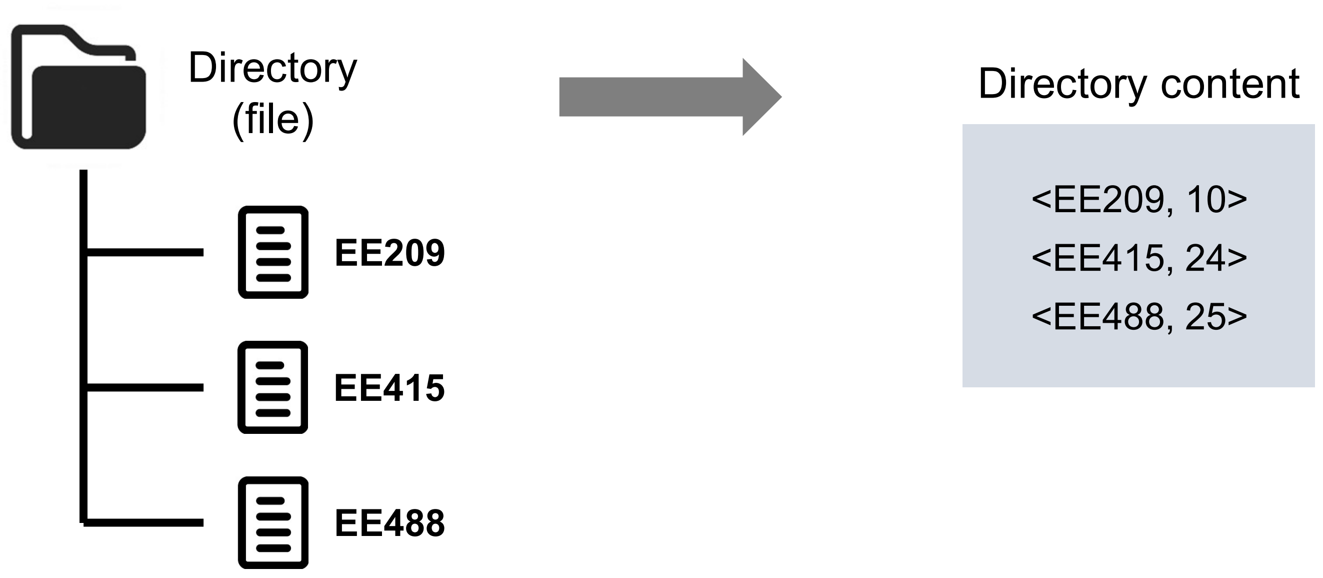
Directory

Youjip Won

KAIST

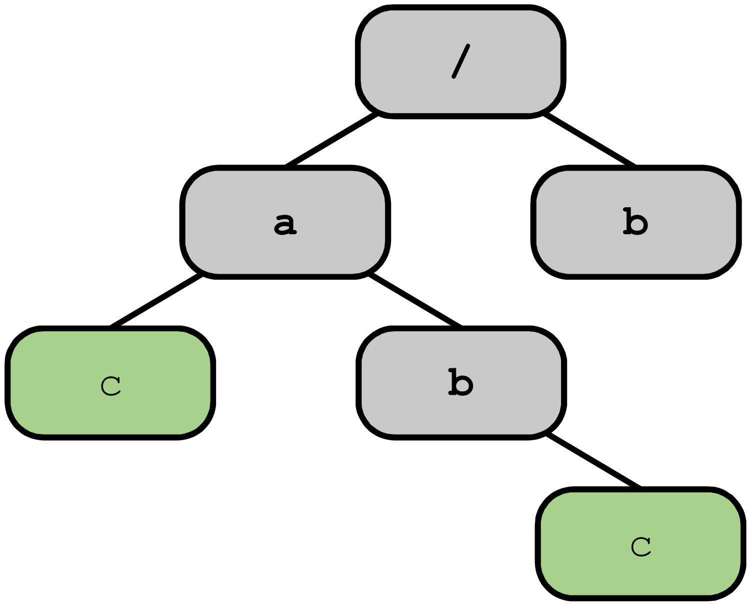
Directory

- The file that its data is a list of directory entries.
- Directory entry is <user-readable filename, inode number> pair.



Hierarchical path name

- The directories form a tree, starting at a special directory called the root.
- In xv6, all files and directories appear under the root directory "/".
  - The slash "/" represents the root directory or is used to separate the name of a file or a directory in a path name.

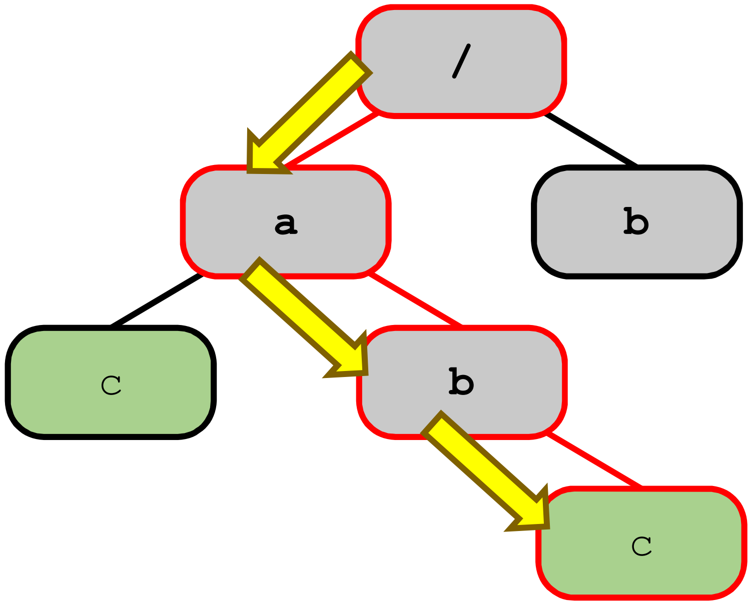


Examples of directory

/
/b
/a/c
/a/b/c

Path lookup

- A path "`/a/b/c`" refers to the file or directory named `c` inside the directory "`b`" inside the directory "`a`" in the root directory.
- xv6 uses recursive lookup to find the file or directory for a given path.



Directory Inode

- The inode that represents a directory has type `T_DIR`.

```c
// in-memory copy of an inode
struct inode {
  uint dev;              // Device number
  uint inum;             // Inode number
  int ref;               // Reference count
  struct sleeplock lock; // protects everything below here
  int valid;             // inode has been read from disk?

  short type;            // copy of disk inode
  short major;
  short minor;
  short nlink;
  uint size;
  uint addrs[NDIRECT+1];
};
```

```c
#define T_DIR  1   // Directory
#define T_FILE 2   // File
#define T_DEV  3   // Device
```

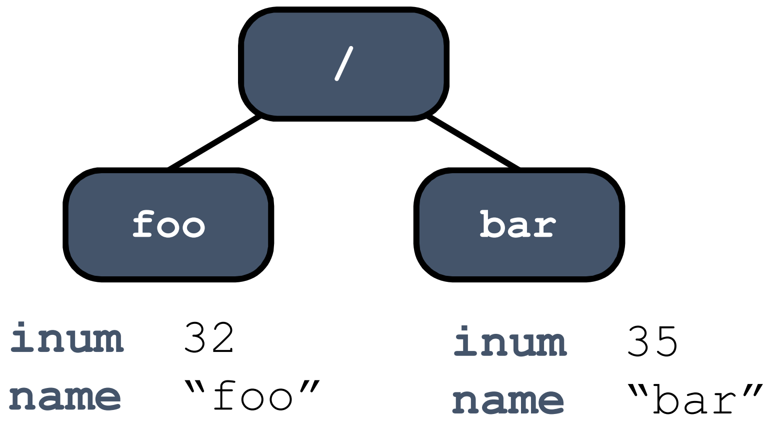
`struct dirent`: Directory Entry

- Data structure `dirent` represent a directory entry.
- Each directory entry contains the inode number and the file name.
  - `dirent` wirh zero inode number is ***free***.
  - Maximum length of the file name is `DIRSIZ`.
  - If the length of file name is less than `DIRSIZ`, it is terminated by a NUL (0) byte.

```
struct dirent {
  ushort inum;
  char name[DIRSIZ];
};
```

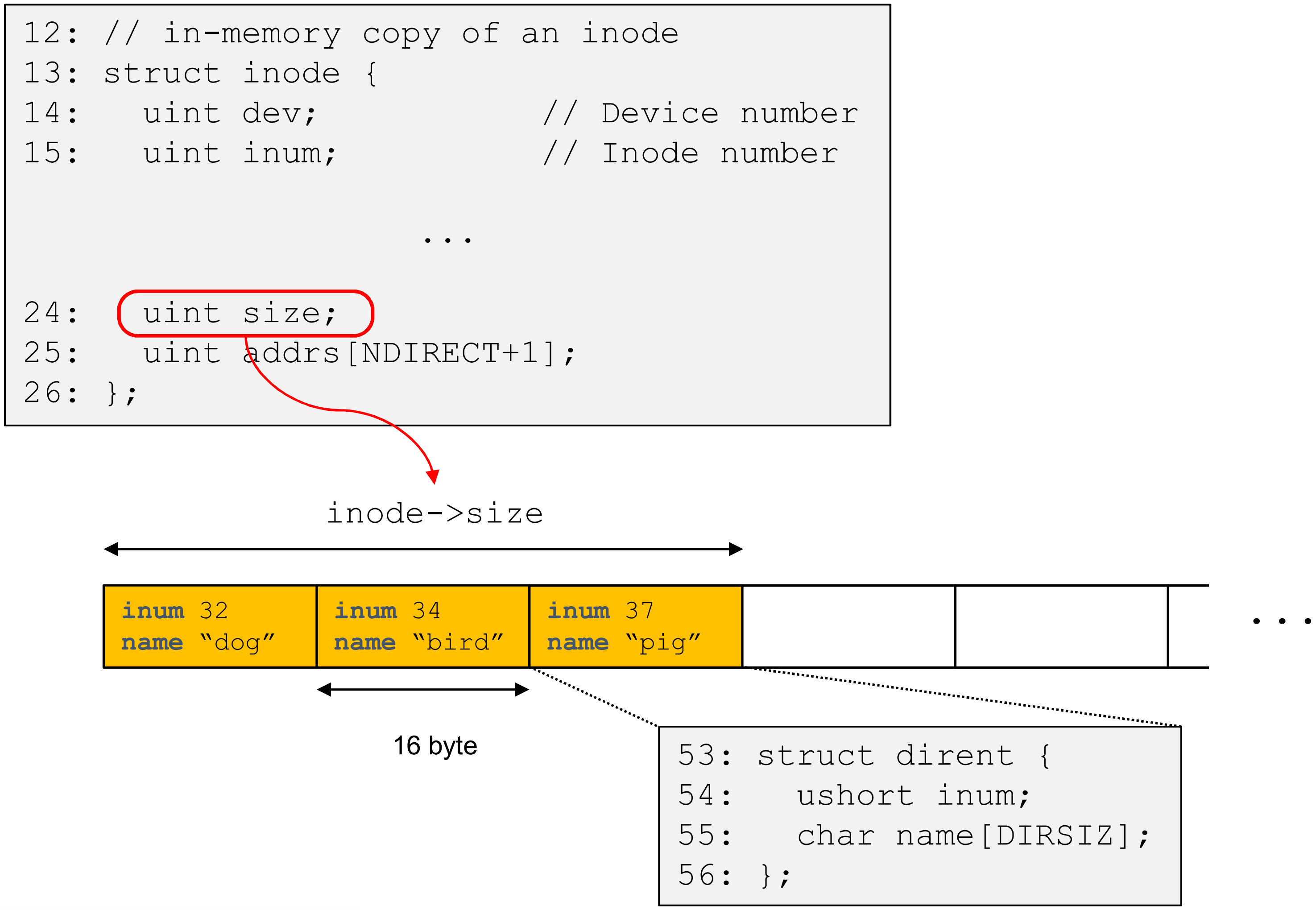
14

16 Byte structure



`struct dirent`: Directory Entry (Cont'd)

- Directory entries are stored in the file block as an array.



```
12: // in-memory copy of an inode
13: struct inode {
14:   uint dev;            // Device number
15:   uint inum;           // Inode number

                     ...

24:   uint size;
25:   uint addrs[NDIRECT+1];
26: };
```

```
53: struct dirent {
54:   ushort inum;
55:   char name[DIRSIZ];
56: };
```

Find or insert an entry in the directory

- `struct inode *dirlookup(inode *dp, char *name,`
  `uint *poff)`
  - Find a file or directory named `name` under the directory that pointed by `dp`.
  - If there is target inode, it returns the pointer of target inode.
  - `poff` is set to the offset of the matched entry in the directory.

- `int dirlink(struct inode *dp, char *name, uint inum)`
  - Add the new directory entry to the directory that pointed by `dp`.
  - The directory entry is a pair of `name` and `inum`.
  - Return 0 on success, -1 on failure.

`dirlookup(inode *dp, char *name, uint *poff)`

① Check the inode parameter `dp` if it is `T_DIR` typed.

② Read an entry and store into the local variable `dp`.

③ If inode number is zero, it is considered as an empty directory entry.

④ Compare the string `de.name` whether it matches the argument `name`.

⑤ Return the pointer of an inode if found by calling `iget()`.

dirlookup()

- Search a directory for an entry with the given name `name`.

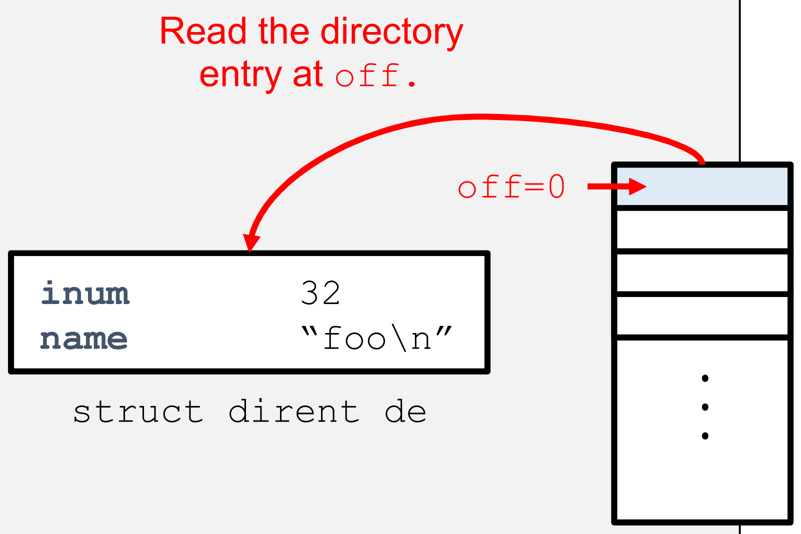
  ① Check the inode parameter `dp` if it is `T_DIR` typed.

```
struct inode*
dirlookup(struct inode *dp, char *name, uint *poff)
{
  uint off, inum;
  struct dirent de;

  if(dp->type != T_DIR)
    panic("dirlookup not DIR");

                                 ...
```

dirlookup() (Cont'd)

- Search a directory for an entry with given name `name`.
  - ② Read an entry and store into the local variable `de`.

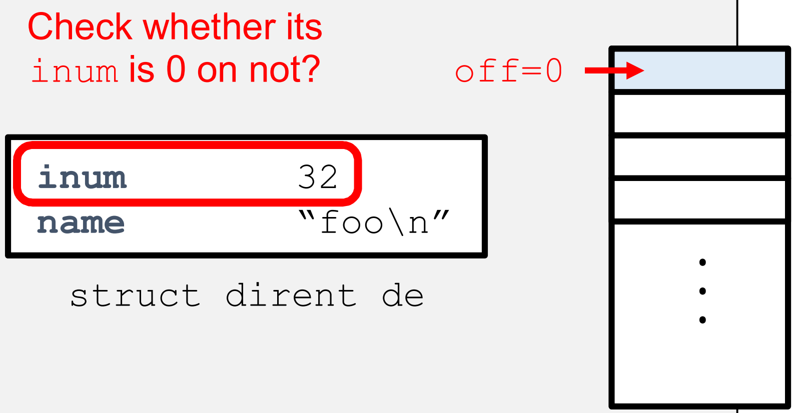
```
                          ...

  for(off = 0; off < dp->size; off += sizeof(de)){
    if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
      panic("dirlookup read");
    if(de.inum == 0)
      continue;
    if(namecmp(name, de.name) == 0){
      // entry matches path element
      if(poff)
        *poff = off;
      inum = de.inum;
      return iget(dp->dev, inum);
    }
  }

  return 0;
}
```



`dirlookup()` (Cont'd)

- Search a directory for an entry with given name `name`.
  - ③ If inode number is zero, it is considered as an empty directory entry.

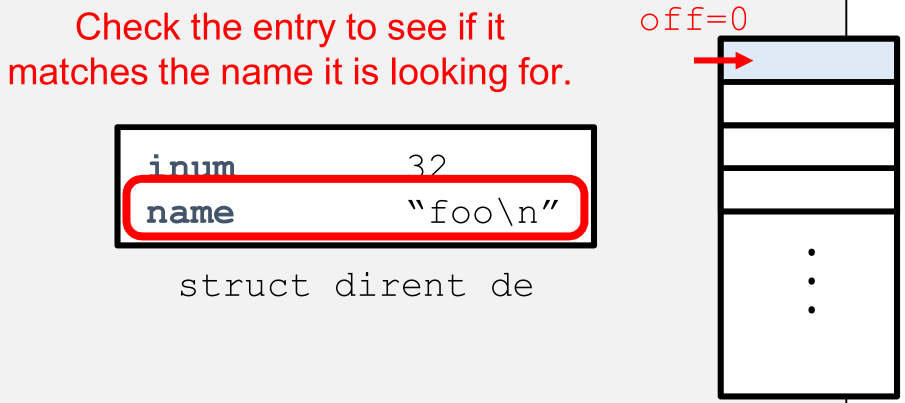
```
                                  ...

  for(off = 0; off < dp->size; off += sizeof(de)){
    if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
      panic("dirlookup read");
    if(de.inum == 0)
      continue;
    if(namecmp(name, de.name) == 0){
      // entry matches path element
      if(poff)
        *poff = off;
      inum = de.inum;
      return iget(dp->dev, inum);
    }
  }

  return 0;
}
```



dirlookup() (Cont'd)

- Search a directory for an entry with given name `name`.
  - ④ Compare the string `de.name` whether it matches the argument `name`.

```
                                   ...

  for(off = 0; off < dp->size; off += sizeof(de)){
    if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
      panic("dirlookup read");
    if(de.inum == 0)
      continue;
    if(namecmp(name, de.name) == 0){
      // entry matches path element
      if(poff)
        *poff = off;
      inum = de.inum;
      return iget(dp->dev, inum);
    }
  }

  return 0;
}
```


Check the entry to see if it matches the name it is looking for.
off=0
inum 32
name "foo\n"
struct dirent de

dirlookup() (Cont'd)

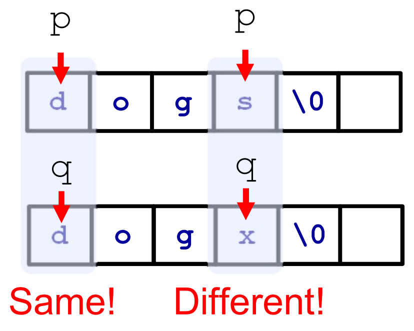
- Search a directory for an entry with given name `name`.
  - ④ Compare the string `de.name` whether it matches the argument `name`.

`strncmp()` : Compare the given string character by character.

```
int
namecmp(const char *s, const char *t)
{
  return strncmp(s, t, DIRSIZ);
}
```

```
int
strncmp(const char *p, const char *q, uint n)
{
  while(n > 0 && *p && *p == *q)
    n--, p++, q++;
  if(n == 0)
    return 0;
  return (uchar)*p - (uchar)*q;
}
```

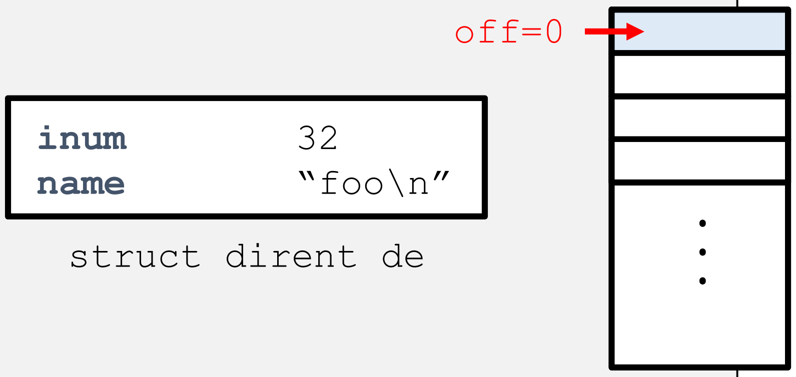
If characters are same, move to the next character



dirlookup() (Cont'd)

- Search a directory for an entry with given name `name`.
  - ⑤ Return the pointer of an inode if found by calling `iget()`.

```
                              ...

  for(off = 0; off < dp->size; off += sizeof(de)){
    if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
      panic("dirlookup read");
    if(de.inum == 0)
      continue;
    if(namecmp(name, de.name) == 0){
      // entry matches path element
      if(poff)
        *poff = off;
      inum = de.inum;
      return iget(dp->dev, inum);
    }
  }

  return 0;
}
```

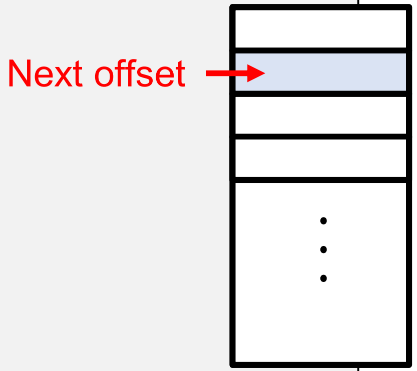


`dirlookup()` (Cont'd)

- Search a directory for an entry with given name `name`.

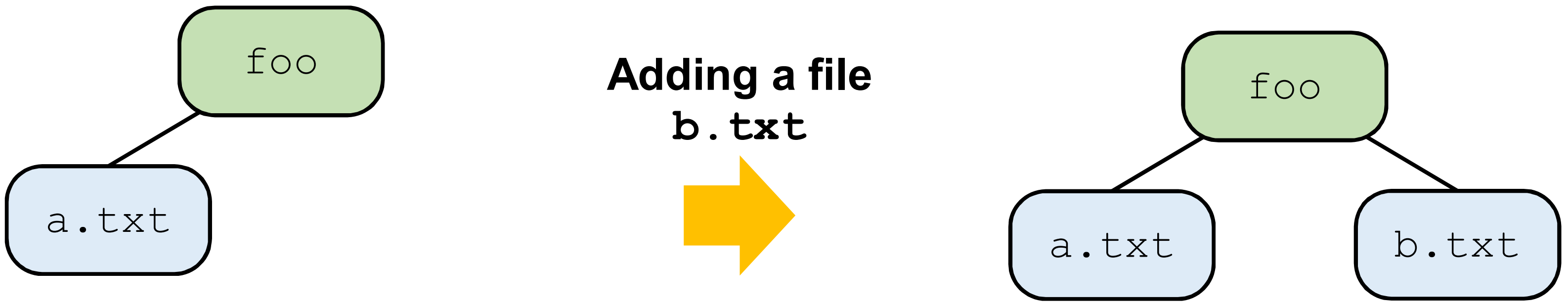
  ⑤ - 2. Otherwise, move to the next entry and repeat ①~⑤.

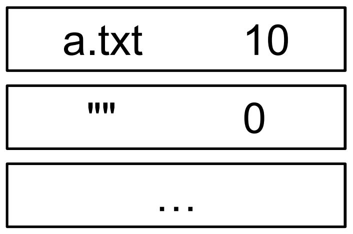
```
                              ...

  for(off = 0; off < dp->size; off += sizeof(de)){
    if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
      panic("dirlookup read");
    if(de.inum == 0)
      continue;
    if(namecmp(name, de.name) == 0){
      // entry matches path element
      if(poff)
        *poff = off;
      inum = de.inum;
      return iget(dp->dev, inum);
    }
  }

  return 0;
}
```



`dirlink(inode *dp, char *name, uint inum)`

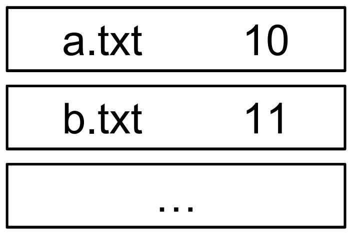
- Add the new directory entry with the given `name` and inode number `inum`.
- If the name already exists, `dirlink()` returns an error (-1).





a.txt	10
""	0
…	

Data of directory `foo`



a.txt	10
b.txt	11
…	

Data of directory `foo`

dirlink()

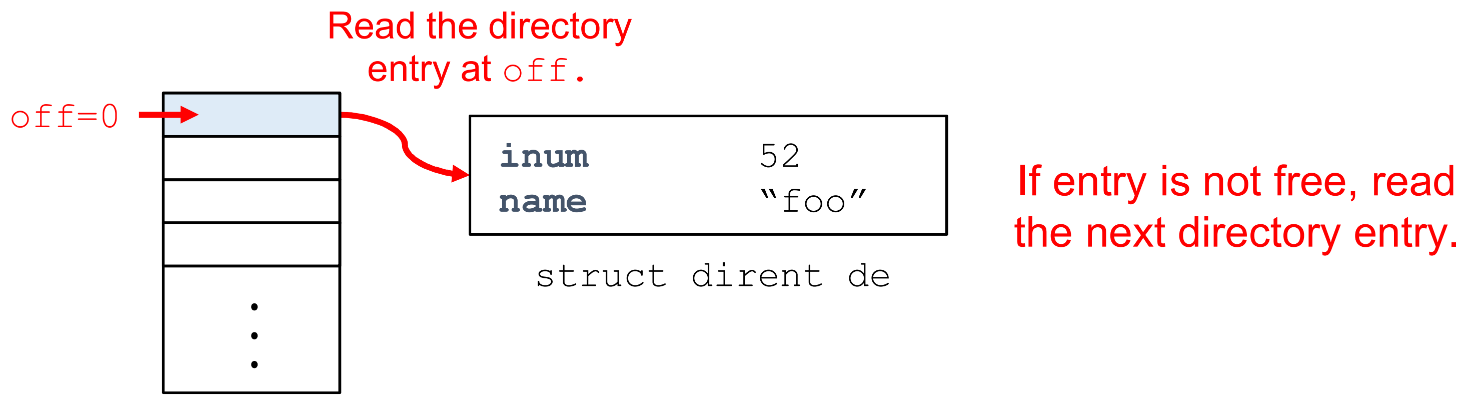
- Call `dirlookup()` to check any directory with the same name exists.
- `dirlookup()` returns zero if a directory entry with name `name` is not found.

```
int dirlink(struct inode *dp, char *name, uint inum){
  int off;
  struct dirent de;
  struct inode *ip;

  // Check that name is not present.
  if((ip = dirlookup(dp, name, 0)) != 0){
    iput(ip);
    return -1;
  }

  … // Removed for saving space.

  return 0;
}
```

`dirlink()` (Cont'd)

① Search for an empty directory entry. It is considered empty if inode number is zero.

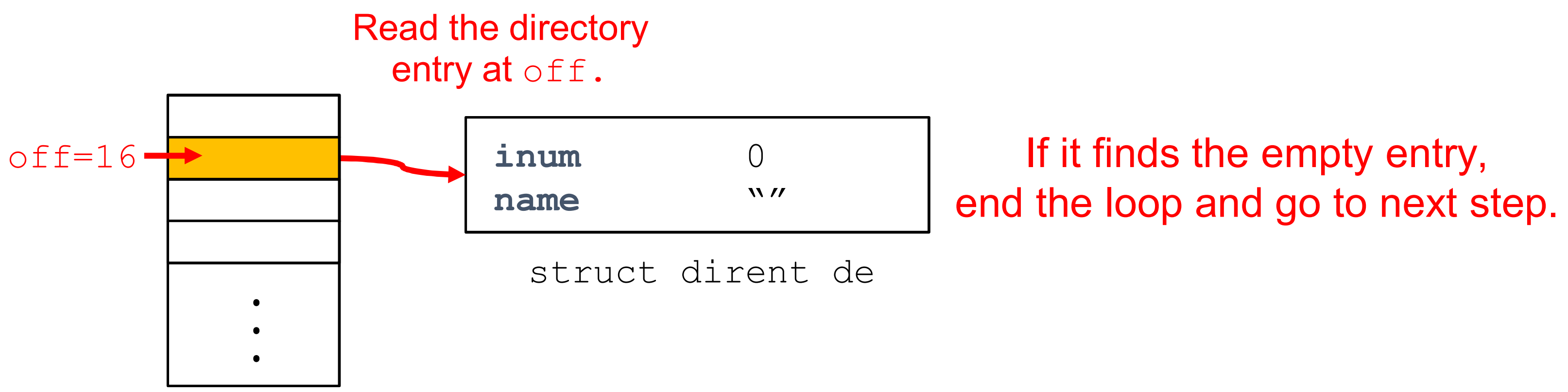
```
int dirlink(struct inode *dp, char *name, uint inum){
  … // Removed for saving space.
  // Look for an empty dirent.
  for(off = 0; off < dp->size; off += sizeof(de)){
    if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
      panic("dirlink read");
    if(de.inum == 0)
      break;
  }
  … // Removed for saving space.
}
```


Read the directory entry at off.
off=0
inum 52
name "foo"
struct dirent de
If entry is not free, read the next directory entry.

`dirlink()` (Cont'd)

① Search for an empty directory entry. It is considered empty if inode number is zero.

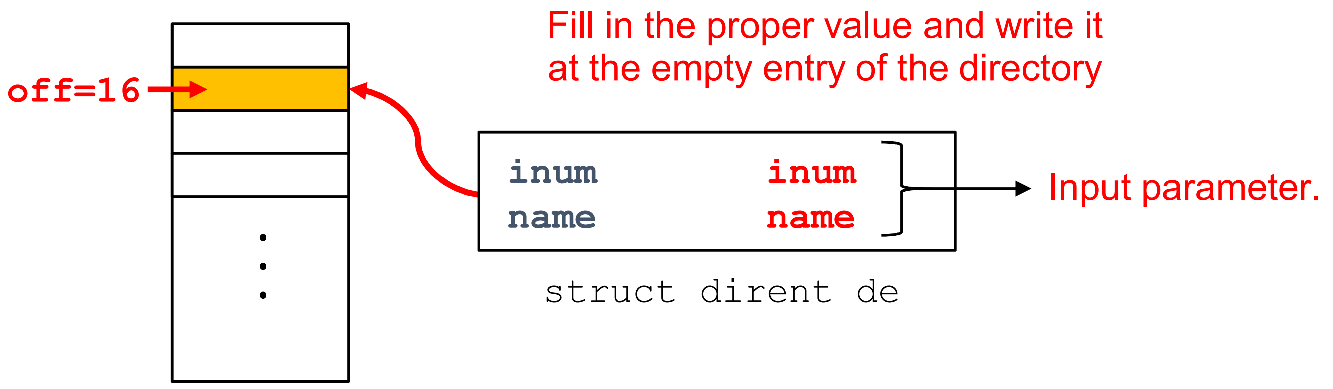
```
int dirlink(struct inode *dp, char *name, uint inum){
    … // Removed for saving space.
    // Look for an empty dirent.
    for(off = 0; off < dp->size; off += sizeof(de)){
      if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
        panic("dirlink read");
      if(de.inum == 0)
        break;
    }
    … // Removed for saving space.
}
```


Read the directory entry at off.
off=16
inum 0
name ""
struct dirent de
If it finds the empty entry, end the loop and go to next step.

`dirlink()` (Cont'd)

② If found an empty entry, write the new entry to the this by calling `writei()`.

```
553 int dirlink(struct inode *dp, char *name, uint inum){
      … // Removed for saving space.
571
572   strncpy(de.name, name, DIRSIZ);
573   de.inum = inum;
574   if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
575     panic("dirlink");
576
577   return 0;
578 }
```



off=16
Fill in the proper value and write it at the empty entry of the directory
inum inum
name name
Input parameter.
struct dirent de

Pathname lookup

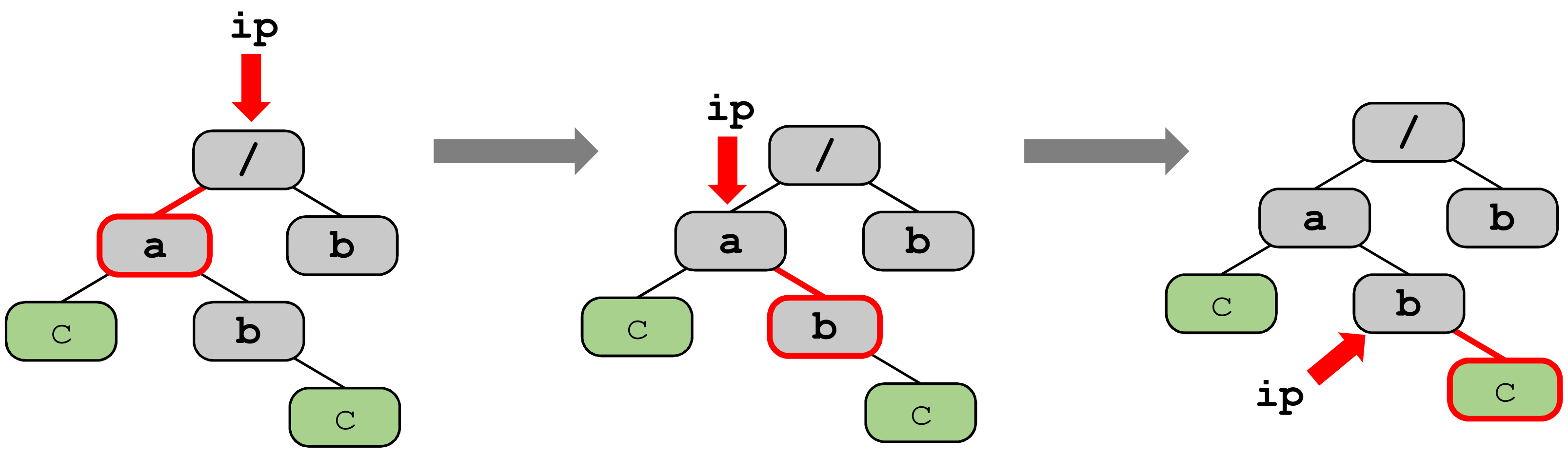
- Path: sequence of directories that ends with the filename or directory

  `/a/b/c`

- Path name lookup involves a succession of `dirlookup()` calls, one for each directory name.
- The lookup, that calls to `dirlookup()`, would start at (1) root directory or (2) process's current directory.

```
dirlookup(struct inode *ip, char *name, uint *poff)
```

① **`dirlookup(ip,"a",)`** ② **`dirlookup(ip,"b",)`** ③ **`dirlookup(ip,"c",)`**



Pathname lookup (Cont.)

- If the path begins with a slash, evaluation begins at the root; otherwise, the current directory.
  - The current directory is the per-process attribute.
  - The system call `chdir()` change the current directory.
- Path element or component
  - For the case of path "/a/b/c", there are three elements; a, b, and c.

```
struct proc {
    … // Removed for saving space.
    struct file *ofile[NOFILE];
    struct inode *cwd;
    char name[16];
};
```

namei() and nameiparent()

- namei()
  - evaluates path and returns the corresponding inode of the last element.
  - calls namex() with 0 nameiparent parameter.
- nameiparent(): evaluates path and returns the inode of the parent of the last element. It copies the last element to name.
  - calls namex() with 1 nameiparent parameter.

```
struct inode*
namei(char *path)
{
  char name[DIRSIZ];
  return namex(path, 0, name);
}

struct inode*
nameiparent(char *path, char *name)
{
  return namex(path, 1, name);
}
```

`namex()`: path lookup function

- `struct inode *namex(char *path, int nameiparent, char *name)`
- If `nameiparent` is 0,
  - Return the inode pointer for `name` if it is found.
- If not,
  - Copy the final component in the `path` to the `name`.
  - Return the inode pointer of the parent directory for a file `name`.
  - It is usually used when the caller should modify the directory content of a file, such as `link()` or `unlink()`.

`char *skipelem(char *path, char *name)`

- `char *skipelem(char *path, char *name)`
  - Copy the first component of the `path` into `name`.
  - Return the pointer to the element following the copied one.
  - Examples:
    - `skipelem("a/bb/c", name);`

      name is set to “a” and return “bb/c”.
    - `skipelem(“a”, name);`

      name is set to “a” and return “”.
    - `skipelem(“”, name);`

      name is set to “” and return NULL.

`namex()`: Get the start inode pointer.

- If the path begins with a slash, lookup begins at the root directory.
- Otherwise, it begins at the current directory.
- The inode pointer assigned to variable `ip`.

```
static struct inode* namex(char *path, int nameiparent, char *name) {
  struct inode *ip, *next;

  if(*path == '/')
    ip = iget(ROOTDEV, ROOTINO);
  else
    ip = idup(myproc()->cwd);

  … // Removed for saving space.
  return ip;
}
```

`namex()`: Loop for each element in the path.

- `char *skipelem(char *path, char *name)`
  - Copy the first path element from path into `name`.
  - Return a pointer to the element following the copied one.

```
static struct inode* namex(char *path, int nameiparent, char *name){
    … // Removed for saving space.

  while((path = skipelem(path, name)) != 0){
    ilock(ip);

      We explain the detailed implementation, later.

    iunlockput(ip);
    ip = next;
  }
    … // Removed for saving space.
  return ip;
}
```

`namex()`: Loop for each element in the path. (Cont.)

- `namex (“a/b”, 1, …);`
  - 1st loop: `path = skipelem(“a/b”, name);` ➔ `name = “a”, path = “b”;`
  - 2nd loop: `path = skipelem(“b”, name);` ➔ `name = “b”, path = “”;`
  - 3rd loop: `path = skipelem(“”, name);` ➔ `name = “”, path = NULL;` ➔ Stop

```
static struct inode* namex(char *path, int nameiparent, char *name){
    … // Removed for saving space.

    while((path = skipelem(path, name)) != 0){
      ilock(ip);

      iunlockput(ip);
      ip = next;
    }
     … // Removed for saving space.
    return ip;
}
```

Loop for each element in a path!

We explain the detailed implementation, later.

`namex()`: Check whether the `ip` is directory or not.

- For each loop (each element), there are three things to do.
- xv6 finds the element named `name` in the directory `ip` at the third step.
- Before doing the third step, xv6 checks whether the `ip` is directory or not.

```
static struct inode* namex(char *path, int nameiparent, char *name){
  … // Removed for saving space.

  while((path = skipelem(path, name)) != 0){
    ilock(ip);
      ① Check whether the ip is directory or not.
      ② If nameiparent is not 0, stop the lookup one step earlier
      ③ By calling dirlookup(), find the inode named name in directory ip.
    iunlockput(ip);
    ip = next;
  }
   … // Removed for saving space.
}
```

`namex()`: Check whether the `ip` is directory or not.

- Before checking it, xv6 acquire the lock for inode `ip`.
- The type of `ip` should be `T_DIR`. Otherwise, release the lock and return the NULL.

```
static struct inode* namex(char *path, int nameiparent, char *name){
   … // Removed for saving space.

  while((path = skipelem(path, name)) != 0){
    ilock(ip);
    if(ip->type != T_DIR){
      iunlockput(ip);
      return 0;
    }

    … // Removed for saving space.
    iunlockput(ip);
    ip = next;
  }
   … // Removed for saving space.
}
```

`namex()`: `nameiparent` is not 0

- If `nameiparent` is not 0,
  - Return **the inode pointer of the parent directory** for the last component in the path.

```
static struct inode* namex(char *path, int nameiparent, char *name){
    … // Removed for saving space.

  while((path = skipelem(path, name)) != 0){
    ilock(ip);
```

① Check whether the `ip` is directory or not.

② If `nameiparent` is not 0, stop the lookup one step earlier.

③ By calling `dirlookup()`, find the inode named `name` in directory `ip`.

```
    iunlockput(ip);
    ip = next;
  }
    … // Removed for saving space.
}
```

`namex()`: `nameiparent` is not 0

- If the first character of `path` is ‘\0’, there is no more component in `path`.
- Since the next `skipelem()` call will return `NULL`, the loop stops at the next step.
- So it returns current `ip`, which is the parent directory of the last component in the path.

```
static struct inode* namex(char *path, int nameiparent, char *name){
   … // Removed for saving space.

  while((path = skipelem(path, name)) != 0){
    ilock(ip);
     … // Removed for saving space.
    if(nameiparent && *path == '\0'){
      iunlock(ip);
      return ip;
    }
     … // Removed for saving space.
    iunlockput(ip);
    ip = next;
  }
   … // Removed for saving space.
}
```

namex(): find the inode named name in directory ip.

- namex() calls the dirlookup(ip, name, 0) for finding the inode for name.

```
static struct inode* namex(char *path, int nameiparent, char *name){
    … // Removed for saving space.

  while((path = skipelem(path, name)) != 0){
    ilock(ip);
```

① Check whether the ip is directory or not.

② If nameiparent is not 0, stop the lookup one step earlier.

③ By calling dirlookup(), find the inode named name in directory ip.

```
    iunlockput(ip);
    ip = next;
  }
    … // Removed for saving space.
}
```

Example: `namex("a/b", 1, …)`

- 1st loop: `name = "a", path = "b";` ➔ `next = dirlookup(cwd, "a", 0)`
  - ➔ `ip = next = 0xdeadbeef` // inode pointer of "a".
- 2nd loop: `name = "b", path = "";` ➔ `next = dirloopup(0xdeadbeef, "b", 0)`
  - ➔ `ip = next = 0x8badf00d` // inode pointer of "a/b".

```
static struct inode* namex(char *path, int nameiparent, char *name){
  … // Removed for saving space.

  while((path = skipelem(path, name)) != 0){
    ilock(ip);
    … // Removed for saving space.
    if((next = dirlookup(ip, name, 0)) == 0){
      iunlockput(ip);
      return 0;
    }
    iunlockput(ip);
    ip = next;
  }
  … // Removed for saving space.
}
```

namex(“”,1,…)

- If input parameter `path` of `namex()` is not empty string, `namex()` calls `return` within the loop.
- Otherwise, it does not go into the loop and return NULL.

```
static struct inode* namex(char *path, int nameiparent, char *name) {
  … // Removed for saving space.

  while((path = skipelem(path, name)) != 0){
    … // Removed for saving space.
  }
  if(nameiparent){
    iput(ip);
    return 0;
  }
  return ip;
}
```

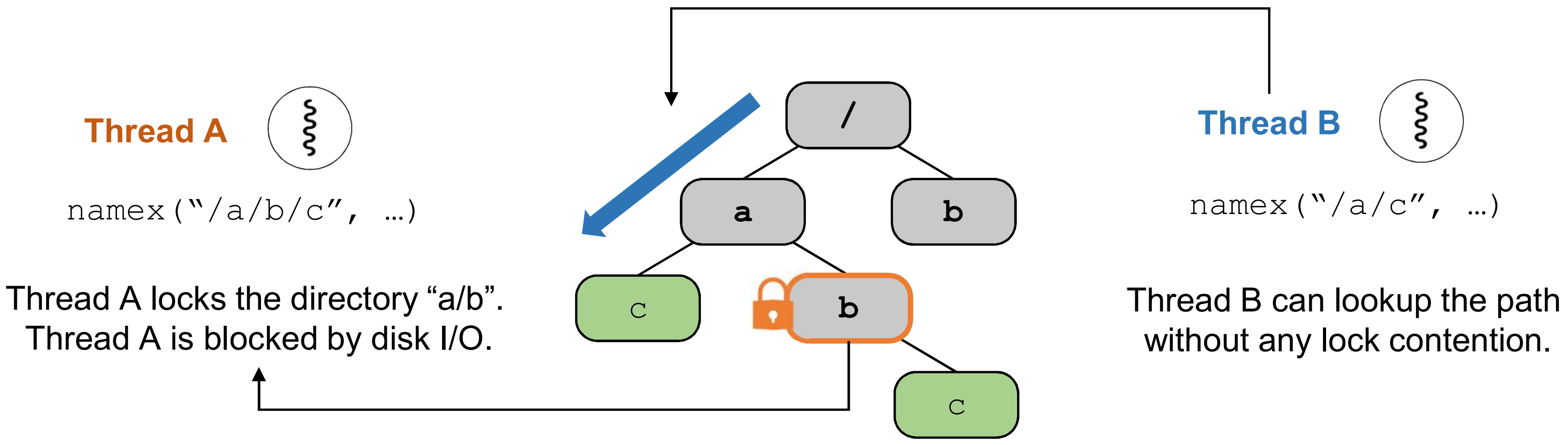
`namex()`: Acquire and release per-inode lock.

- Each iteration of the loop begins by locking `ip` and find the inode named `name` in `ip`.
- Then, release the lock of `ip` before the end of the iteration.
- `namex()` locks each directory in the path separately.
  - ➔ Lookups in different directories can proceed in parallel.

```
static struct inode* namex(char *path, int nameiparent, char *name){
  … // Removed for saving space.

  while((path = skipelem(path, name)) != 0){
    ilock(ip);

      By calling dirlookup(), find the inode named name in directory ip.

    iunlockput(ip);
    ip = next;
  }
  … // Removed for saving space.
  return ip;
}
```

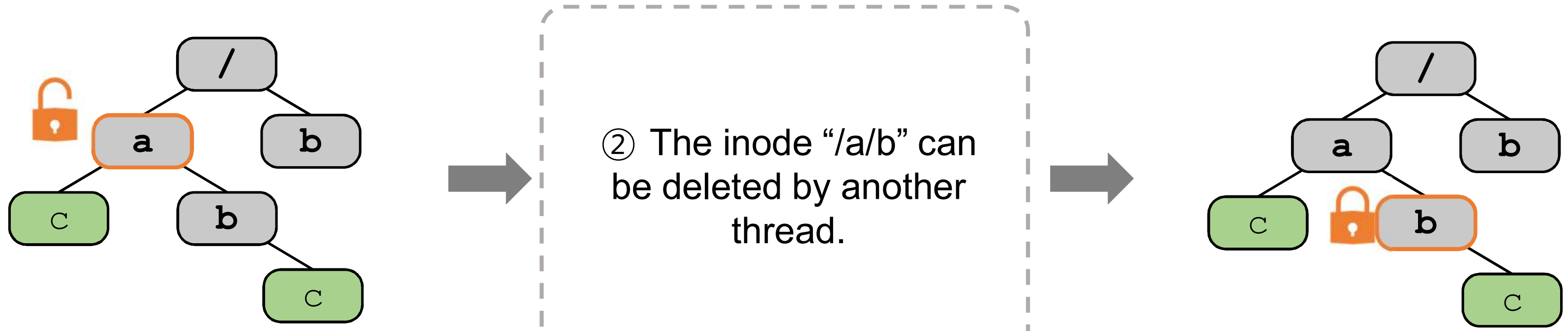
Concurrency of `namex()`

- The procedure `namex()` may take a long time to complete.
  - It could involve several disk operations.
  - `ilock()` could read on-disk inodes to load the inode structure in memory.
  - `dirlookup()` could read file blocks of directories to traverse its entries.
- `namex()` locks each directory in the path separately.
- If a thread invokes `namex()`, another thread looking up a different pathname can proceed concurrently.


Thread A
namex("/a/b/c", …)
Thread A locks the directory "a/b".
Thread A is blocked by disk I/O.
/
a
b
c
b
c
Thread B
namex("/a/c", …)
Thread B can lookup the path
without any lock contention.

Risk of concurrency: race condition

- In `namex()`, each iteration only locks a single inode.
  - `dirlookup()` returns the pointer of next inode.
  - The returned inode pointer is locked after releasing the lock of parent directory.
- There can be following situation in xv6.



① Release the lock on 'a'.

② The inode "/a/b" can be deleted by another thread.

③ Acquire the lock on 'b'.

Is the inode pointer returned from `dirlookup()` still valid?
Can xv6 invoke `ilock()` for this inode pointer?

Risk of concurrency: race condition (Cont.)

- `dirlookup()` returns an inode pointer that was obtained using `iget()`.
  - `iget()` increases the reference count of the inode.
- In xv6, **if reference count is larger than 0,** the inode is not deleted from inode cache and from the file system. (`iput()`)
- By separating the `iget()` and `ilock()`, xv6 avoids the race condition.

```
void iput(struct inode *ip){
  acquiresleep(&ip->lock);
  if(ip->valid && ip->nlink == 0){
    acquire(&icache.lock);
    int r = ip->ref;
    release(&icache.lock);
    if(r == 1){
        Remove the in-memory inode as well as on-disk inode.
    }
  }
  releasesleep(&ip->lock);
                                    …
}
```

If this process is the last reference, xv6 removes this inode.

Risk of concurrency: deadlock

- What happen if locking the next inode before releasing the lock on the parent directory?
  - It may result in a deadlock.
  - If `namex("/./a",…)` is invoked, the next inode "/." is same with parent directory "/" in the first iteration.
  - In this case, the thread may try to acquire the lock that already held.

Summary

- Directory layer
  - `dirlookup()` and `dirlink()`
- Path lookup
  - `namex()`, `namei()`, and `nameiparent()`
  - Concurrency of `namex()`