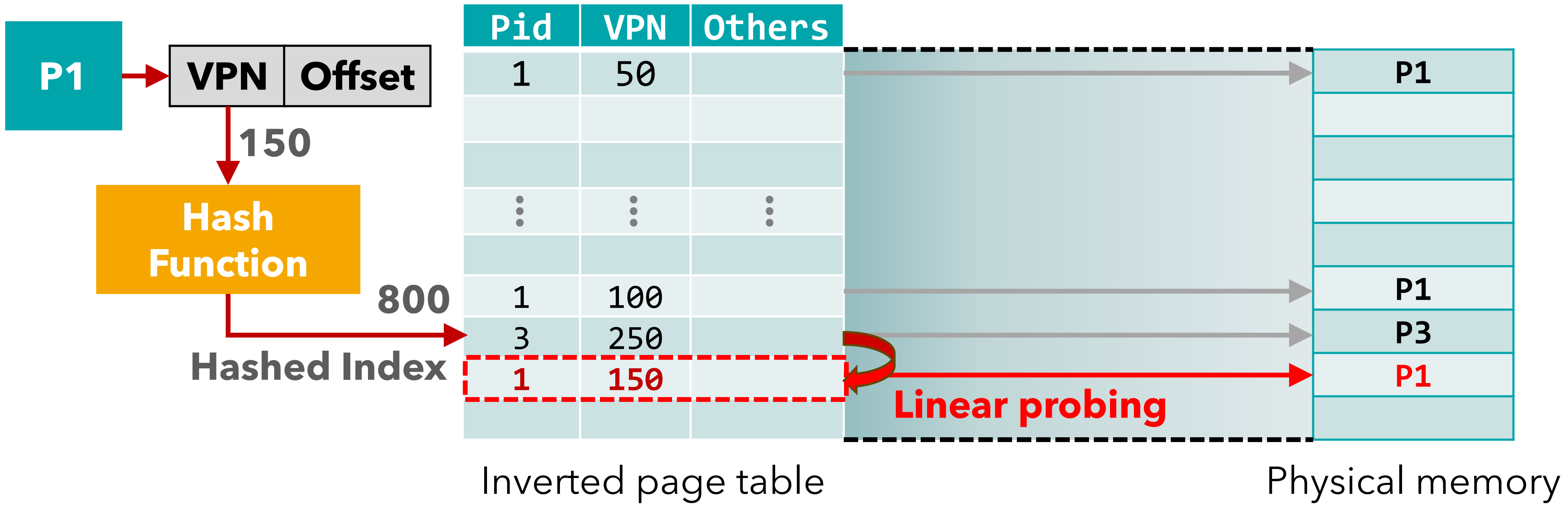
- A virtual address is hashed to a specific PTE in the table
  - **Hash collision** may occur (must handle!)
- The index of the PTE is equal to the page frame number of the page it maps



Traversing inverted page table

Inverted Page Table: Summary

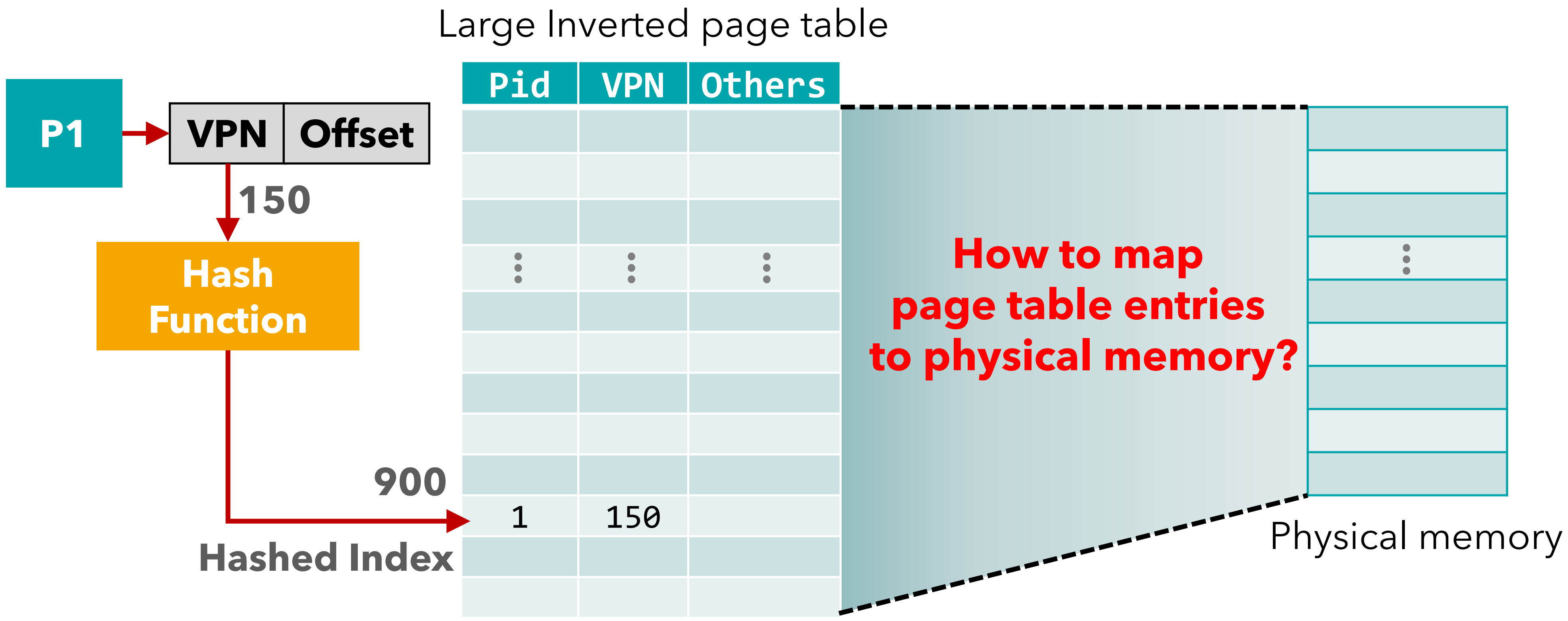
- Pros
  - Lower memory consumption
- Cons
  - Long tail latency by hash collision… **How to improve?**



Traversing inverted page table

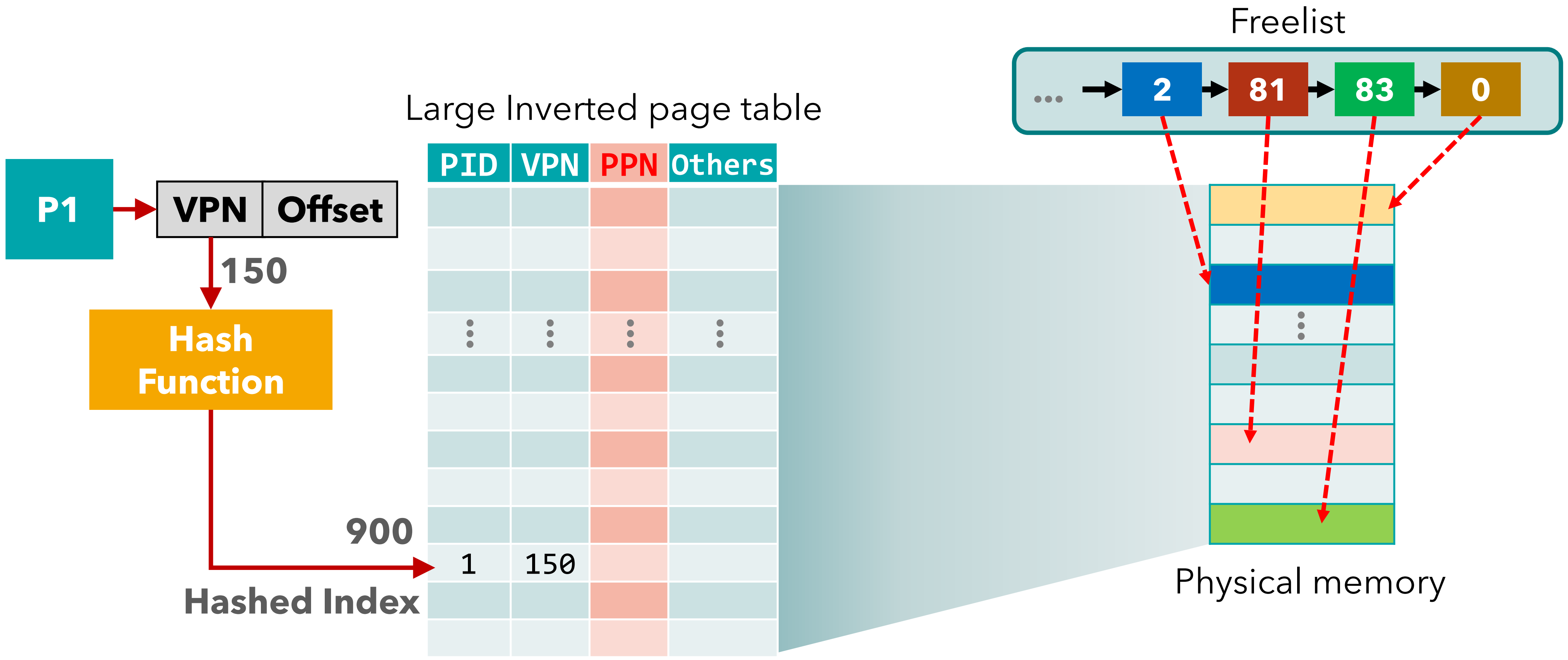
Large Inverted Page Table: Overview

- Main concept: Increase the number of entries in the inverted page table to reduce hash collision!
- Then, the index of the inverted page table will no longer be the physical page frame number


Large Inverted page table
P1
VPN
Offset
150
Hash Function
900
Hashed Index
Pid
VPN
Others
1
150
How to map page table entries to physical memory?
Physical memory

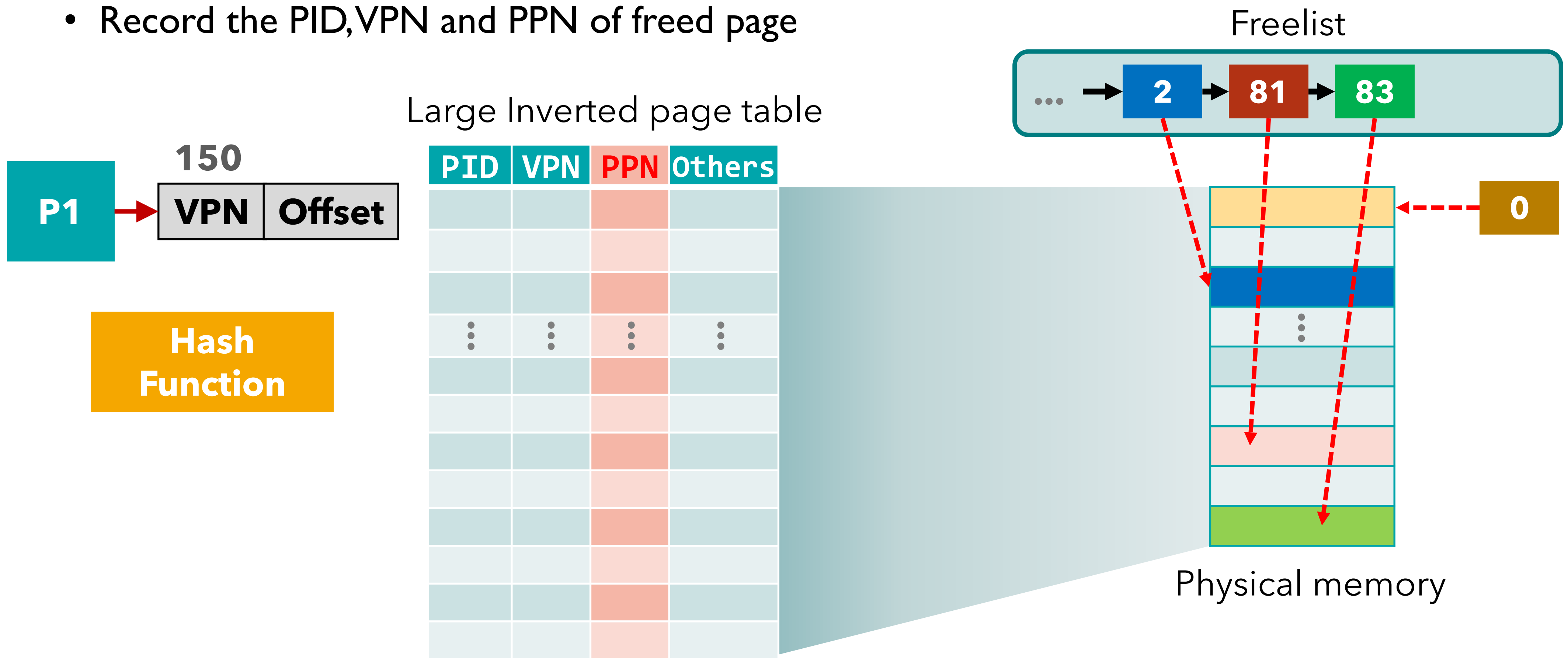
Large Inverted Page Table: Overview

- Difference from the original inverted page table
  - New column "**PPN**": Record the physical page number
  - **Freelist**: Linked list composed of pointers to free physical memory pages



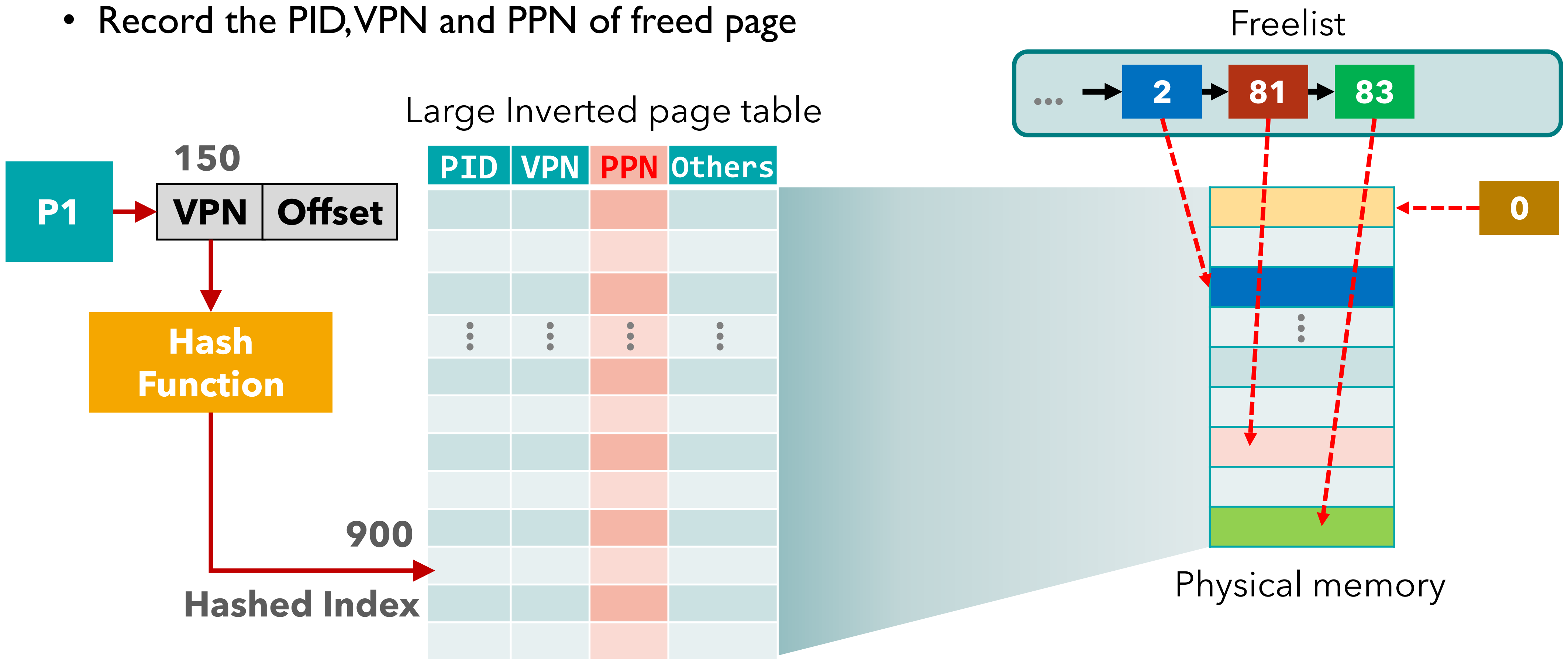
Large Inverted Page Table: Mapping Process

- Memory page allocation
  - **Remove the page at the tail of the freelist, freeing it for use**
  - Using hash function and linear probing, find the proper page table entry
  - Record the PID, VPN and PPN of freed page



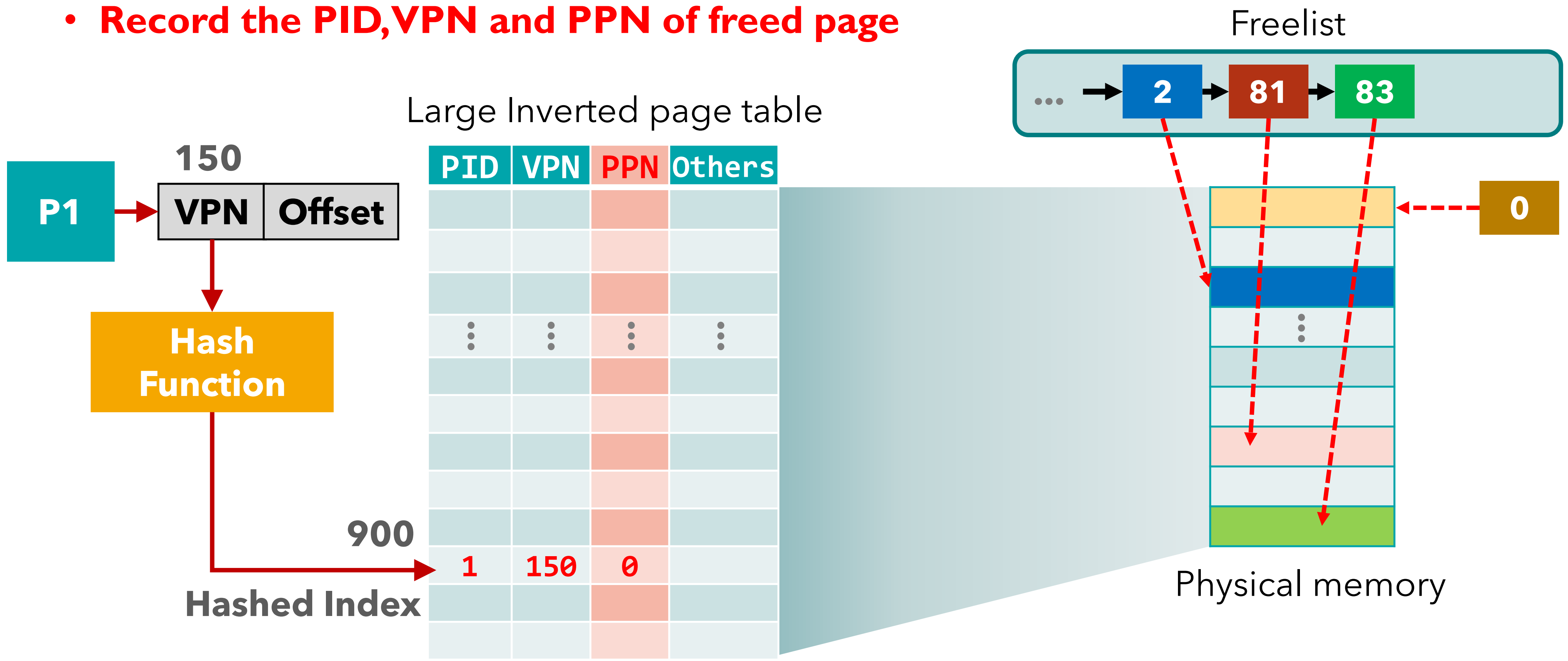
Large Inverted Page Table: Mapping Process

- Memory page allocation
  - Remove the page at the tail of the freelist, freeing it for use
  - **Using hash function and linear probing, find the proper page table entry**
  - Record the PID, VPN and PPN of freed page



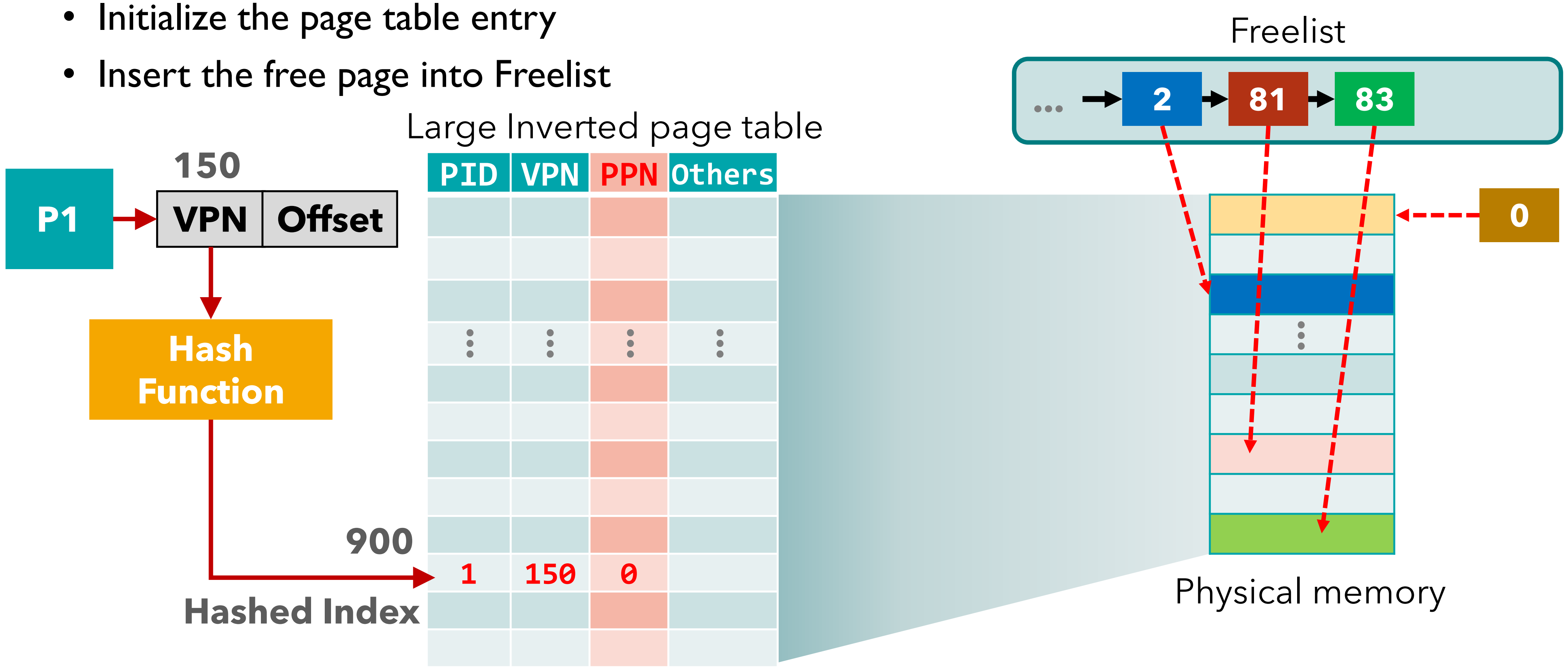
Large Inverted Page Table: Mapping Process

- Memory page allocation
  - Remove the page at the tail of the freelist, freeing it for use
  - Using hash function and linear probing, find the proper page table entry
  - **Record the PID, VPN and PPN of freed page**



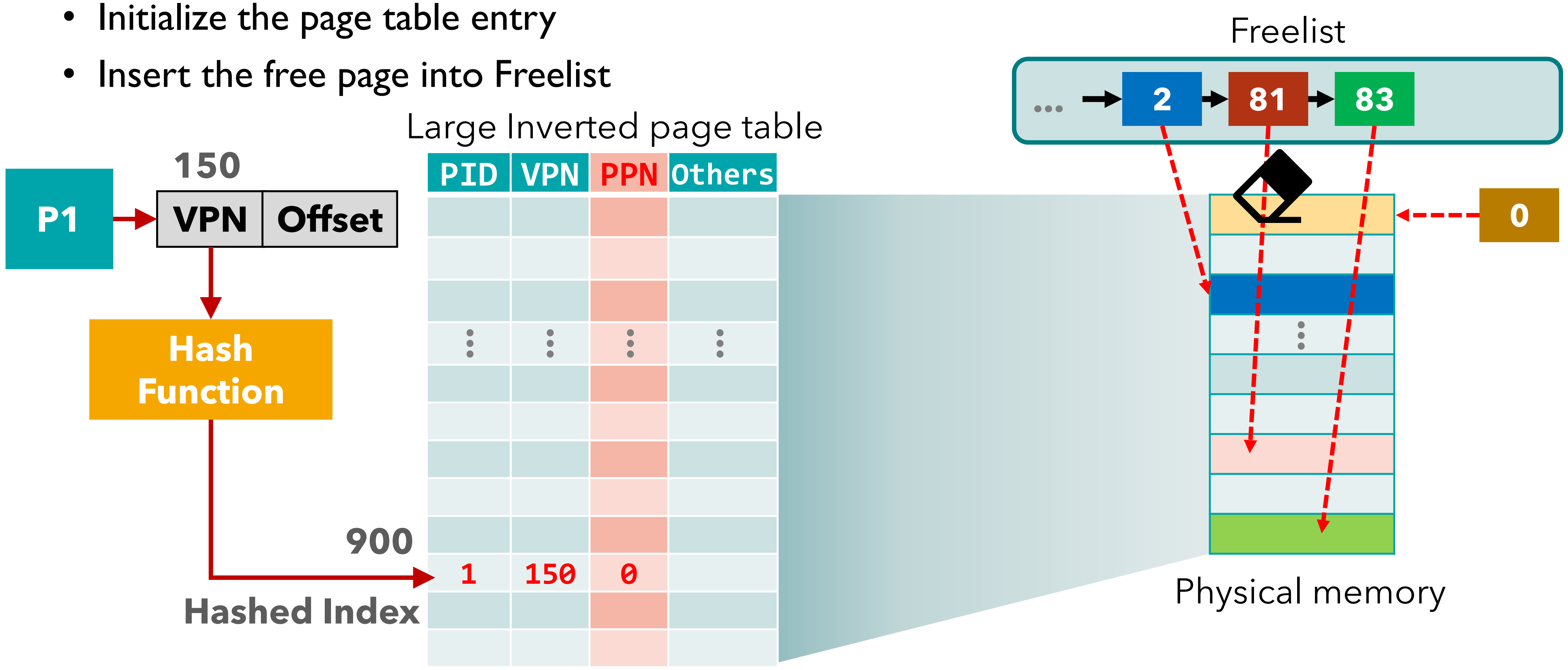
Large Inverted Page Table: Mapping Process

- Memory page deallocation (free)
  - **Find the PPN for given PID and VPN**
  - Erase the content of the corresponding page (memset)
  - Initialize the page table entry
  - Insert the free page into Freelist



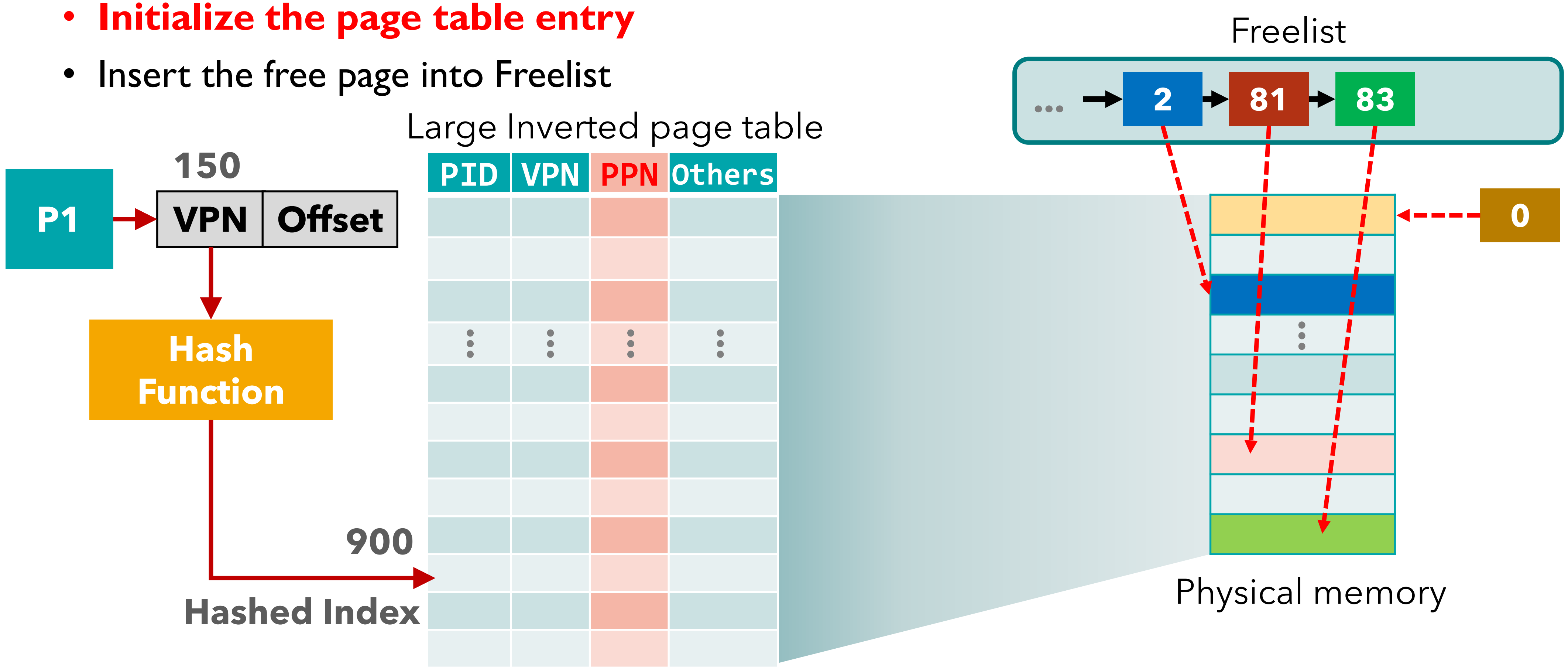
Large Inverted Page Table: Mapping Process

- Memory page deallocation (free)
  - Find the PPN for given PID and VPN
  - **Erase the content of the corresponding page (memset)**
  - Initialize the page table entry
  - Insert the free page into Freelist



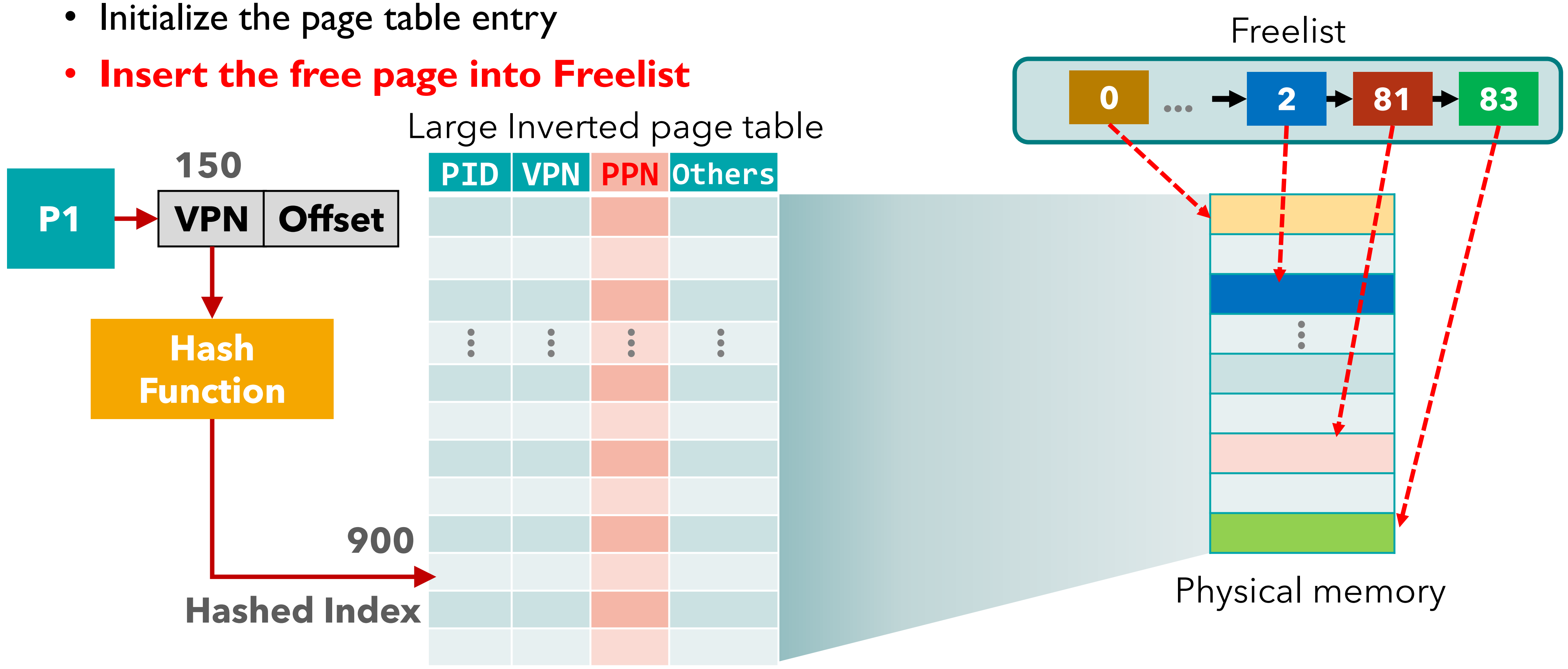
Large Inverted Page Table: Mapping Process

- Memory page deallocation (free)
  - Find the PPN for given PID and VPN
  - Erase the content of the corresponding page (memset)
  - **Initialize the page table entry**
  - Insert the free page into Freelist



Large Inverted Page Table: Mapping Process

- Memory page deallocation (free)
  - Find the PPN for given PID and VPN
  - Erase the content of the corresponding page (memset)
  - Initialize the page table entry
  - **Insert the free page into Freelist**



Project #2: Implementing Large Inverted Page Table

- Virtual Memory for supporting inverted page table
  - Allocating and Freeing
    - When kalloc() is called, given pid and virtual address, you must allocate the proper page
    - When kfree() is called, you must find the corresponding PTE and free that page
    - If hash collision occur, you must handle that by linear probing
    - Where to implement: **kalloc() and kfree()** function in kalloc.c and you can add functions in that code
  - Traverse
    - A virtual address is hashed to a specific PTE in the table
    - If hash collision occurs, perform linear probing
    - Where to implement: **ittraverse()** function in vm.c
  - Deallocate
    - When the process is terminated, free allocated pages and clear the corresponding entry
    - Where to implement: **deallocuvm()** function in vm.c

1. Allocating and Freeing Page

- There are four structures: the number of entries for each structure is 40000
  - PID[i]: Stores a PID whose process requests allocation of page i
  - VPN[i]: Stores a VPN that corresponds to page i
  - PPN[i]: Stores a PPN of physical page taken from Freelist
  - PTE_XV6[i]: This structure is for compatibility of xv6, and this structure stores the physical address with permission, (same as PTE of multi-level paging in original xv6)

- Allocation (kalloc(pid, v))
  - Get free page from freelist
  - Find the empty entry using hash function and linear probing
  - Record the PID, VPN, PPN into the page table entry
  - Return (char*)r; (r is the P2V(physical address))
  - Edge case: if v == -1, set vpn to P2V(physical address)
    - Here, v is the virtual address

1. Allocating and Freeing Page

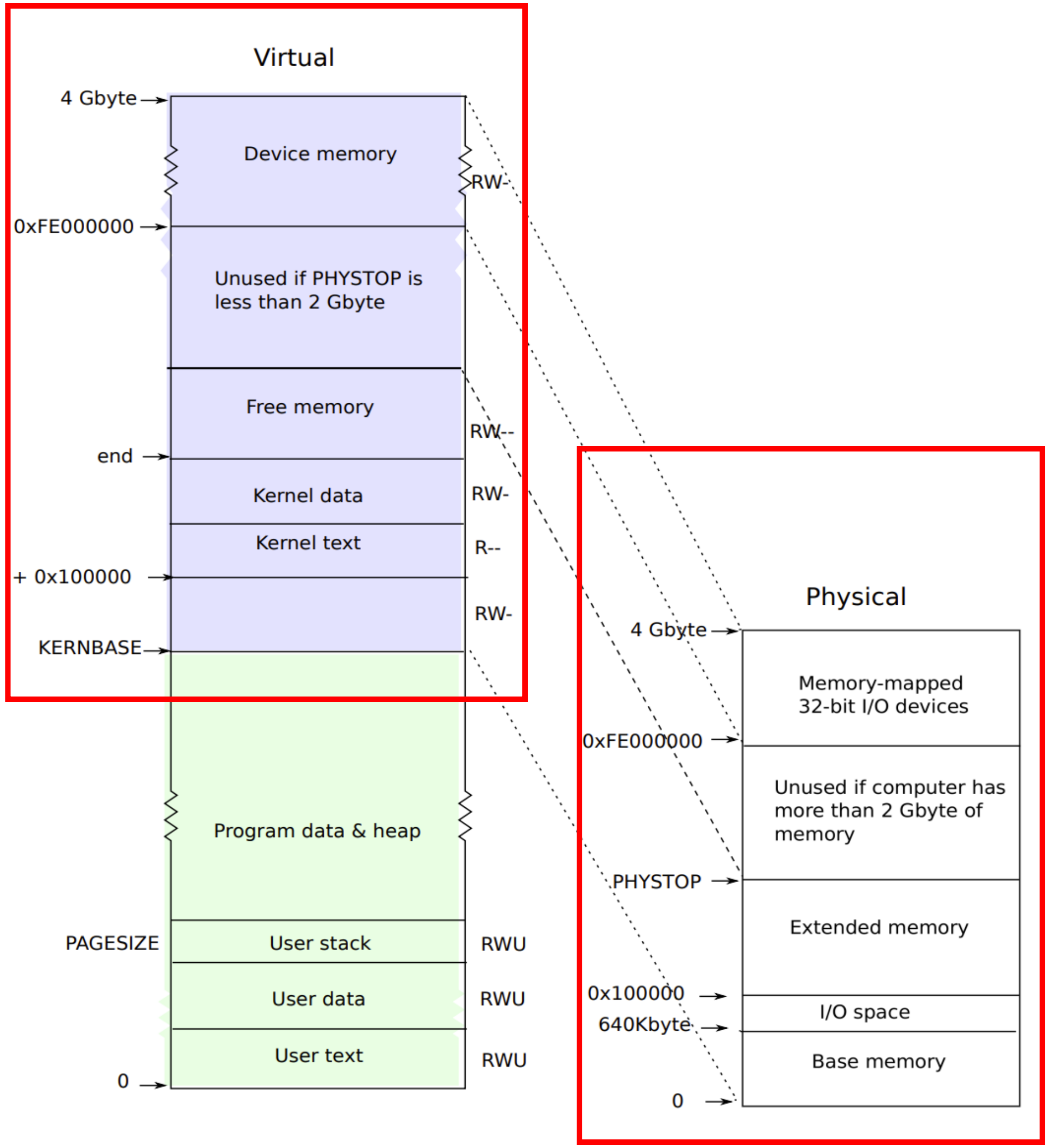
- There are four structures: the number of entries for each structure is 40000
  - PID[i]: Stores a PID whose process requests allocation of page i
  - VPN[i]: Stores a VPN that corresponds to page i
  - PPN[i]: Stores a PPN of physical page taken from Freelist
  - PTE_XV6[i]: This structure is for compatibility of xv6, and this structure stores the physical address with permission, (same as PTE of multi-level paging in original xv6)

- Deallocation (kfree)
  - Find the target entry using hash function and linear probing
  - Initialize the PID, VPN, PPN into the page table entry
  - Insert erased page into Freelist

2. Traverse

- You must consider Kernel address space
  - Kernel address space is directly mapped to physical memory space by subtracting VA from KERNBASE (0x8000000)

- **ittraverse()** function in vm.c
  - If VA >= KERNBASE:
    - Return &PTE_KERN[V2P(VA)/PGSIZE]
  - Else?
    - Return &PTE_XV6[idx]
    - Idx: corresponding index for given PID and VPN



Virtual and Physical memory space in xv6

2. Traverse

- **ittraverse()** function in vm.c
  - If VA > KERNBASE:
    - Return: &PTE_KERN[V2P(VA)]
  - Else?
    - **Very easy! You just traverse the inverted page table for given pid and virtual address (VR)**
      - Make a function for general purpose (searchit())
    - Return: &PTE_XV6[idx]

```
ittraverse()
{
if VA > KERNBASE:
        return &PTE_KERN[V2P(VA)/PGSIZE];

else:
        idx = searchit(va, pid);
        return &PTE_XV6[idx];
}
```

Pseudocode of ittraverse()

- Why use PTE_XV6?
  - In original xv6 code that support multi-level paging, the ittraverse() return page table entry itself, which can contain physical address and permission.
  - To minimize the modification of xv6, our implementation follows same format of output

3. Deallocating User Virtual Memory

- When you deallocate the user process's virtual memory space, you must find how the process requests allocation for kernel
  - For example, When process A has 10 allocated pages for kernel, you should find them in the inverted page table
  - **Deallocuvm() in vm.c**
    - For a given range, deallocate the previous-allocated pages within the range
    - How?
      - **By your own idea!**
        - **For example, traverse whole inverted page table and when the condition for deallocation, call kfree()**

Testing

- **To evaluate your implementation, you should perform usertests and hashtests**

- You should capture the result of usertests
  - Just type "usertests" in xv6 console

- In hashtests, linear probing occurs. You must capture the linear probing case
  - Print this information
    - Example (PID and VA for requests)
      - [Hash collision for idx: 2] PID: 0, VA: 0x100000, VA is different
      - [Hash collision for idx: 3] PID: 0, VA: 0x100000, PID is different
      - [Hash collision for idx: 4] PID: 0, VA: 0x100000, PID is different
      - [Completion idx: 5] PID: 0, VA: 0x100000
  - Change the MAXENTRY from 50000 to 70000 (in param.h)
    - Perform hashtests and analyze the impact of inverted page table size

Summary

- **You should modify allocation, deallocation, traversal code to support Inverted page table in context of user's process**
  - If needed, you can add functions
  - kalloc(), kfree(), ittraverse(), deallocuvm()
- **To evaluate your implementation, you should perform usertests and hashtests()**
- Fill in the code wherever it contains *//TODO*

Project #2 – Inverted Page Table

- Deadline
  - ~ 2024.11.25 (Mon) 23:59
- Project repo. Generation
  - mkdir os-pj2; cd os-pj2; git clone https://github.com/dgist-datalab/xv6; cd xv6; git checkout inverted
- Hand-in procedure
  - p2_201812345.patch
  - Run the following command and upload p2_201812345.patch
    - git diff > p2_201812345.patch
  - Check the patch file with Notepad and confirm your modifications are in the patch file
  - Report (Important!)
    - Submit an 1~3 pages report
      - Free format (Korean/English)
      - Description of your implementation in detail (kalloc, kfree, ittraverse, deallocuvm)
      - Explain your test code and answer the following two tests and explain why the results are as follows
      - Insert test code result image

Finally …

<u>Do NOT hesitate </u>to ask questions!

Mini Project #1, #2	Juhyung Park	*arter97@dgist.ac.kr*
Project #1	Minjae Kim	*jwya2149@dgist.ac.kr*
Project #2	**Seonggyun Oh**	***<u>sungkyun123@dgist.ac.kr</u>***
Project #3	Jeeyun Kim	*kimgyun@dgist.ac.kr*