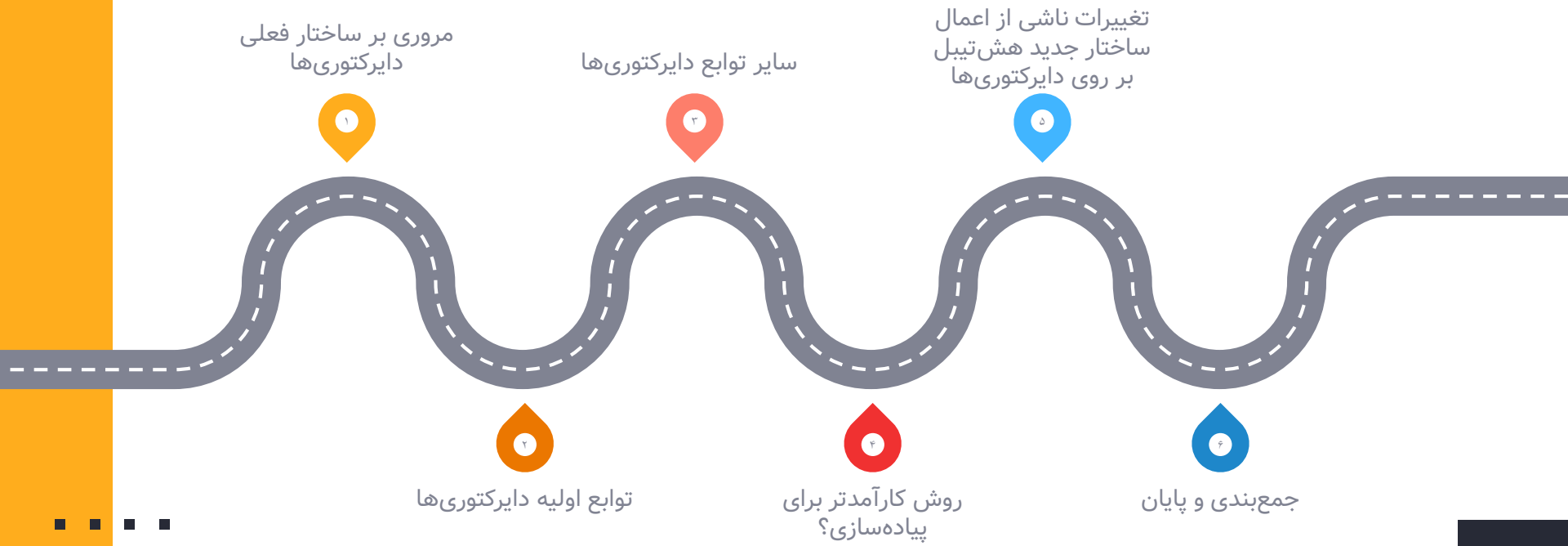
پروژه عملی دوم
پینتوس

اعضای گروه
امیر حلاجی
حامد خادمی خالدی
متین زیودار
محمد خدام

استاد درس
دکتر شهاب‌الدّین نبوی

بهار ۱۴۰۰

مسیر راه


مروری بر ساختار فعلی دایرکتوری‌ها
۱
سایر توابع دایرکتوری‌ها
۳
تغییرات ناشی از اعمال ساختار جدید هش‌تیبل بر روی دایرکتوری‌ها
۵
توابع اولیه دایرکتوری‌ها
۲
روش کارآمدتر برای پیاده‌سازی؟
۴
جمع‌بندی و پایان
۶

۱. مروری بر ساختار فعلی دایرکتوری‌ها

مقدمه

اهداف

- اضافه‌کردن قابلیت سلسله مراتبی فایل سیستم و حالت تو در توی دایرکتوری‌ها
- هر دایرکتوری، یک یا چند entry شامل فایل‌ها و دایرکتوری‌های دیگر درون خودش داشته باشد.

چشم‌انداز

- مفهوم دایرکتوری و دایرکتوری‌اینتری
- بررسی کلی دایرکتوری‌ها و توابع مربوط به آن

ساختار

دایرکتوری (Directory)

```
/* A directory. */
struct dir
  {
    struct inode *inode;
/* Backing store. */
    off_t pos;
/* Current position. */
  };
```

دایرکتوری‌اینتری (Directory Entry)

```
/* A single directory entry. */
struct dir_entry
  {
    block_sector_t inode_sector;
/* Sector number of header. */
    char name[NAME_MAX + 1];
/* Null terminated file name. */
    bool in_use;
/* In use or free? */
  };
```

نمایش بصری

اینتری‌بلاک برای دایرکتوری /

که شامل

.

..

etc

bin

است.

Directory block for /

Entry	Field	Value
0	Inode	1
	Name	“.”
1	Inode	1
	Name	“..”
2	Inode	2
	Name	“etc”
3	Inode	3
	Name	“bin”
4	Inode	0
	Name	0

۲. توابع اولیه دایرکتوری‌ها

dir_open

```
bool
dir_create (block_sector_t sector, size_t entry_cnt)
{
  return inode_create (sector, entry_cnt * sizeof (struct dir_entry), true);
}
```

dir_open

```
struct dir *
dir_open (struct inode *inode)
{
  struct dir *dir = calloc (1, sizeof *dir);
  if (inode != NULL && dir != NULL)
    {
      dir->inode = inode;
      dir->pos = 0;
      return dir;
    }
  else
    {
      inode_close (inode);
      free (dir);
      return NULL;
    }
}
```

dir_open_root

```
struct dir *
dir_open_root (void)
{
  return dir_open (inode_open (ROOT_DIR_SECTOR));
}
```

dir_reopen

```
struct dir *
dir_reopen (struct dir *dir)
{
  return dir_open (inode_reopen (dir->inode));
}
```

dir_close

```
void
dir_close (struct dir *dir)
{
  if (dir != NULL)
    {
      inode_close (dir->inode);
      free (dir);
    }
}
```

dir_get_inode

```
struct inode *
dir_get_inode (struct dir *dir)
{
  return dir->inode;
}
```

lookup 1st part

```
static bool
lookup (const struct dir *dir, const char *name,
        struct dir_entry *ep, off_t *ofsp)
{
  struct dir_entry e;
  size_t ofs;

  ASSERT (dir != NULL);
  ASSERT (name != NULL);
```

lookup 2nd part

```
    for (ofs = 0; inode_read_at (dir->inode, &e, sizeof e, ofs) == sizeof e;
         ofs += sizeof e)
      if (e.in_use && !strcmp (name, e.name))
        {
          if (ep != NULL)
            *ep = e;
          if (ofsp != NULL)
            *ofsp = ofs;
          return true;
        }
    return false;
  }
```

dir-lookup 1st part

```
bool
dir_lookup (const struct dir *dir, const char *name,
            struct inode **inode)
{
  struct dir_entry e;

  ASSERT (dir != NULL);
  ASSERT (name != NULL);
```

dir-lookup 2nd part

```
  inode_lock(dir_get_inode((struct dir *) dir));
  if (lookup (dir, name, &e, NULL))
    *inode = inode_open (e.inode_sector);
  else
    *inode = NULL;
  inode_unlock(dir_get_inode((struct dir *) dir));

  return *inode != NULL;
}
```

۳. سایر توابع دایرکتوری‌ها

dir-add 1st part

```
bool
dir_add (struct dir *dir, const char *name, block_sector_t inode_sector)
{
  struct dir_entry e;
  off_t ofs;
  bool success = false;

  ASSERT (dir != NULL);
  ASSERT (name != NULL);

```

dir-add 2nd part

```
  inode_lock(dir_get_inode(dir));
  /* Check NAME for validity. */
  if (*name == '\0' || strlen (name) > NAME_MAX)
    goto done;

  /* Check that NAME is not in use. */
  if (lookup (dir, name, NULL, NULL))
    goto done;
  if (!inode_add_parent(inode_get_inumber(dir_get_inode(dir)), inode_sector))
    {
      goto done;
    }
```

dir-add 3rd part

```
    for (ofs = 0; inode_read_at (dir->inode, &e, sizeof e, ofs) == sizeof e;
         ofs += sizeof e)
      if (!e.in_use)
        break;

    /* Write slot. */
    e.in_use = true;
    strlcpy (e.name, name, sizeof e.name);
    e.inode_sector = inode_sector;
    success = inode_write_at (dir->inode, &e, sizeof e, ofs) == sizeof e;
```

dir-add 4th part

```
    done:
     inode_unlock(dir_get_inode(dir));
     return success;
    }
```

dir-remove 1st part

```
bool
dir_remove (struct dir *dir, const char *name)
{
  struct dir_entry e;
  struct inode *inode = NULL;
  bool success = false;
  off_t ofs;

  ASSERT (dir != NULL);
  ASSERT (name != NULL);
```

dir-remove 2nd part

```
  inode_lock(dir_get_inode(dir));
  /* Find directory entry. */
  if (!lookup (dir, name, &e, &ofs))
    goto done;

  /* Open inode. */
  inode = inode_open (e.inode_sector);
  if (inode == NULL)
    goto done;
```

dir-remove 3rd part

```
/* Directory to be deleted is used by other processes */
if (inode_is_dir(inode) && inode_get_open_cnt(inode) > 1)
  goto done;

/* Directory to be deleted is nonempty */
if (inode_is_dir(inode) && !dir_is_empty(inode))
  goto done;

/* Erase directory entry. */
e.in_use = false;
if (inode_write_at (dir->inode, &e, sizeof e, ofs) != sizeof e)
  goto done;
```

dir-remove 4th part

```
    /* Remove inode. */
    inode_remove (inode);
    success = true;

 done:
    inode_close (inode);
    inode_unlock(dir_get_inode(dir));
    return success;
}
```

dir_readdir

```
bool
dir_readdir (struct dir *dir, char name[NAME_MAX + 1])
{
  struct dir_entry e;
  inode_lock(dir_get_inode(dir));
  while (inode_read_at (dir->inode, &e, sizeof e, dir->pos) == sizeof e)
    {
      dir->pos += sizeof e;
      if (e.in_use)
        {
          strlcpy (name, e.name, NAME_MAX + 1);
          inode_unlock(dir_get_inode(dir));
          return true;
        }
    }
  inode_unlock(dir_get_inode(dir));
  return false;
}
```

dir_is_empty

```
bool dir_is_empty (struct inode *inode)
{
  struct dir_entry e;
  off_t pos = 0;

  while (inode_read_at (inode, &e, sizeof e, pos) == sizeof e)
    {
      pos += sizeof e;
      if (e.in_use)
        { return false; }
    }
  return true;
}
```

dir_is_root

```
bool dir_is_root (struct dir* dir)
{
  if (!dir)
    {
      return false;
    }
  if (inode_get_inumber(dir_get_inode(dir)) == ROOT_DIR_SECTOR)
    {
      return true;
    }
  return false;
}
```

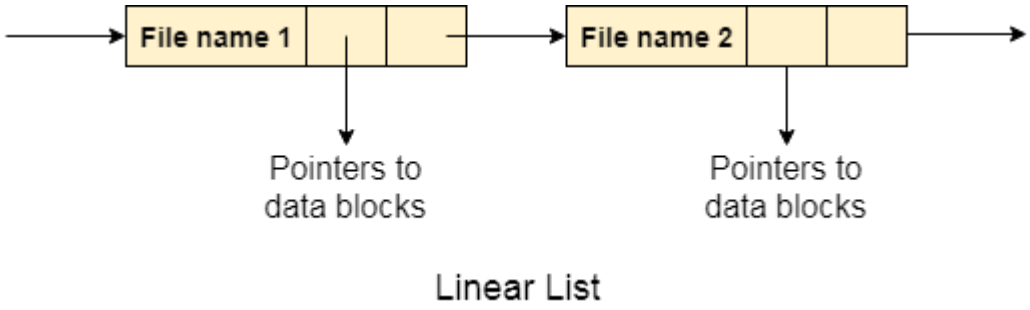
dir_get_parent

```
bool dir_get_parent (struct dir* dir, struct inode **inode)
{

  block_sector_t sector = inode_get_parent(dir_get_inode(dir));
  *inode = inode_open (sector);
  return *inode != NULL;
}
```

پیاده‌سازی
دایرکتوری‌ها

روش Linear List



Linear List

- برای عملیات پایه نظیر

ایجاد

حذف

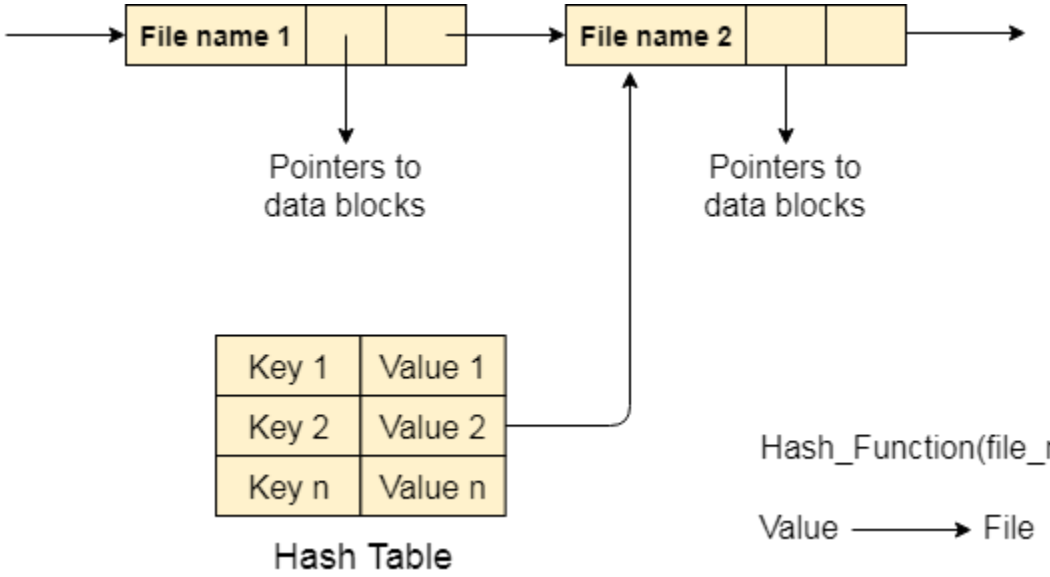
آپدیتِ

دایرکتوری

کل لیست، باید پیمایش شود.

۴.روش کارآمدتر برای پیاده‌سازی؟

روش Hash Table



حل مشکلات

روش پیشین

با استفاده از تکنیک

Hashing

۵. تغییرات ناشی از اعمال ساختار جدید هش‌تیبل بر روی دایرکتوری‌ها

ساختار

BTable

Btable Node Structure

```
struct btNode {
    int isLeaf;     /* is this a leaf node? */
    int numKeys;    /* how many keys does this node contain? */
    int keys[MAX_KEYS];
    struct dir_entries entries[MAX_KEYS];
    off_t offests[MAX_KEYS];
    struct btNode *kids[MAX_KEYS+1];  /* kids[i] holds nodes < keys[i] */
};
```

btCreate

```
bTree
btCreate(void)
{
    bTree b;

    b = malloc(sizeof(*b));
    ASSERT (b);

    b->isLeaf = 1;
    b->numKeys = 0;

    return b;
}
```

btDestroy

```
void
btDestroy(bTree b)
{
    int i;

    if(!b->isLeaf) {
        for(i = 0; i < b->numKeys + 1; i++) {
            btDestroy(b->kids[i]);
        }
    }

    free(b);
}
```

searchKey 1st part

```
static int
searchKey(int n, const int *a, int key)
{
    int lo;
    int hi;
    int mid;

    /* invariant: a[lo] < key <= a[hi] */
    lo = -1;
    hi = n;

```

searchKey 2nd part

```
        while(lo + 1 < hi) {
            mid = (lo+hi)/2;
            if(a[mid] == key) {
                return mid;
            } else if(a[mid] < key) {
                lo = mid;
            } else {
                hi = mid;
            }
        }
        return hi;
    }
```

btSearch 1st part

```
int
btSearch(bTree b, int key,
         struct dir_entry *ep, off_t *ofsp)
{
    int pos;

    /* have to check for empty tree */
    if(b->numKeys == 0) {
        return 0;
    }

```

btSearch 2nd part

```
        /* look for smallest position that key fits below */
        pos = searchKey(b->numKeys, b->keys, key);

        if(pos < b->numKeys && b->keys[pos] == key) {
            ep = b->entries[pos];
            ofsp = b->offests[pos];
            return 1;
        } else {
            return(!b->isLeaf && btSearch(b->kids[pos], key, &ep, &ofsp));
        }
    }
```

btInsertInternal 1st part

```
static bTree
btInsertInternal(bTree b, int key, int *median,
            struct dir_entry *ep, off_t *ofsp)
{
    int pos;
    int mid;
    bTree b2;
    pos = searchKey(b->numKeys, b->keys, key);
    if(pos < b->numKeys && b->keys[pos] == key) {
        /* nothing to do */
        return 0;
    }
```

btInsertInternal 2nd part

```
    if(b->isLeaf) {

            /* everybody above pos moves up one space */
            memmove(&b->keys[pos+1], &b->keys[pos], sizeof(*(b->keys)) * (b->numKeys -
     pos));
            b->keys[pos] = key;
            b->entries[pos] = ep;
            b->offests[pos] = ofsp;
            b->numKeys++;

        }
```

btInsertInternal 3rd part

```
    else {

            /* insert in child */
            b2 = btInsertInternal(b->kids[pos], key, &mid, &ep, &ofsp);

            /* maybe insert a new key in b */
            if(b2) {

                /* every key above pos moves up one space */
                memmove(&b->keys[pos+1], &b->keys[pos], sizeof(*(b->keys)) * (b->numKeys -
     pos));
                memmove(b->entries[pos+1], &b->entries[pos], sizeof(*(b->entries)) * b-
    >numKeys - pos);
```

btInsertInternal 4th part

```
            memmove(b->offests[pos+1], &b->offests[pos], sizeof(*(b->offests)) * b->numKeys - pos);
            /* new kid goes in pos + 1*/
            memmove(&b->kids[pos+2], &b->kids[pos+1], sizeof(*(b->keys)) * (b->numKeys - pos));
            memmove(b->entries[pos+2], &b->entries[pos+1], sizeof(*(b->entries)) * b->numKeys - pos);
            memmove(b->offests[pos+2], &b->offests[pos+1], sizeof(*(b->offests)) * b->numKeys - pos);

            b->keys[pos] = mid;
            b->kids[pos+1] = b2;
            b->numKeys++;
        }
```

btInsertInternal 5th part

```
/* we waste a tiny bit of space by splitting now
 * instead of on next insert */
if(b->numKeys >= MAX_KEYS) {
    mid = b->numKeys/2;

    *median = b->keys[mid];
    b2 = malloc(sizeof(*b2));
    b2->numKeys = b->numKeys - mid - 1;
    b2->isLeaf = b->isLeaf;
```

```
/* make a new node for keys > median */
/* picture is:
 *
 *      3 5 7
 *      A B C D
 *
 * becomes
 *          (5)
 *      3       7
 *      A B     C D
 */
```

btInsertInternal 6th part

```
memmove(b2->keys, &b->keys[mid+1], sizeof(*(b->keys)) * b2->numKeys);
        memmove(b2->entries, &b->entries[mid+1], sizeof(*(b->entries)) * b2->numKeys);
        memmove(b2->offests, &b->offests[mid+1], sizeof(*(b->offests)) * b2->numKeys);
        if(!b->isLeaf) {
            memmove(b2->kids, &b->kids[mid+1], sizeof(*(b->kids)) * (b2->numKeys + 1));
            memmove(b2->entries, &b->entries[mid+1], sizeof(*(b->entries)) * b2->numKeys + 1);
            memmove(b2->offests, &b->offests[mid+1], sizeof(*(b->offests)) * b2->numKeys + 1);
        }
        b->numKeys = mid;
        return b2;
    } else {return 0;}}
```

btInsert 1st part

```
void
btInsert(bTree b, int key,
        struct dir_entry *ep, off_t *ofsp)
{
    bTree b1;   /* new left child */
    bTree b2;   /* new right child */
    int median;

    b2 = btInsertInternal(b, key, &median, &ep, &ofsp);

```

btInsert 2nd part

```
    if(b2) {
        /* basic issue here is that we are at the root */
        /* so if we split, we have to make a new root */
        b1 = malloc(sizeof(*b1));
        ASSERT (b1);
        /* copy root to b1 */
        memmove(b1, b, sizeof(*b));
        /* make root point to b1 and b2 */
        b->numKeys = 1;
        b->isLeaf = 0;
        b->keys[0] = median;
        b->kids[0] = b1;
        b->kids[1] = b2; }}
```

ساختار
HTable

Htable Node Structure

```
struct htNode {
    struct bTree b;
    struct hash h;
};
```

htCreate

```
struct hTree
htCreate(void)
{
    hTree ht;

    ht->b = btCreate();
    hash_init(ht->h, dir_hash, dir_less, NULL);

    return ht;
}
```

htDestroy

```
void
htDestroy(hTree ht)
{
    btDestroy(ht->b);
    hash_clear(ht->h);
}
```

htSearch

```
struct bool
htSearch(const struct hTree ht, const char *name,
        struct dir_entry *ep, off_t *ofsp)
{
    int key;

    e = hash_find (ht->h, &name);
    if (e == NULL)
        return NULL
    key = hash_entry (e, struct dir_entry, char);
    btSearch(ht->b, key, &ep, &ofsp);
}
```

htInsert

```
void
htInsert(const struct hTree ht, const char *name, block_sector_t inode_sector,
        struct dir_entry *ep, off_t *ofsp)
{
    struct dir_entry e;
    int key;
    if (hash_insert (ht->h, &name) == NULL)
        return;
    btInsert(ht->b, key, &ep, &ofsp)        // panic if sth goes wrong
    e.in_use = true;
    strlcpy (e.name, name, sizeof e.name);
    e.inode_sector = inode_sector;
}
```

تغییرات

directory.c

تغییرات directory.c

```
struct dir
  {
    struct inode *inode;              /* Backing store. */
    // off_t pos;                        /* Current position. */
    struct htree htree;               /* Hash balanced tree. */
  };
```

تغییرات directory.c

```
struct dir *
dir_open (struct inode *inode)
{
  struct dir *dir = calloc (1, sizeof *dir);
  if (inode != NULL && dir != NULL)
    {
      dir->inode = inode;
      // dir->pos = 0;
      dir->htree = htCreate();
      return dir;
    }
  else { . . . }
```

تغییرات 1/3 directory.c

```
static bool
lookup (const struct dir *dir, const char *name,
        struct dir_entry *ep, off_t *ofsp)
{
  struct dir_entry e;
  size_t ofs;

  ASSERT (dir != NULL);
  ASSERT (name != NULL);

  . . .
```

تغییرات directory.c 2/3

```
// for (ofs = 0; inode_read_at (dir->inode, &e, sizeof e, ofs) == sizeof e;
  //      ofs += sizeof e)
  //   if (e.in_use && !strcmp (name, e.name))
  //     {
  //       if (ep != NULL)
  //         *ep = e;
  //       if (ofsp != NULL)
  //         *ofsp = ofs;
  //       return true;
  //     }
```

تغییرات directory.c 3/3

```
  htSearch(dir->htree, name, &e, &ofs)
  if (e != NULL){
    if (ep != NULL)
      *ep = e;
    if (ofsp != NULL)
      *ofsp = ofs;
    return true;
  }
  return false;
}
```

تغییرات directory.c

```
bool
dir_add (struct dir *dir, const char *name, block_sector_t inode_sector)
{ . . .
// for (ofs = 0; inode_read_at (dir->inode, &e, sizeof e, ofs) == sizeof e;
  //      ofs += sizeof e)
  //   if (!e.in_use)
  //     break;
  /* Write slot. */
  // e.in_use = true;
  // strlcpy (e.name, name, sizeof e.name);
  // e.inode_sector = inode_sector;
  if (!htInsert(dir->htree, name, inode_sector, &e, &ofs))
    goto done; . . . }
```

تغییرات directory.c

```
bool
dir_remove (struct dir *dir, const char *name)
{     . . .

  /* Remove inode. */
  htDestroy(dir->htree);      // Destroy hTree
  inode_remove (inode);
  success = true;

. . .             }
```

افزودنی به directory.c

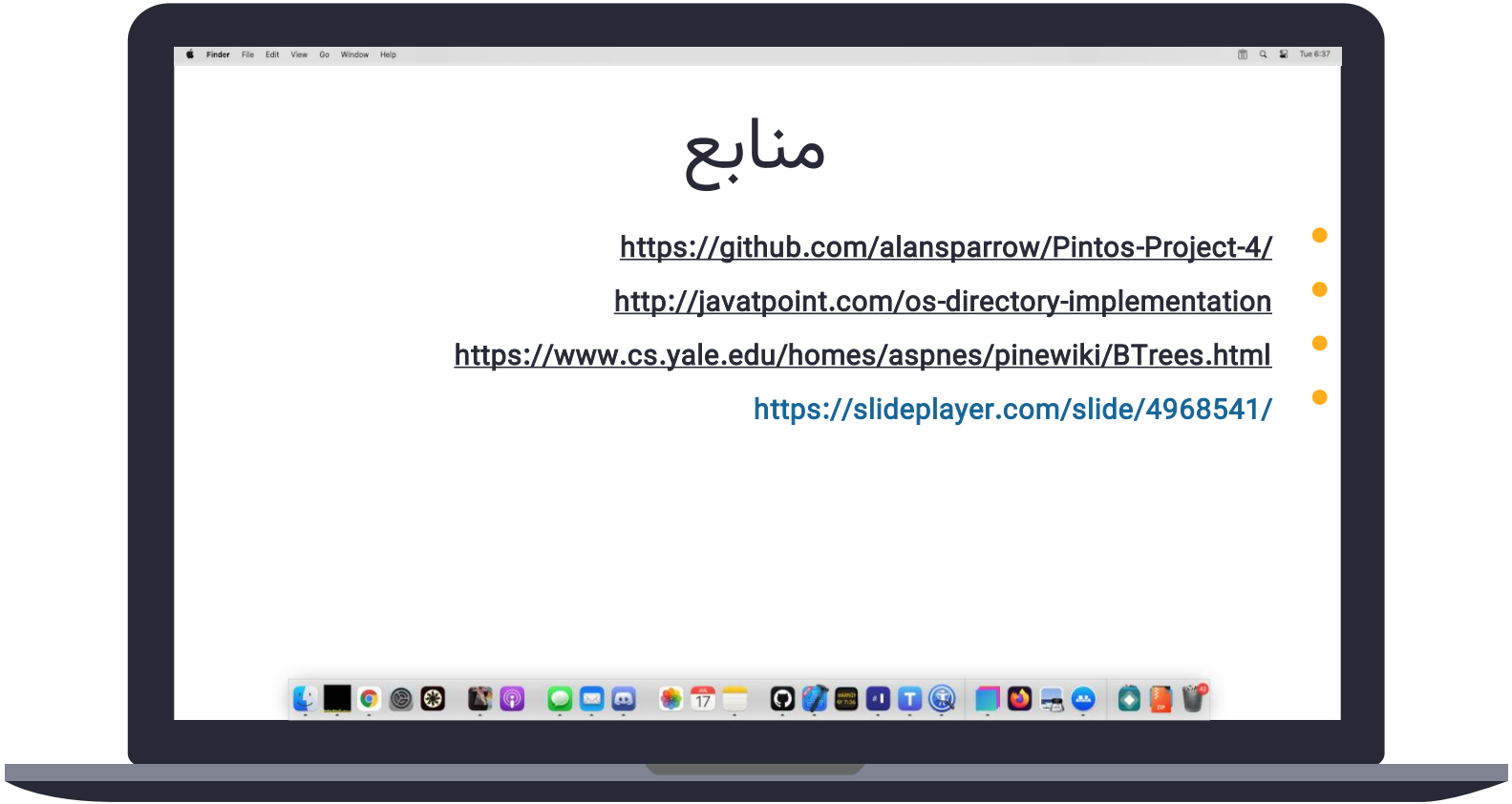
```c
/* Returns a hash value for the dir that E refers to. */
unsigned
dir_hash (const struct hash_elem *e, void *aux UNUSED)
{
  const struct dir *d = hash_entry (e, struct dir, hash_elem);
  return ((uintptr_t) d->addr) >> NAME_MAX;
}
```

افزودنی به directory.c

```
/* Returns true if dir A precedes dir B. */
bool
dir_less (const struct hash_elem *a_, const struct hash_elem *b_,
          void *aux UNUSED)
{
  const struct dir *a = hash_entry (a_, struct dir, hash_elem);
  const struct dir *b = hash_entry (b_, struct dir, hash_elem);

  return a->addr < b->addr;
}
```

۶. جمع‌بندی


منابع
https://github.com/alansparrow/Pintos-Project-4/
http://javatpoint.com/os-directory-implementation
https://www.cs.yale.edu/homes/aspnes/pinewiki/BTrees.html
https://slideplayer.com/slide/4968541/

ممنون از توجه شما!