Pintos Lab3a/3b Overview

TA: Yinmin Zhong

Stolen from Spring'22 slides made by Zhineng Zhong

Today

- Lab 3 overview

- Lab 3a/3b tasks

- Bitmap and hash

- Tips

- Q&A

Today

- **Lab 3 overview**

- Lab 3a/3b tasks

- Bitmap and hash

- Tips

- Q&A

Virtual Memory Overview

- a *memory management* technique

- *an idealized abstraction of the storage resources*

- address translation using a combination of hardware and software

- easy to implement linking

- easy to implement loading

- easy to implement sharing

- increased security due to memory isolation

- …

Much of our focus

- **Virtual memory management**

  - Virtual pages

- **Physical memory management**

  - Physical frames

  - Replacement policy

- **Page table**

  - Address translation

  - Page fault

- **Swap Space**

Before we dive in …

- A good design goes a long way towards accomplishing your tasks and has a long way to go.

- Get your hands on early.

This assignment is an open-ended design problem.

- We are going to say as little as possible about how to do things. Instead we will focus on what functionality we require your OS to support.

- We will expect you to come up with a design that makes sense. You will have the freedom to choose how to handle page faults, how to organize the swap partition, how to implement paging, etc.

Today

- Lab 3 overview

- **Lab 3a/3b tasks**

- Bitmap and hash

- Tips

- Q&A

A brief look at lab 3a tasks

✓ **Exercise 1.1**

Implement paging for segments loaded from executables.
- All of these pages should be loaded **lazily**, that is, only as the kernel intercepts page faults for them.
- Upon eviction:
  - Pages **modified** since load (e.g. as indicated by the "dirty bit") should **be written to swap**.
  - **Unmodified** pages, including read-only pages, should **never be written to swap** because they can always be read back from the executable.

✓ **Exercise 1.2**

Implement a global page replacement algorithm that approximates LRU.
- Your algorithm should perform **at least as well as** the simple variant of the "second chance" or "clock" algorithm.

✓ **Exercise 2.1**

Adjust user memory access code in *system call handling* to deal with potential page faults.

- 1.1
  - load an executable
  - lazy loading
  - write back
  - swap space
- 1.2
  - replacement policy
- 1.3
  - modify previous implementation

A brief look at lab 3b tasks

Exercise 1.1

Implement stack growth.

- In project 2, the stack was **a single page** at the top of the user virtual address space, and programs were limited to that much stack.
- Now, **if the stack grows past its current size, allocate additional pages as necessary**.

Exercise 2.1

Implement memory mapped files, including the following system calls.

- mapid_t **mmap** (int fd, void *addr)
- void **munmap** (mapid_t mapping)

- 1.1
  - allocate a new page when stack grows past current page

- 1.2
  - mmap, munmap
  - can map many kinds of files

Let's look at the tasks in another way…

When we execute a program…
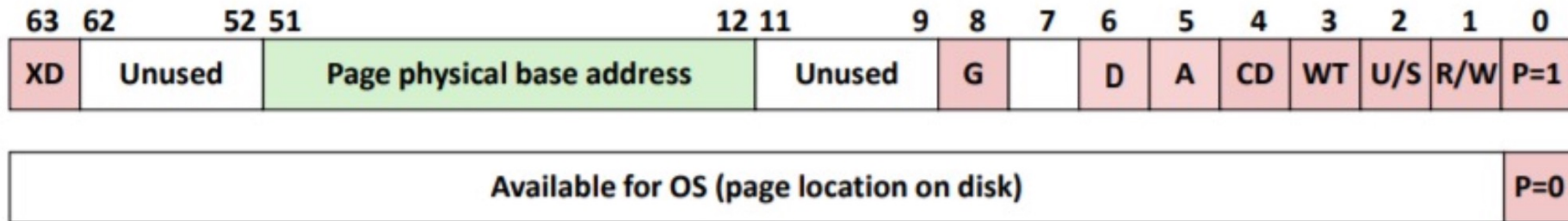
- **When we load an executable…**

  - Only build the mappings

    - map user pages to file content
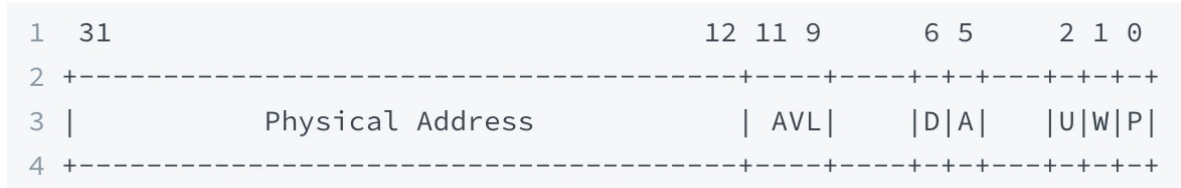
    - *Q: How to maintain the mappings?*

      - first access will trigger page fault and then load the page

      - Store disk address in PTE like we have already known?

63	62	...	52	51	...	12	11	...	9	8	7	6	5	4	3	2	1	0
XD	Unused			Page physical base address			Unused			G		D	A	CD	WT	U/S	R/W	P=1

Available for OS (page location on disk)	P=0

When we execute a program…

- What do we know about Pintos PTE?

  - Check lab document

```
1   31                                          12 11 9      6 5      2 1 0
2   +-------------------------------------------+----+----+-+-+---+-+-+-+
3   |               Physical Address            | AVL|    |D|A|   |U|W|P|
4   +-------------------------------------------+----+----+-+-+---+-+-+-+
```

- **Macro: PTE_P**
  - **Bit 0, the "present" bit.**
  - When this bit is 1, the other bits are interpreted as described below. When this bit is 0, any attempt to access the page will page fault. The remaining bits are then not used by the CPU and may be used by the OS for any purpose.

- OK! It is designed as we expected.

- Do you think storing information in PTE is *a good design* and *easy to implement* in THIS PINTOS LAB?

  - 31-bit might be a challenge?

  - You decide.

When we execute a program…

- **When we load an executable…**

  - Only build the mappings

    - map user pages to file content

    - *Q: How to maintain the mappings?*

      - first access will trigger page fault

      - **Idea 1:** Store disk address in PTE like we have already known?

      - **Idea 2:** You can design other data structures and stored in *thread*

        - Use these newly designed data structures when handling page fault

When we execute a program…

- **When we access it…**

  - Trigger page fault on first access

  - Turn to page fault handler to load the page

    - *Q: How can we know it's time to load the page instead of other cases?*

    - *Q: How to load the page?*

      - Read the page into memory

        - **Replacement policy**

    - Update pte etc.

      - Interfaces in ***threads/pte.h, userprog/pagedir.h, userprog/pagedir.c …***
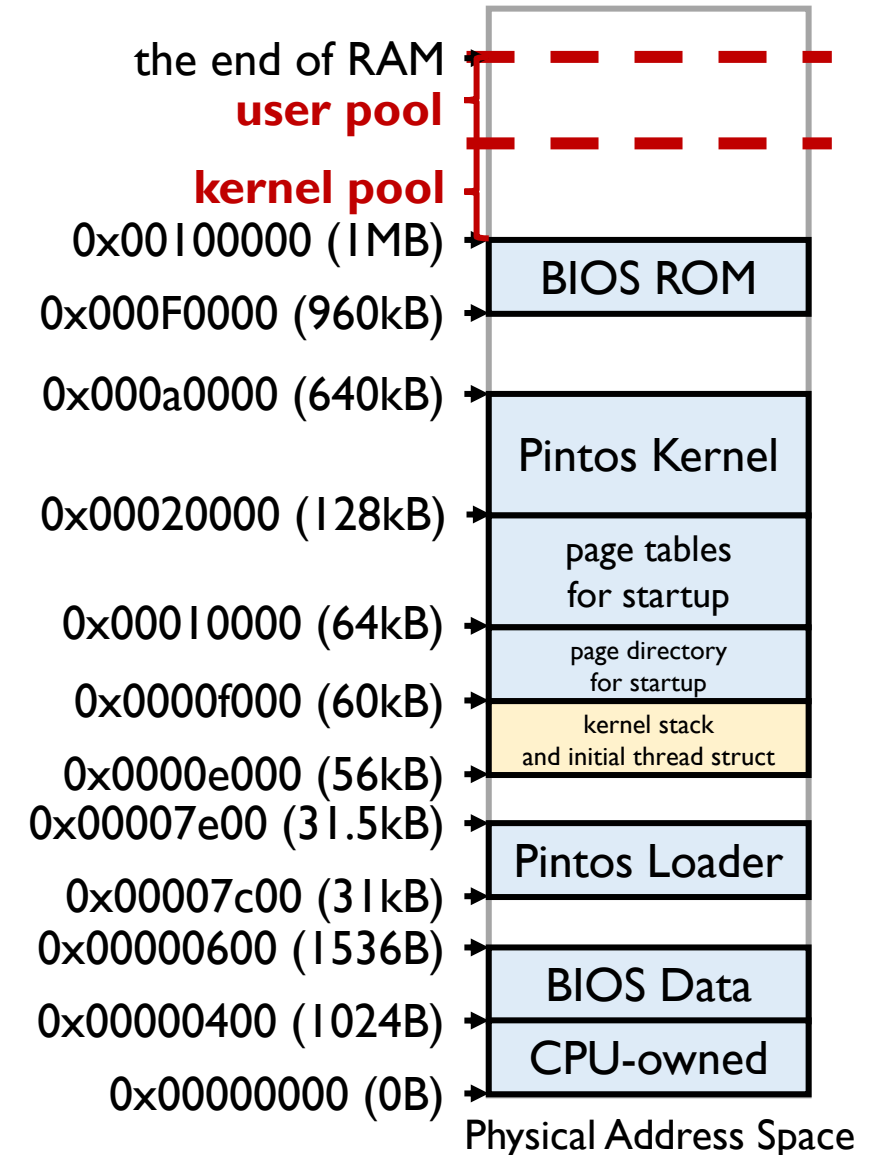
Replacement policy

- Require one that approximates LRU

  - second chance, clock, …

  - some interfaces may be useful: ***userprog/pagedir.h***

```
bool pagedir_is_dirty (uint32_t *pd, const void *upage);
void pagedir_set_dirty (uint32_t *pd, const void *upage, bool dirty);
bool pagedir_is_accessed (uint32_t *pd, const void *upage);
void pagedir_set_accessed (uint32_t *pd, const void *upage, bool accessed);
```

- All algorithms have a common thing to do…

- *Q: How to manage the frames and their related information?*

  - Frame management

Frame management

- *Q: What frames do we need to manage?*

- *Claim: In lab3, you only need to manage frames in user pool.*

  - Why?
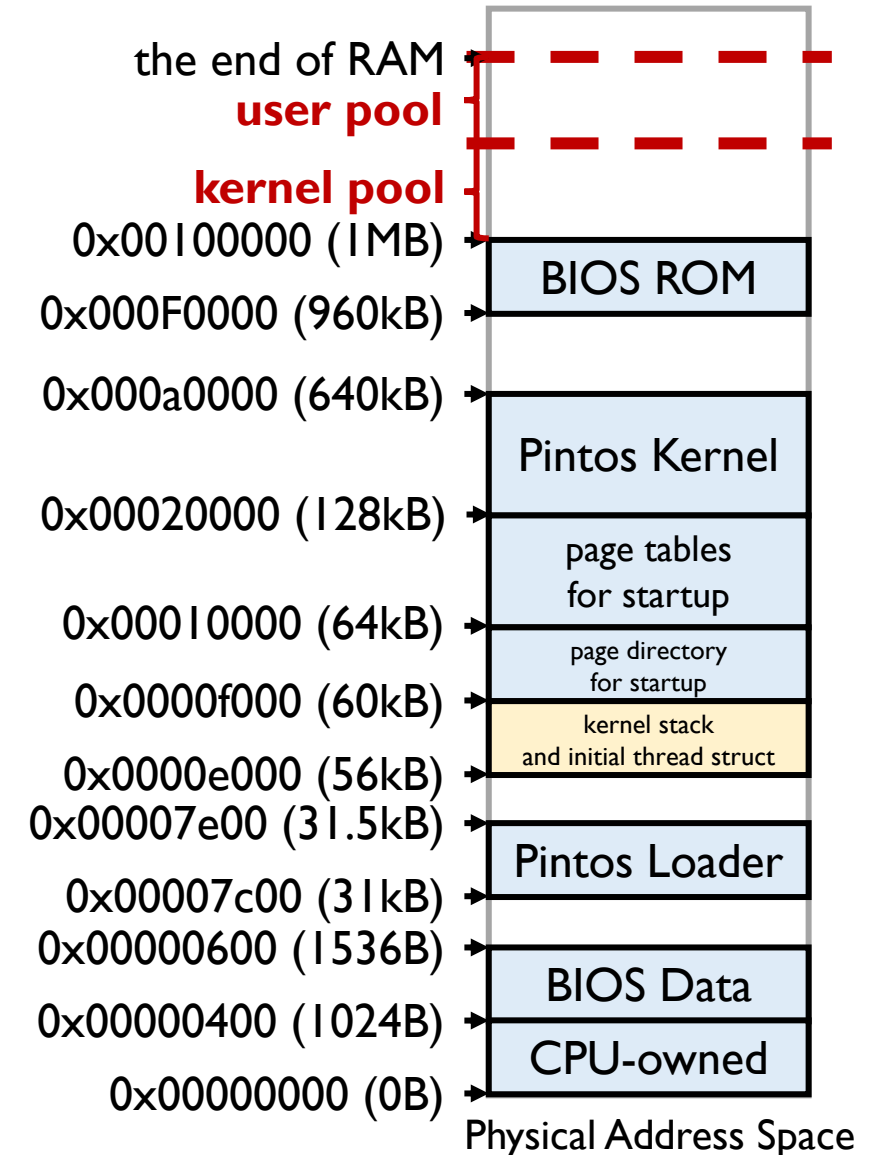
  - Let's look at the definitions of the two pools again.

the end of RAM	
user pool	
kernel pool	
0x00100000 (1MB)	BIOS ROM
0x000F0000 (960kB)	
0x000a0000 (640kB)	
	Pintos Kernel
0x00020000 (128kB)	
	page tables for startup
0x00010000 (64kB)	
	page directory for startup
0x0000f000 (60kB)	
	kernel stack and initial thread struct
0x0000e000 (56kB)	
0x00007e00 (31.5kB)	
	Pintos Loader
0x00007c00 (31kB)	
0x00000600 (1536B)	
	BIOS Data
0x00000400 (1024B)	
	CPU-owned
0x00000000 (0B)	

Physical Address Space

Frame management

The user pool should be used for allocating memory for user processes and *the kernel pool* for all other allocations. This will only become important starting with project 3. Until then, all allocations should be made from *the kernel pool*.

- The fact that an executable or another kind of file is mapped to user space indicates that *you can only load it or reload it into user pool*.

- No allocation in user pool before lab3 means that if you manage all frames in user pool you don't need to worry about evicting a frame that you don't know.
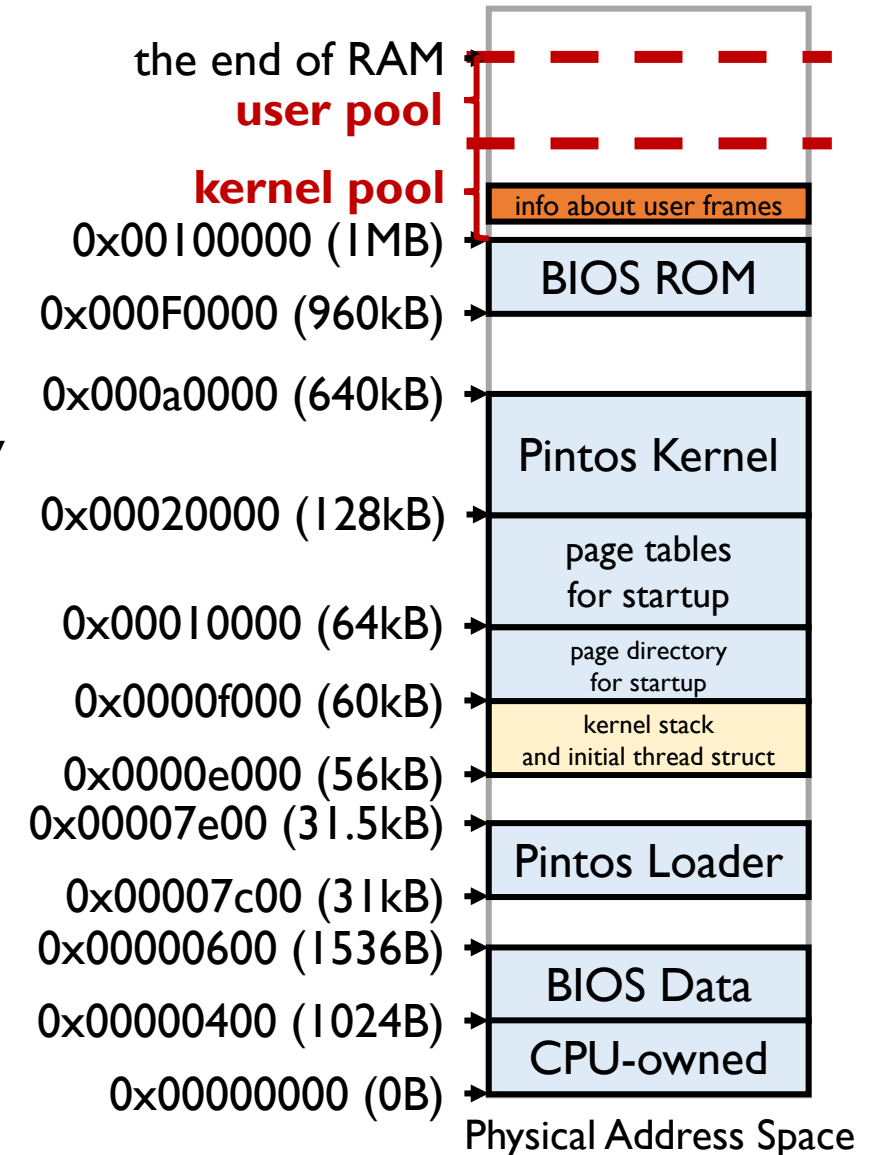
Frame management

- *Q: What frames do we need to manage?*

- *Claim: In lab3, you only need to manage frames in user pool.*

- So there will be no replacement in kernel pool?

  - We use *palloc* to allocate memory in kernel pool.

  - At least our tests will not ask for that much memory allocation in kernel pool. **So you don't need to consider that.**

  - **BUT THIS IS ACTUALLY A PROBLEM.**

the end of RAM →
user pool
kernel pool →
0x00100000 (1MB) →

BIOS ROM

0x000F0000 (960kB) →

0x000a0000 (640kB) →

Pintos Kernel

0x00020000 (128kB) →

page tables
for startup

0x00010000 (64kB) →

page directory
for startup

0x0000f000 (60kB) →

kernel stack
and initial thread struct

0x0000e000 (56kB) →
0x00007e00 (31.5kB) →

Pintos Loader

0x00007c00 (31kB) →

0x00000600 (1536B) →

BIOS Data

0x00000400 (1024B) →

CPU-owned

0x00000000 (0B) →

Physical Address Space

Frame management

- **Q: What frames do we need to manage?**

- ***Claim: In lab3, you only need to manage frames in user pool.***

- So what's good about this claim?

  - The data not in user pool resides in physical memory

  - Feel free to allocate your global data structures in kernel pool and use them to manage all the user frames!

    - Initialization can be done in *pintos_init*

the end of RAM

user pool

kernel pool

info about user frames

0x00100000 (1MB)

BIOS ROM

0x000F0000 (960kB)

0x000a0000 (640kB)

Pintos Kernel

0x00020000 (128kB)

page tables
for startup

0x00010000 (64kB)

page directory
for startup

0x0000f000 (60kB)

kernel stack
and initial thread struct

0x0000e000 (56kB)

0x00007e00 (31.5kB)

Pintos Loader

0x00007c00 (31kB)

0x00000600 (1536B)

BIOS Data

0x00000400 (1024B)

CPU-owned

0x00000000 (0B)

Physical Address Space

Replacement policy

- Require one that approximates LRU

  - second chance, clock, …

  - some interfaces may be useful: *userprog/pagedir.h*

```c
bool pagedir_is_dirty (uint32_t *pd, const void *upage);
void pagedir_set_dirty (uint32_t *pd, const void *upage, bool dirty);
bool pagedir_is_accessed (uint32_t *pd, const void *upage);
void pagedir_set_accessed (uint32_t *pd, const void *upage, bool accessed);
```

- All algorithms have a common thing to do…

- *Q: How to manage the frames and their related information?*

  - Frame management

  - Swap space management

Swap space management

- Swap space is a space on a hard disk that is a substitute for physical memory.

- In pintos, swap space is a **block device**.

- *The interfaces of block device have been provided.*

```c
/** A block device. */
struct block
  {
    struct list_elem list_elem;        /**< Element in all_blocks. */

    char name[16];                     /**< Block device name. */
    enum block_type type;              /**< Type of block device. */
    block_sector_t size;               /**< Size in sectors. */

    const struct block_operations *ops;  /**< Driver operations. */
    void *aux;                         /**< Extra data owned by driver. */

    unsigned long long read_cnt;       /**< Number of sectors read. */
    unsigned long long write_cnt;      /**< Number of sectors written. */
  };
```

Swap space management

- Block device

  - *devices/block.h, devices/block.c*

  - Initialized in *pintos_init*

  - **Get a block device**

    - block_get_role

  - **Read from a block device**

    - block_read

  - **Write to a block device**

    - block_write

  - Other interfaces you may use

```
/** Finding block devices. */
struct block *block_get_role (enum block_type);
void block_set_role (enum block_type, struct block *);
struct block *block_get_by_name (const char *name);
```

```
/** Type of a block device. */
enum block_type
  {
    /* Block device types that play a role in Pintos. */
    BLOCK_KERNEL,                /**< Pintos OS kernel. */
    BLOCK_FILESYS,               /**< File system. */
    BLOCK_SCRATCH,               /**< Scratch. */
    BLOCK_SWAP,                  /**< Swap. */
    BLOCK_ROLE_CNT,
```

```
/** Block device operations. */
block_sector_t block_size (struct block *);
void block_read (struct block *, block_sector_t, void *);
void block_write (struct block *, block_sector_t, const void *);
const char *block_name (struct block *);
enum block_type block_type (struct block *);
```

Good!

Swap space management

- Are all these interfaces enough for our task?

- How to manage swap space?

When we execute a program...

- **When we access it...**

  - Trigger page fault on first access

  - Turn to page fault handler to load the page

    - *Q: How can we know it's time to load the page instead of other cases?*

    - *Q: How to load the page?*

      - Read the page into memory

        - **Replacement policy**

      - Update pte

        - Interfaces in ***threads/pte.h, userprog/pagedir.h, userprog/pagedir.c …***

When we execute a program…

- **When we read it…**

  - …

- **When we write it…**

  - write back instead of write through

  - dirty bit

  - some interfaces may be useful: *userprog/pagedir.h*

```c
bool pagedir_is_dirty (uint32_t *pd, const void *upage);
void pagedir_set_dirty (uint32_t *pd, const void *upage, bool dirty);
bool pagedir_is_accessed (uint32_t *pd, const void *upage);
void pagedir_set_accessed (uint32_t *pd, const void *upage, bool accessed);
```

Revisit lab 3a tasks

Exercise 1.1

Implement paging for segments loaded from executables.
- All of these pages should be loaded **lazily**, that is, only as the kernel intercepts page faults for them.
- Upon eviction:
  - Pages **modified** since load (e.g. as indicated by the "dirty bit") should **be written to swap**.
  - **Unmodified** pages, including read-only pages, should **never be written to swap** because they can always be read back from the executable.

✓ **Exercise 1.2**

Implement a global page replacement algorithm that approximates LRU.
- Your algorithm should perform **at least as well as** the simple variant of the "second chance" or "clock" algorithm.

✓ **Exercise 2.1**

Adjust user memory access code in *system call handling* to deal with potential page faults.

- 1.1
  - load an executable
  - lazy loading
  - write back
  - swap space

- 1.2
  - replacement policy

- 1.3
  - modify previous implementation

Revisit lab 3b tasks

- 1.1
  - allocate a new page when stack grows past current page

- 1.2
  - mmap, munmap
  - can map many kinds of files

Key parts

- **User pages mapping maintenance**

- **Frame management**

- **Swap space management**

Today

- Lab 3 overview

- Lab 3a/3b tasks

- **Bitmap and hash**

- Tips

- Q&A

Bitmap

- *lib/kernel/bitmap.h, lib/kernel/bitmap.c*

- Bitmap is an effective way to make marks.

- Bitmap is used in memory *pool* to mark whether the pages in pool are used.

```c
/** A memory pool. */
struct pool
  {
    struct lock lock;              /**< Mutual exclusion. */
    struct bitmap *used_map;       /**< Bitmap of free pages. */
    uint8_t *base;                 /**< Base of pool. */
  };
```

```c
/** Creation and destruction. */
struct bitmap *bitmap_create (size_t bit_cnt);
struct bitmap *bitmap_create_in_buf (size_t bit_cnt, void *, size_t byte_cnt);
size_t bitmap_buf_size (size_t bit_cnt);
void bitmap_destroy (struct bitmap *);

/** Bitmap size. */
size_t bitmap_size (const struct bitmap *);
```

```c
/** Setting and testing single bits. */
void bitmap_set (struct bitmap *, size_t idx, bool);
void bitmap_mark (struct bitmap *, size_t idx);
void bitmap_reset (struct bitmap *, size_t idx);
void bitmap_flip (struct bitmap *, size_t idx);
bool bitmap_test (const struct bitmap *, size_t idx);

/** Setting and testing multiple bits. */
void bitmap_set_all (struct bitmap *, bool);
void bitmap_set_multiple (struct bitmap *, size_t start, size_t cnt, bool);
size_t bitmap_count (const struct bitmap *, size_t start, size_t cnt, bool);
bool bitmap_contains (const struct bitmap *, size_t start, size_t cnt, bool);
bool bitmap_any (const struct bitmap *, size_t start, size_t cnt);
bool bitmap_none (const struct bitmap *, size_t start, size_t cnt);
bool bitmap_all (const struct bitmap *, size_t start, size_t cnt);
```

Hash

- ***lib/kernel/hash.h, lib/kernel/hash.c***

- The element stored in the hash table can be quickly retrieved.

- Like *list_elem* for *List*, *hash_elem* is used in *hash*.

```
/** Basic life cycle. */
bool hash_init (struct hash *, hash_hash_func *, hash_less_func *, void *aux);
void hash_clear (struct hash *, hash_action_func *);
void hash_destroy (struct hash *, hash_action_func *);

/** Search, insertion, deletion. */
struct hash_elem *hash_insert (struct hash *, struct hash_elem *);
struct hash_elem *hash_replace (struct hash *, struct hash_elem *);
struct hash_elem *hash_find (struct hash *, struct hash_elem *);
struct hash_elem *hash_delete (struct hash *, struct hash_elem *);


/** Iteration. */
void hash_apply (struct hash *, hash_action_func *);
void hash_first (struct hash_iterator *, struct hash *);
struct hash_elem *hash_next (struct hash_iterator *);
struct hash_elem *hash_cur (struct hash_iterator *);

/** Information. */
size_t hash_size (struct hash *);
bool hash_empty (struct hash *);
```

```
/** Computes and returns the hash value for hash element E, given
    auxiliary data AUX. */
typedef unsigned hash_hash_func (const struct hash_elem *e, void *aux);

/** Compares the value of two hash elements A and B, given
    auxiliary data AUX.  Returns true if A is less than B, or
    false if A is greater than or equal to B. */
typedef bool hash_less_func (const struct hash_elem *a,
                             const struct hash_elem *b,
                             void *aux);
```

```
/** Hash element. */
struct hash_elem
  {
    struct list_elem list_elem;
  };
```

How to use hash table to implement a fast mapping (hash map)

- Given a key *k*, get its correspondent value *v* efficiently

- This is quite helpful in implementing our mapping from user pages to swap space content or file content.

- **Just use hash table to maintain keys, and combine correspondent values with keys.**

  - e.g. put key and value in a struct, and use key as the hash key
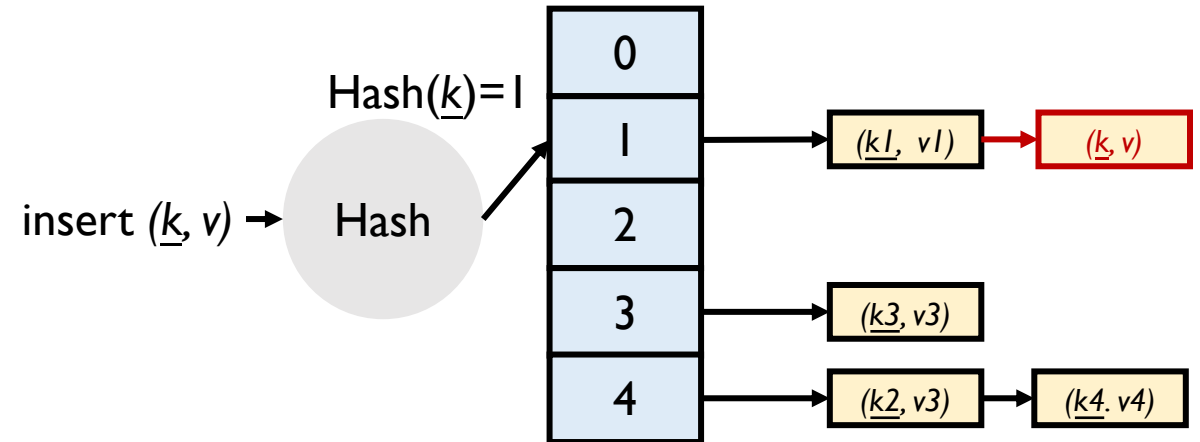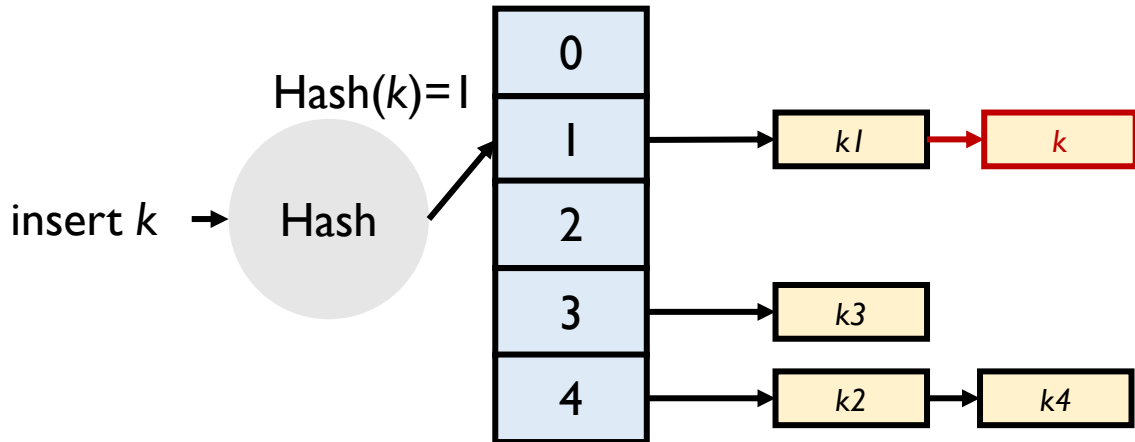
Today

- Lab 3 overview

- Lab 3a/3b tasks

- Bitmap and hash

- **Tips**

- Q&A

Other things you may need to give attention to…

- **Synchronization problem**

  - especially when accessing some global variables

- **Remember to free the resources when the process exits**

- **Note that a disk sector is *512B* while a page is *4096B***

- **Read requirements in document carefully**

- …

Today

- Lab 3 overview

- Lab 3a/3b tasks

- Bitmap and hash

- Tips

- **Q&A**

Thanks for your listening.