Operating System Concepts

Lecture 30: Directory Structure

Omid Ardakanian
oardakan@ualberta.ca
University of Alberta

MWF 12:00-12:50 VVC 2 215

File operation: `open(file_name, mode)`

- search the directory for the named file (file name lookup)
- check access mode (create, read-only, read-write, append-only, etc.) against file permissions
- if the requested access is permitted, copy the entry associated with the file to the system-wide open-file table (if it isn't opened already by another process) and increment the open count

File operation: `open(file_name, mode)`

- search the directory for the named file (file name lookup)
- check access mode (create, read-only, read-write, append-only, etc.) against file permissions
- if the requested access is permitted, copy the entry associated with the file to the system-wide open-file table (if it isn't opened already by another process) and increment the open count
- add a pointer to the open-file table entry to the process's file descriptor table (kept in its PCB)

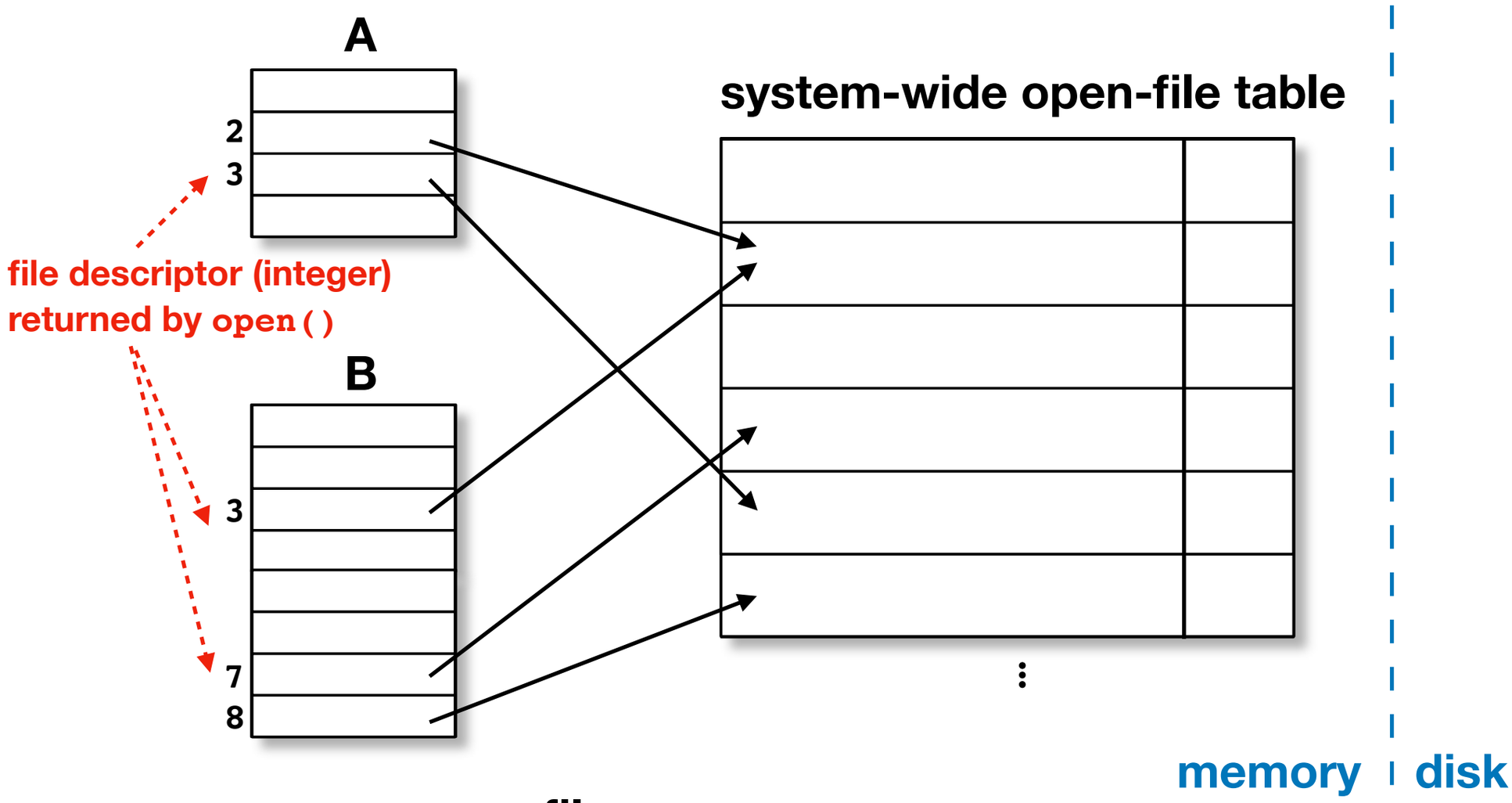
File operation: `open(file_name, mode)`

- search the directory for the named file (file name lookup)
- check access mode (create, read-only, read-write, append-only, etc.) against file permissions
- if the requested access is permitted, copy the entry associated with the file to the system-wide open-file table (if it isn't opened already by another process) and increment the open count
- add a pointer to the open-file table entry to the process's file descriptor table (kept in its PCB)
- initialize the current-file-position pointer to the start of the file

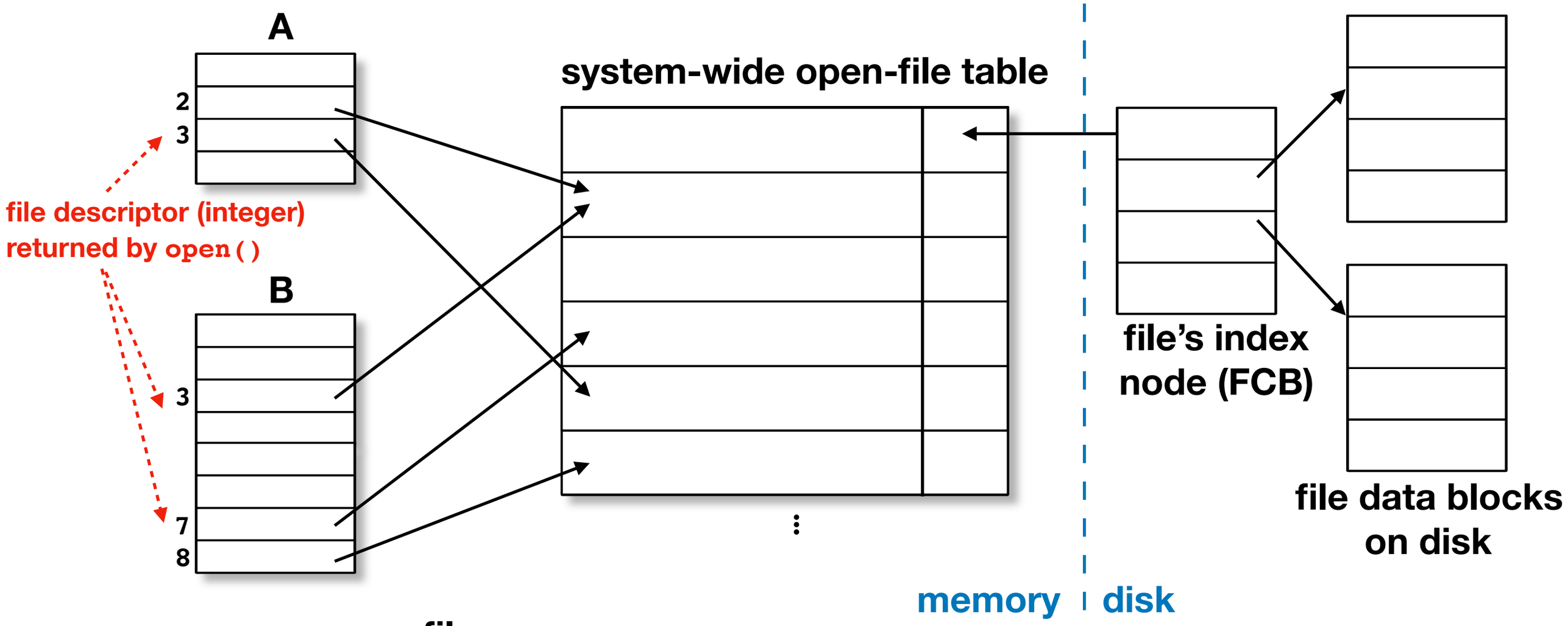
File operation: `open(file_name, mode)`

- search the directory for the named file (file name lookup)
- check access mode (create, read-only, read-write, append-only, etc.) against file permissions
- if the requested access is permitted, copy the entry associated with the file to the system-wide open-file table (if it isn't opened already by another process) and increment the open count
- add a pointer to the open-file table entry to the process's file descriptor table (kept in its PCB)
- initialize the current-file-position pointer to the start of the file
- return a pointer to the file entry in the process file descriptor table (referred to as **file descriptor**)

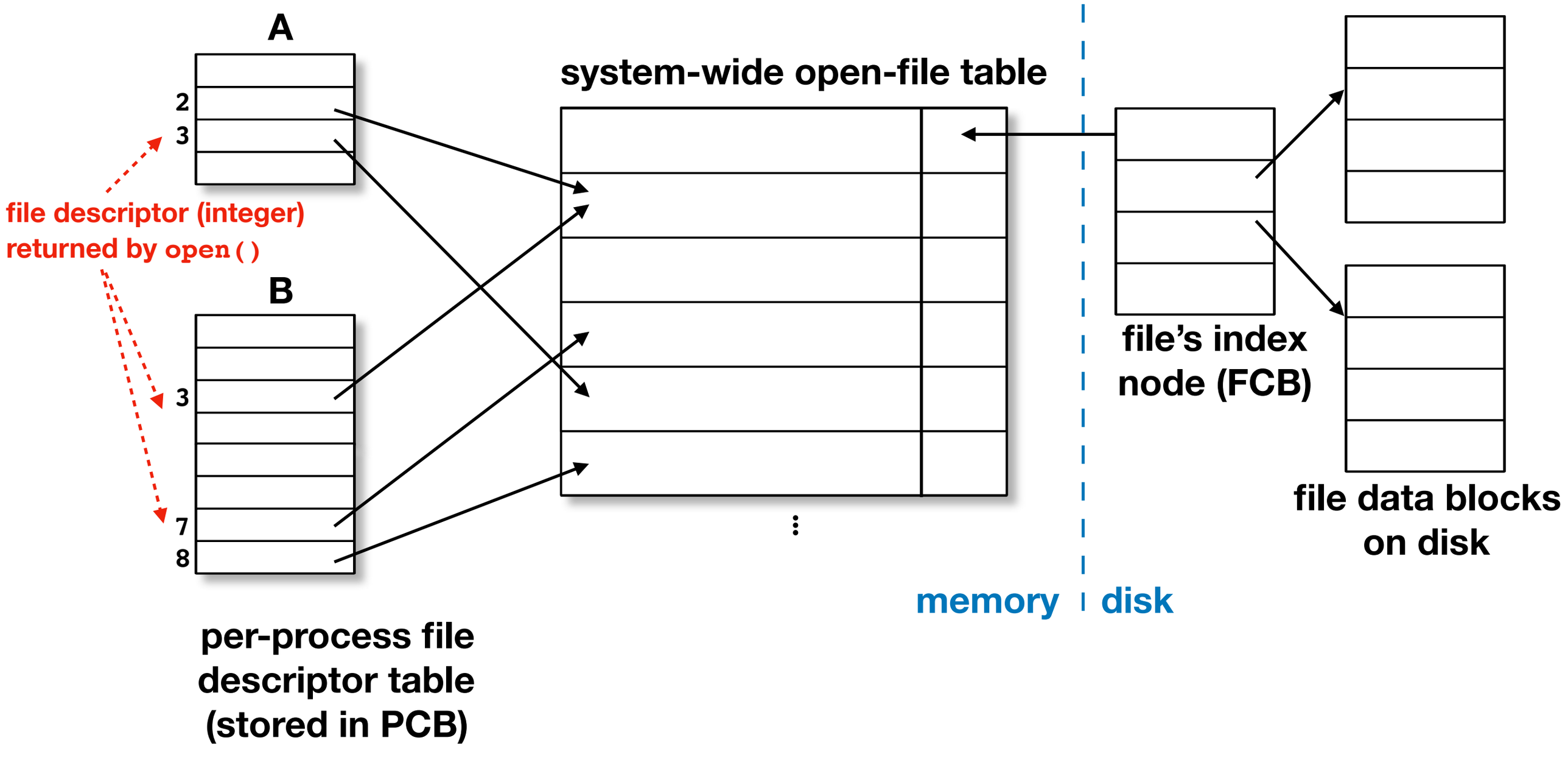
Basic idea


A
2
3
B
3
7
8
system-wide open-file table
file descriptor (integer)
returned by open()
per-process file
descriptor table
(stored in PCB)
memory
disk

Basic idea


A
2
3
B
3
7
8
file descriptor (integer)
returned by open()
per-process file
descriptor table
(stored in PCB)
system-wide open-file table
file's index
node (FCB)
file data blocks
on disk
memory
disk

Basic idea



when a file operation is requested, it is described via an index into the file descriptor table; OS quickly gets to the corresponding entry in the system-wide open-file table

File operation: `close(file_descriptor)`

- locate and remove the file's entry from the process file table
- decrement the open count in the system-wide open-file table
- if the open count reaches zero, remove the entry from the system-wide open-file table
  - otherwise the kernel may run out space

Modified operations: using file descriptor

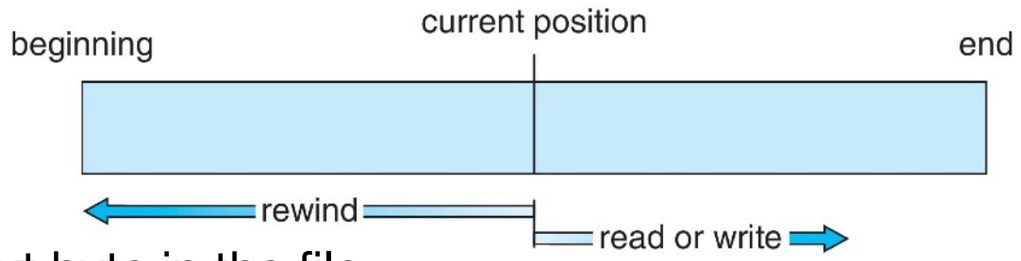
by calling `open` before other operations, they can be executed faster via the file descriptor, eliminating the need to perform file name lookup for each operation

- so the lookup cost is amortized over many operations

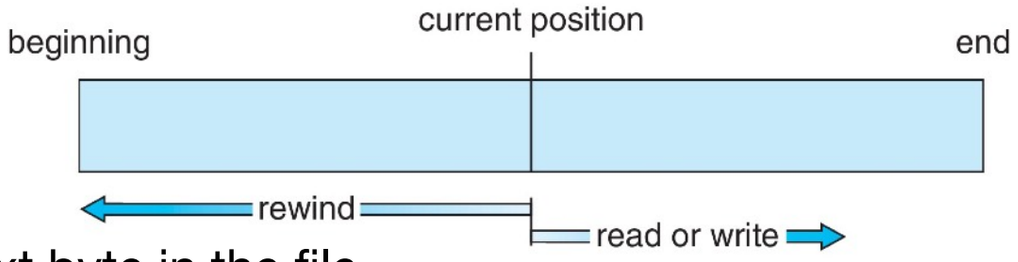
```
fd = Open(“filename”, mode)

Close(fd)

Truncate(fd)

Seek(fd, offset)

Read(fd, buffer, length)

Write(fd, buffer, length)
```

File access methods

- sequential: information in the file is processed/accessed in order
  - is based on the tape model: keep a pointer to the next byte in the file
  - update the pointer on each read/write/seek operation
  - commonly used in text editors, compilers

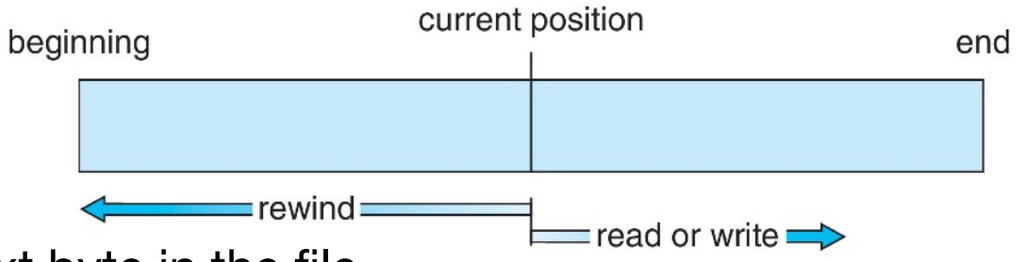


File access methods



- sequential: information in the file is processed/accessed in order
  - is based on the tape model: keep a pointer to the next byte in the file
  - update the pointer on each read/write/seek operation
  - commonly used in text editors, compilers
- direct access: address any block in the file rapidly given its offset within the file
  - is based on the disk model: disks allow random access to any block
  - the block number to access is relative to the beginning of the file
    - so the programmer doesn't need to know where the file is stored on disk
  - it is possible to read block 8 then 12 and then 5
  - databases use this method of access; they compute which block contains the answer (e.g., by hashing the key provided in the query) and read that block directly

File access methods



- sequential: information in the file is processed/accessed in order
  - is based on the tape model: keep a pointer to the next byte in the file
  - update the pointer on each read/write/seek operation
  - commonly used in text editors, compilers
- direct access: address any block in the file rapidly given its offset within the file
  - is based on the disk model: disks allow random access to any block
  - the block number to access is relative to the beginning of the file
    - so the programmer doesn't need to know where the file is stored on disk
  - it is possible to read block 8 then 12 and then 5
  - databases use this method of access; they compute which block contains the answer (e.g., by hashing the key provided in the query) and read that block directly
- some systems only support one access method, some support both methods, and some support only the method declared when the file was created

Simulating sequential access on a direct-access file

sequential access	implementation for direct access
reset	cp = 0;
read_next	read cp; cp = cp + 1;
write_next	write cp; cp = cp + 1;

Simulating sequential access on a direct-access file

sequential access	implementation for direct access
reset	cp = 0;
read_next	read cp; cp = cp + 1;
write_next	write cp; cp = cp + 1;

- simulating direct access on a sequential-access file is extremely inefficient (**why?**)

Today's class

- On-disk and in-memory data structures
- Directory
  - structures
  - implementation
- Naming strategies
- Sharing and protection

On-disk file system data structures

- boot control block contains information needed to boot OS from that volume
  - is typically the first block of a volume
  - may not exist if the volume does not contain an OS

On-disk file system data structures

- boot control block contains information needed to boot OS from that volume
  - is typically the first block of a volume
  - may not exist if the volume does not contain an OS
- per-file file-control block (FCB) contains information about the file including a unique identifier which allows association with a directory entry, ownership, permissions, file size, pointers to file data blocks, etc.
  - FCB is called index node (**inode**) in UNIX File System (UFS)

On-disk file system data structures

- boot control block contains information needed to boot OS from that volume
  - is typically the first block of a volume
  - may not exist if the volume does not contain an OS
- per-file file-control block (FCB) contains information about the file including a unique identifier which allows association with a directory entry, ownership, permissions, file size, pointers to file data blocks, etc.
  - FCB is called index node (**inode**) in UNIX File System (UFS)
- volume control block contains information about the volume, such as block count, block size, free-block count, free-block pointers, free-FCB count, FCB pointers, etc.
  - volume control block is called **superblock** in UFS and master file table in NTFS

On-disk file system data structures

- boot control block contains information needed to boot OS from that volume
  - is typically the first block of a volume
  - may not exist if the volume does not contain an OS
- per-file file-control block (FCB) contains information about the file including a unique identifier which allows association with a directory entry, ownership, permissions, file size, pointers to file data blocks, etc.
  - FCB is called index node (**inode**) in UNIX File System (UFS)
- volume control block contains information about the volume, such as block count, block size, free-block count, free-block pointers, free-FCB count, FCB pointers, etc.
  - volume control block is called **superblock** in UFS and master file table in NTFS
- directory structure keeps the mapping between file names and inode numbers

In-memory file system data structures

the following data structures are loaded in memory at mount time for performance improvement

- **mount table** contains a pointer to the volume control block (i.e., the superblock) of each mounted file system, the file system type, etc.

In-memory file system data structures

the following data structures are loaded in memory at mount time for performance improvement

- **mount table** contains a pointer to the volume control block (i.e., the superblock) of each mounted file system, the file system type, etc.
- cached directory structure holds directory information for recently accessed directories

In-memory file system data structures

the following data structures are loaded in memory at mount time for performance improvement

- **mount table** contains a pointer to the volume control block (i.e., the superblock) of each mounted file system, the file system type, etc.
- cached directory structure holds directory information for recently accessed directories
- system-wide **open-file table** contains a copy of the FCB of each open file as well as other information

In-memory file system data structures

the following data structures are loaded in memory at mount time for performance improvement

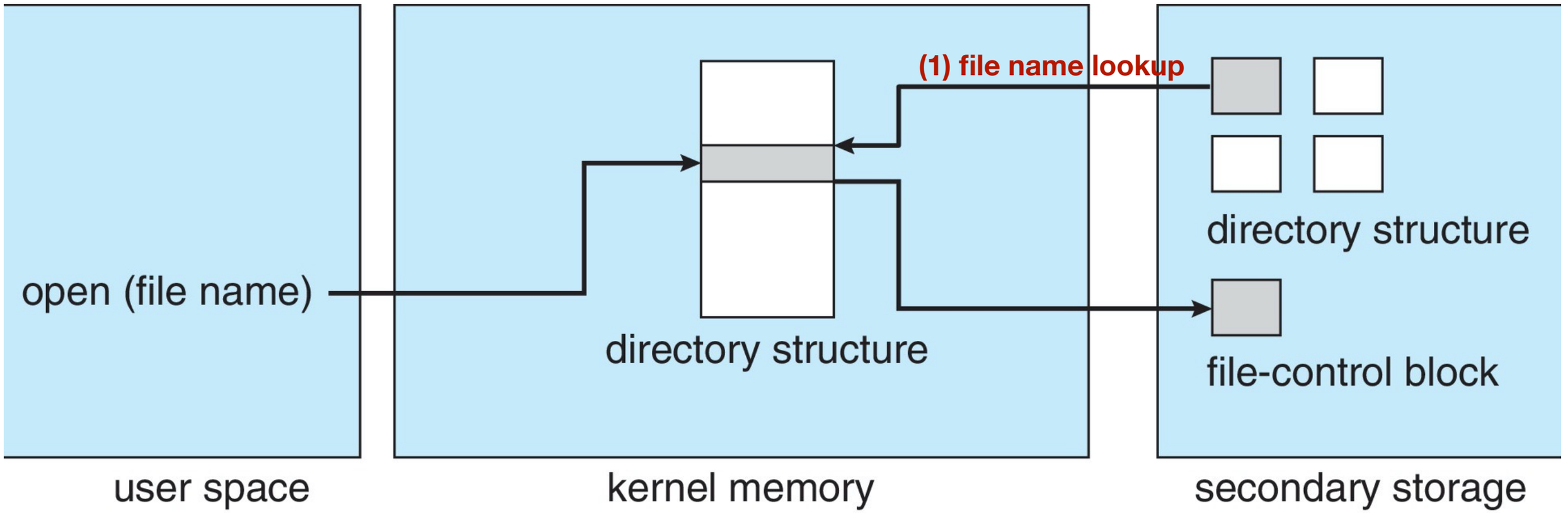
- **mount table** contains a pointer to the volume control block (i.e., the superblock) of each mounted file system, the file system type, etc.
- cached directory structure holds directory information for recently accessed directories
- system-wide **open-file table** contains a copy of the FCB of each open file as well as other information
- per-process open-file table (aka **file descriptor table**) contains a pointer to the appropriate entry in the system-wide open-file table
  - stored in PCB

In-memory file system data structures

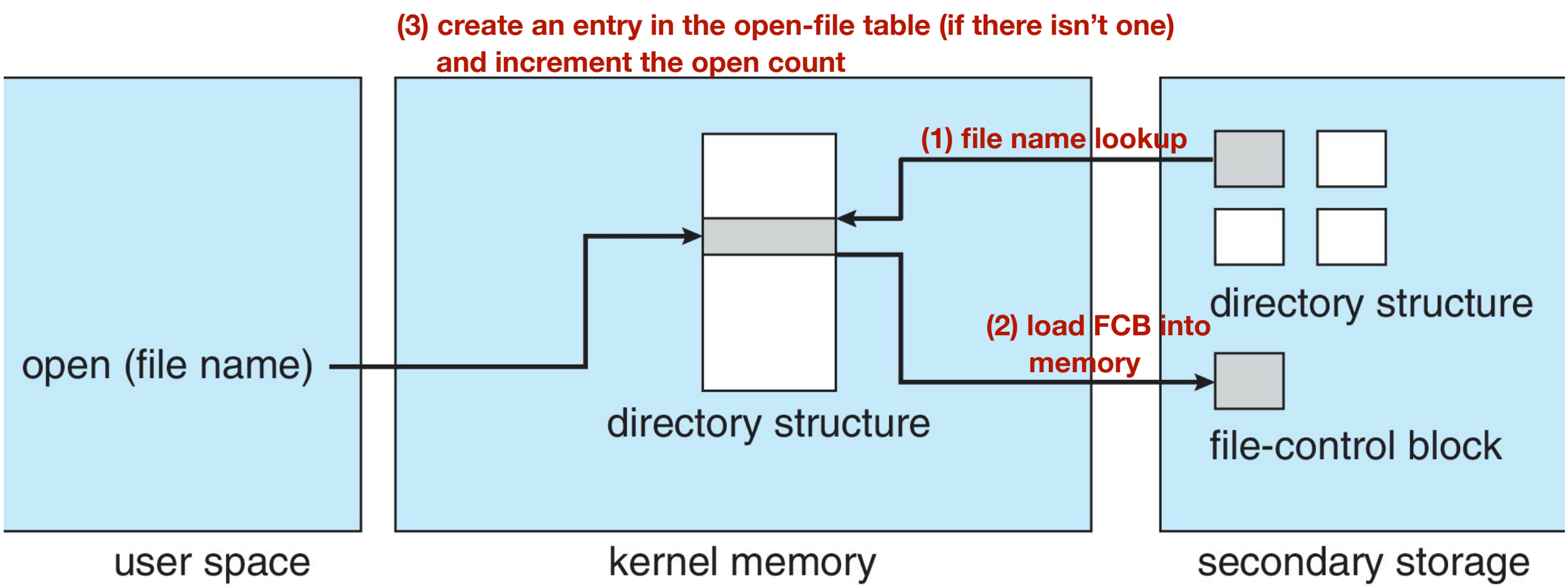
the following data structures are loaded in memory at mount time for performance improvement

- **mount table** contains a pointer to the volume control block (i.e., the superblock) of each mounted file system, the file system type, etc.
- cached directory structure holds directory information for recently accessed directories
- system-wide **open-file table** contains a copy of the FCB of each open file as well as other information
- per-process open-file table (aka **file descriptor table**) contains a pointer to the appropriate entry in the system-wide open-file table
  - stored in PCB
- buffers hold blocks that are being read from disk or written to disk
  - a process writing to disk is actually writing to a buffer; OS writes buffered data to disk asynchronously when convenient
  - a process reading from disk is actually reading from a buffer; OS may read blocks from disk ahead of time to fill the buffer!

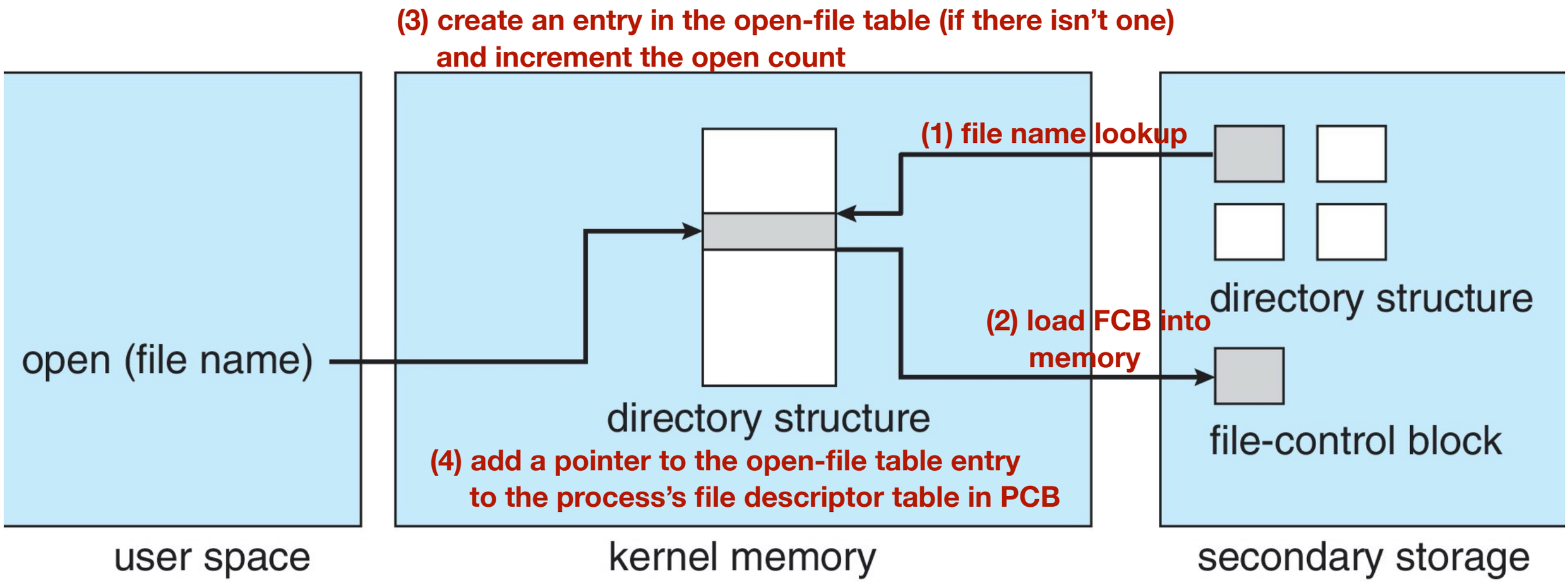
Revisiting the `open` operation



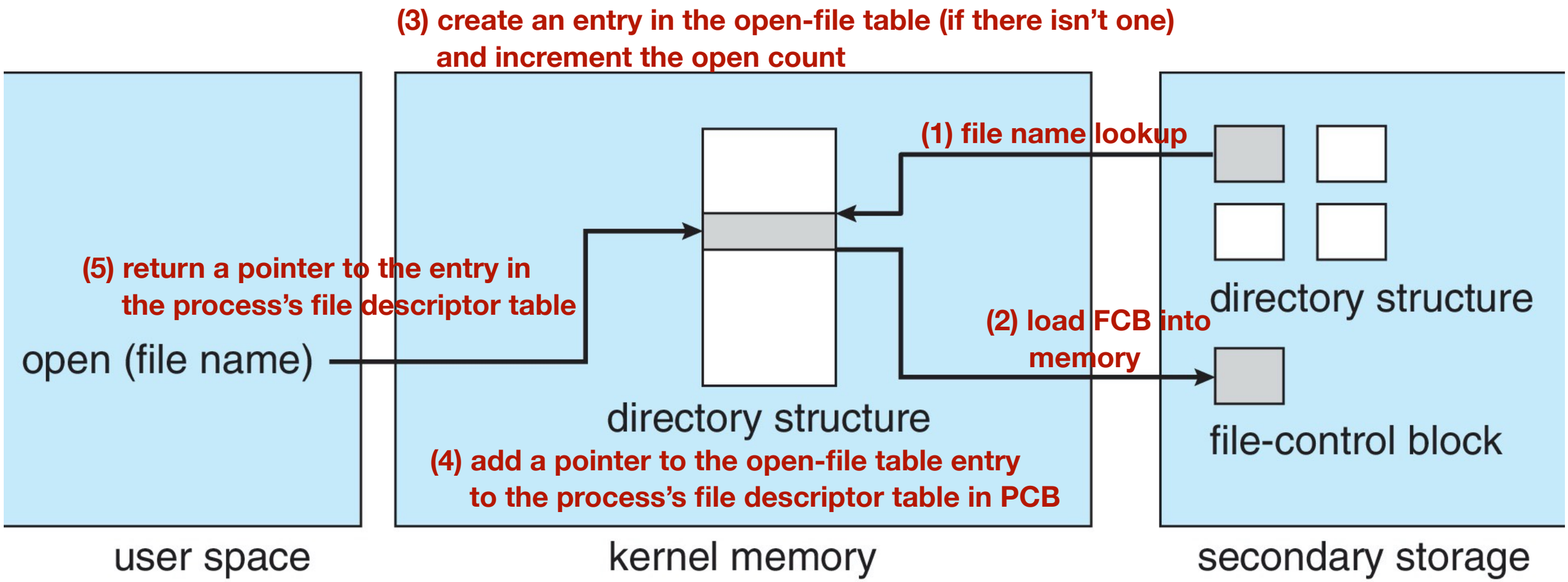
Revisiting the `open` operation



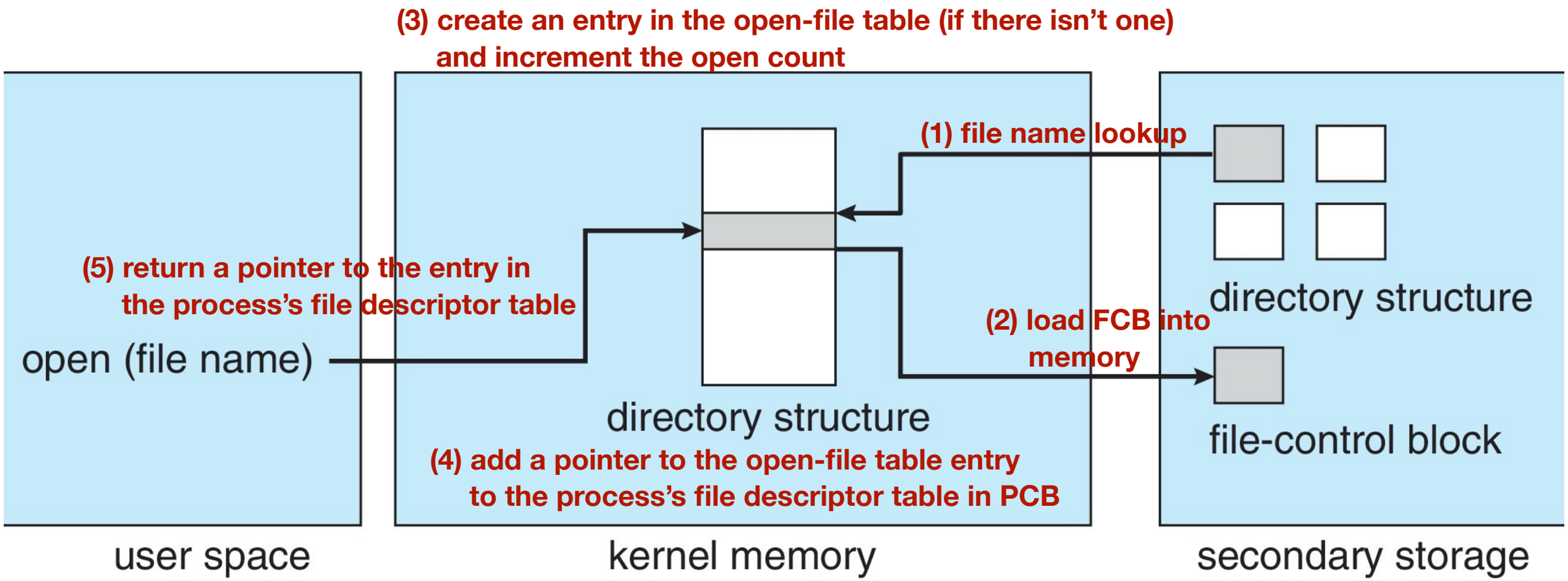
Revisiting the `open` operation



Revisiting the `open` operation



Revisiting the open operation

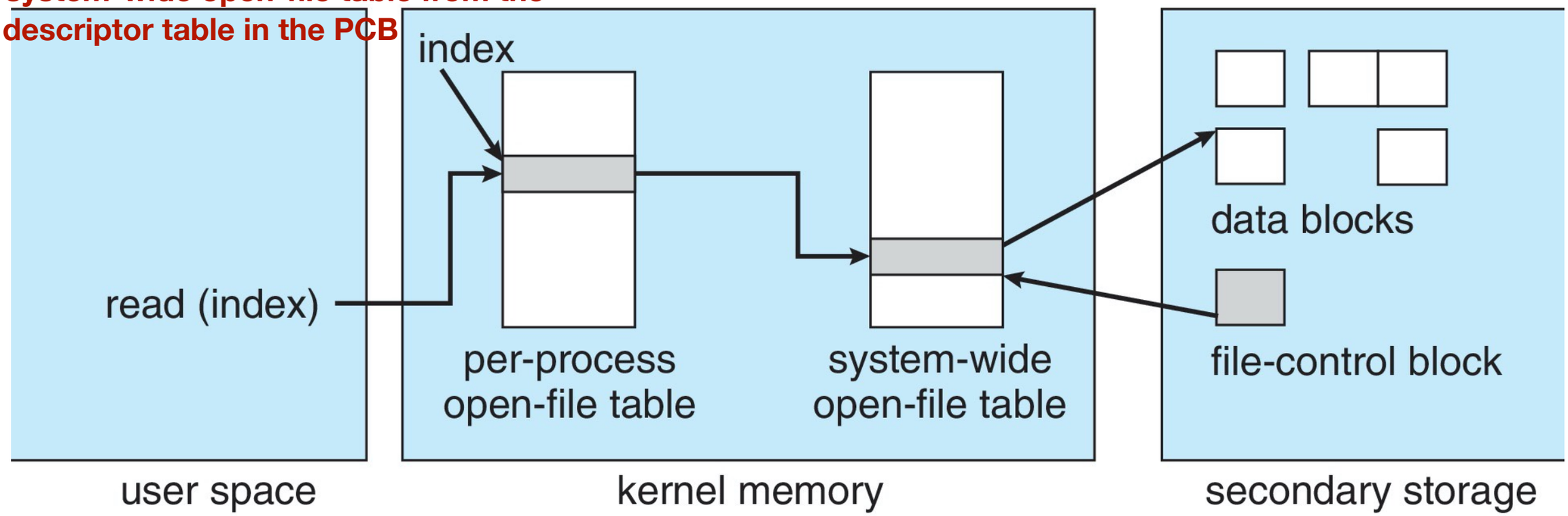


Notes:

1. Steps 1 and 2 are skipped if file is already in the system-wide open file table
this saves substantial overhead
2. parts of the directory structure are cached in memory to speed up the look up process

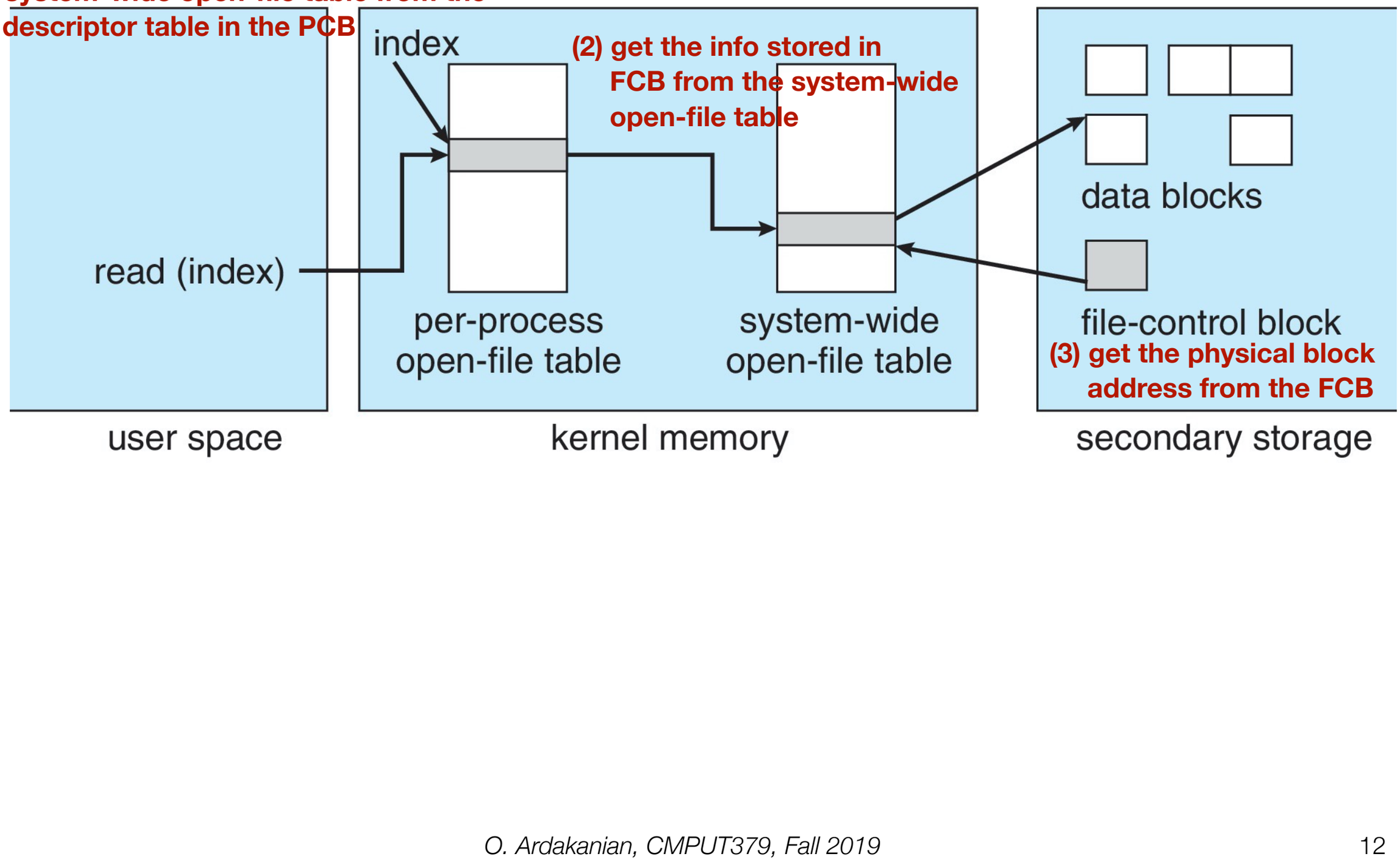
Revisiting the `read` operation

(1) use the file descriptor to get the pointer to the system-wide open-file table from the file descriptor table in the PCB

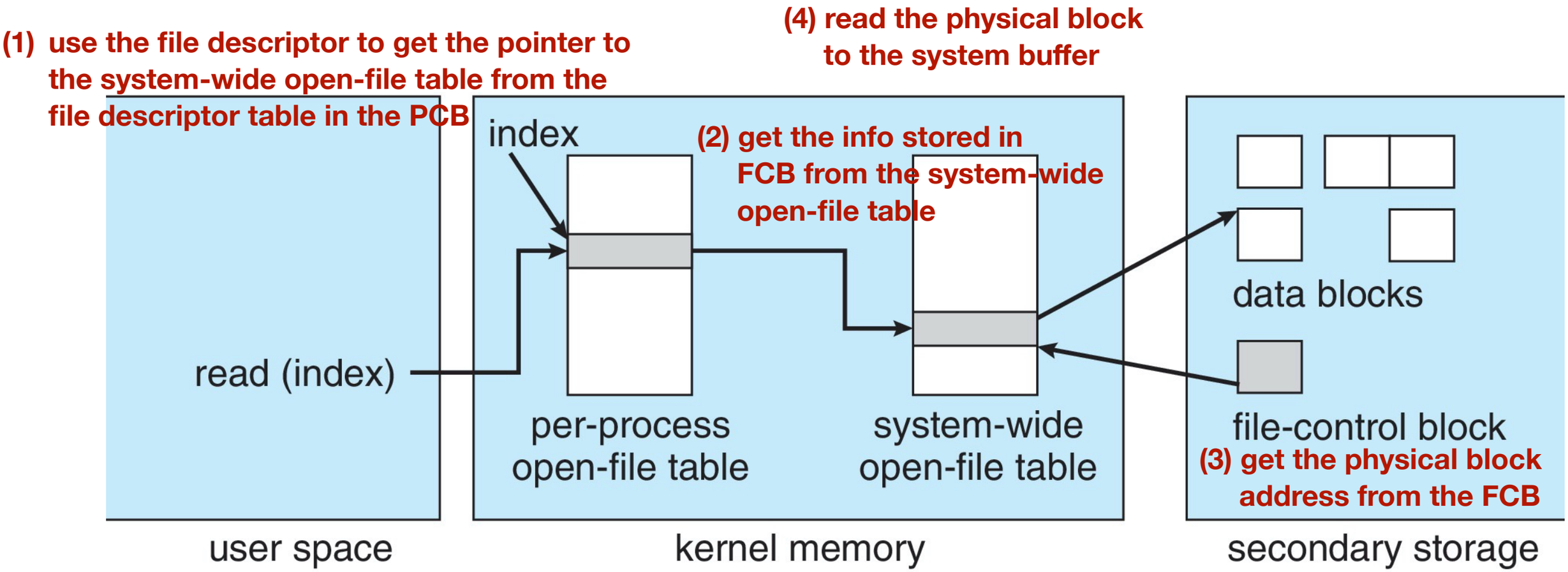


Revisiting the `read` operation

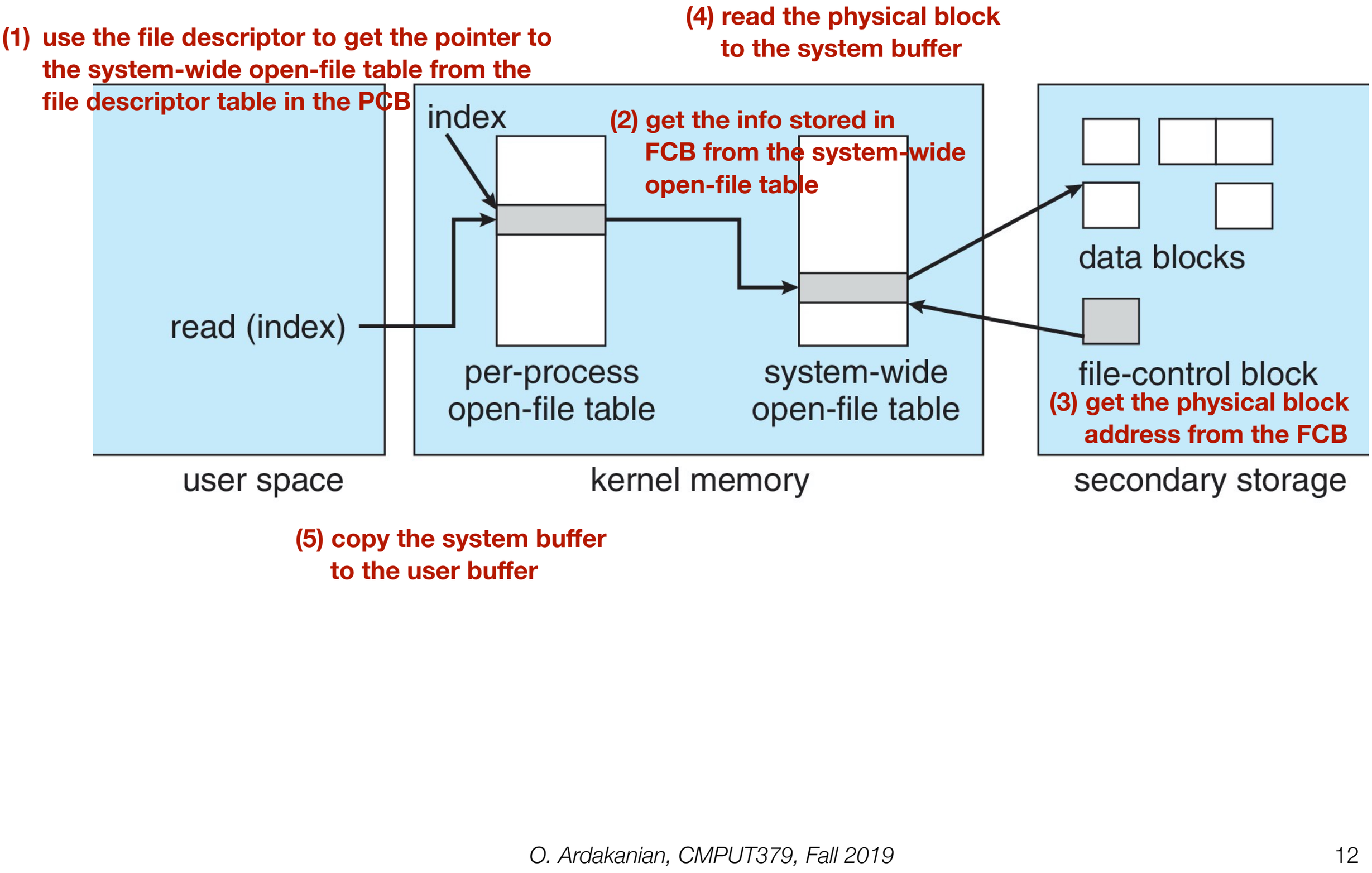
(1) use the file descriptor to get the pointer to the system-wide open-file table from the file descriptor table in the PCB



Revisiting the `read` operation



Revisiting the `read` operation



Directory

- directory is a symbol table used to translate file name into index node number
  - OS uses numbers to refer to files; users prefer textual names
  - to get a pointer to FCB, we have to lookup the file name (unless file descriptor is available)
- directory creates the namespace of files

Directory

- directory is a symbol table used to translate file name into index node number
  - OS uses numbers to refer to files; users prefer textual names
  - to get a pointer to FCB, we have to lookup the file name (unless file descriptor is available)
- directory creates the namespace of files
- each volume (holding a file system) contains information about files in the volume table of contents or directory structure
  - a directory structure organizes all files and directories (into a large tree)

Directory implementation

- linear list of directory entries (**dentry** nodes) where each entry is a file name and its associated index node number
  - simple to program but slow if implemented using a linked list (linear search time)
  - could keep the list sorted and use binary search to find a file name (complicates creating and deleting files)
  - could use a balanced tree (e.g., B+ tree) for best performance

Directory implementation

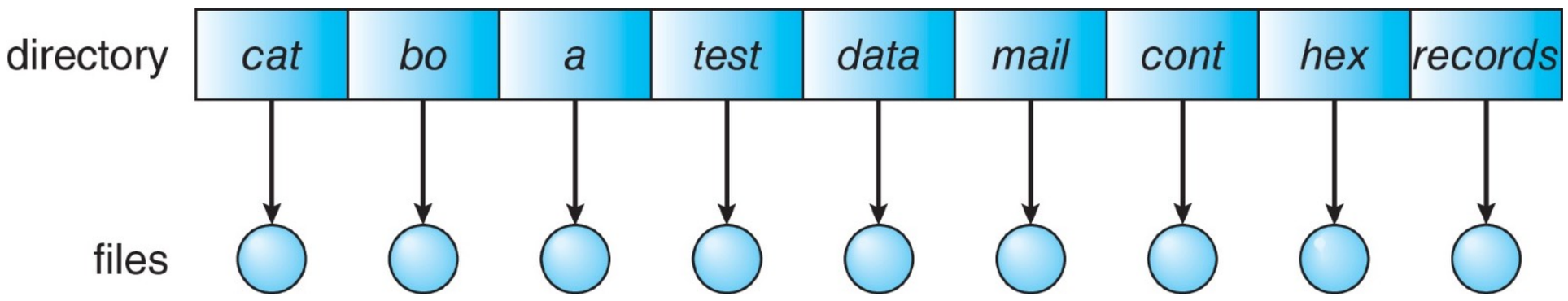
- linear list of directory entries (**dentry** nodes) where each entry is a file name and its associated index node number
  - simple to program but slow if implemented using a linked list (linear search time)
  - could keep the list sorted and use binary search to find a file name (complicates creating and deleting files)
  - could use a balanced tree (e.g., B+ tree) for best performance
- hash table: linear list of directory entries with a hash table used to get a pointer to the directory entry
  - decreases directory search time
  - what if two file names are hashed to the same location? collisions must be handled
    - each location can be a linked list of entries instead of a single entry (chained-overflow method)

Operations performed on a directory

- search for a file in directory structure
  - locate the entry for a given file name
- create a file
  - add an entry to the directory
- delete a file
  - remove the corresponding entry from the directory
- list a directory (the `ls` command in UNIX)
  - list all files in the directory along with the content of their directory entries
- rename a file
  - modify the appropriate directory entry
- traverse the file system

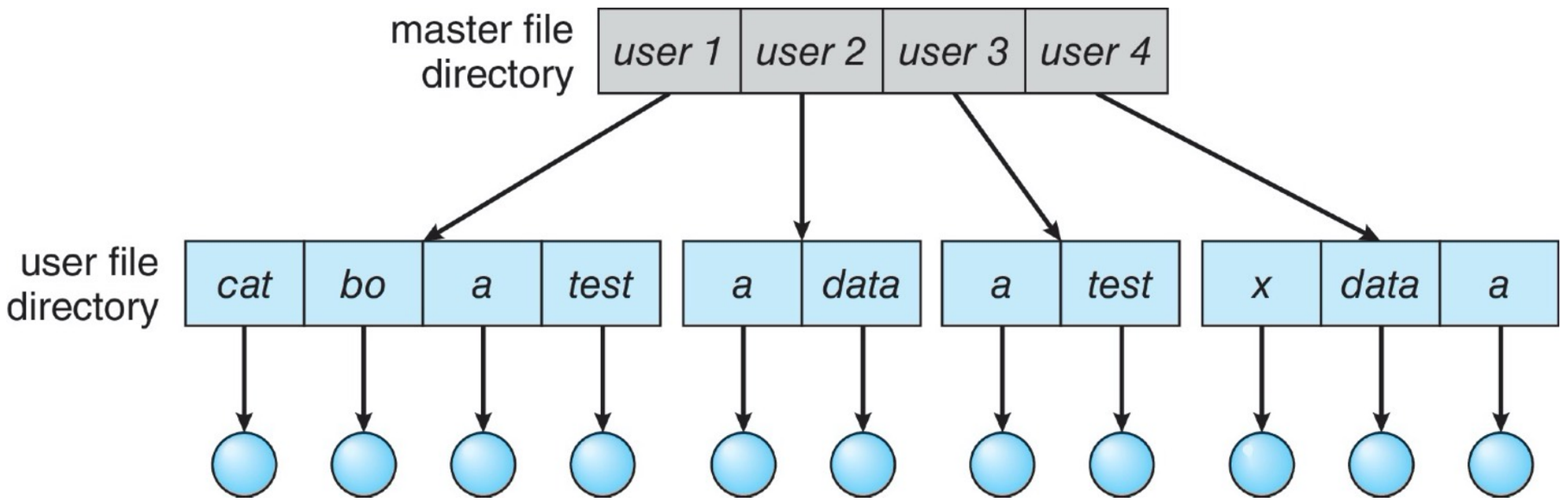
Single-level directory structure

- one name space for the entire disk
  - file names must be unique (naming problem)
    - difficult when there are many files
    - confusion of file names among different users
  - directory structure is held in a special area of disk
  - files cannot be grouped into subdirectories (grouping problem)



Two-level directory structure

- each user has a separate directory, known as the **user file directory** (UFD)
  - can have the same file name for different users, but a user's files must still have unique names
- the **master file directory** (MFD) is indexed by user name (or uid) and each of its entries points to the UFD of a user
  - new UFDs can be created by system administrators and added to MFD
- to create or delete a file, the UFD of the user is searched only
- files cannot be grouped into subdirectories (grouping problem)



Two-level directory structure

- a user name and a file name define a **path name** which is unique
  - for example `/userx/a.txt` is the path name
  - the volume name is specified in the path name in some systems, e.g., `/u/userx/a.txt` might specify volume `u` in UNIX

Two-level directory structure

- a user name and a file name define a **path name** which is unique
  - for example `/userx/a.txt` is the path name
  - the volume name is specified in the path name in some systems, e.g., `/u/userx/a.txt` might specify volume `u` in UNIX
- disadvantage: isolates one user from another, making sharing and cooperation complex

Two-level directory structure

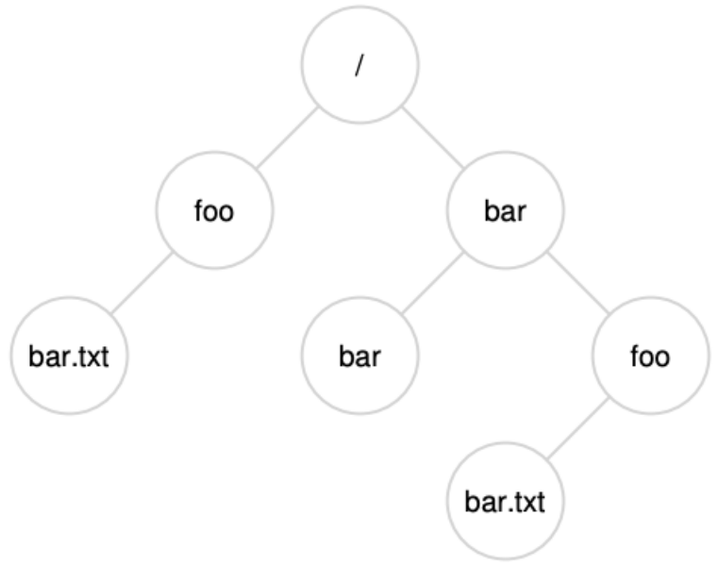
- a user name and a file name define a **path name** which is unique
  - for example `/userx/a.txt` is the path name
  - the volume name is specified in the path name in some systems, e.g., `/u/userx/a.txt` might specify volume `u` in UNIX
- disadvantage: isolates one user from another, making sharing and cooperation complex
- where should we put the system files? in each UFD? or in a special UFD?
  - the system should always search the local UFD first and if the file is not found, search the special UFD; this is known as the **search path**

Multilevel directory structure

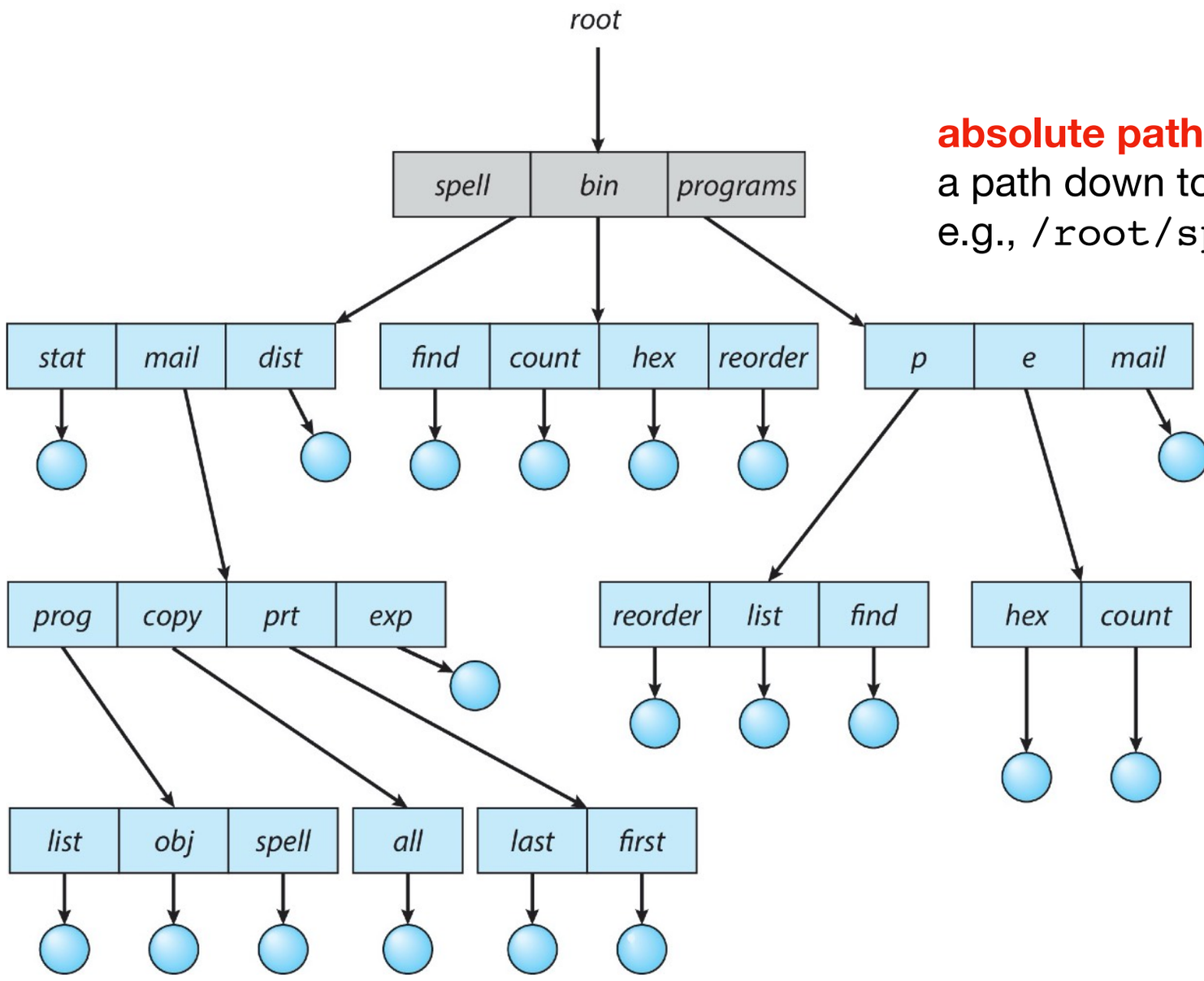
- why not allow users to create subdirectories to better organize their files?
  - each directory is a special file that may contain other files; one bit defines each directory entry as a file (when 0) or a subdirectory (when 1)

Multilevel directory structure

- why not allow users to create subdirectories to better organize their files?
  - each directory is a special file that may contain other files; one bit defines each directory entry as a file (when 0) or a subdirectory (when 1)
- tree-structured name space; adopted in UNIX and all other modern operating systems
  - store directories on disk, just like files except that the file descriptor for directories has a special flag bit
  - user programs read directories just like any other file, but only special system calls can write directories
  - each process has a current directory
  - there is one special root directory
  - each directory contains dentry nodes in no particular order
    - the file referred to by a name may be another directory



Multilevel directory structure



absolute path name: begins at the root and follows a path down to the specified file
e.g., `/root/spell/mail/prt/last`

possible to change the current working directory

relative path name: defines a path from the current working directory
e.g., `prt/last` points to the same file if the pwd is `/root/spell/mail`
`../prt/last` points to the same file if the pwd is `/root/spell/mail/prog`

Referential naming

- hard/nonsymbolic links (`ln` command in UNIX)
  - a hard link adds a second connection to a file
- OS maintains reference counts, so it will only delete a file after the last link to it has been deleted

Referential naming

- hard/nonsymbolic links (`ln` command in UNIX)
  - a hard link adds a second connection to a file
- OS maintains reference counts, so it will only delete a file after the last link to it has been deleted
- **Problem**: user can create circular links with directories and then the OS can never delete the disk space
- **Solution**: no hard links to directories

Referential naming

- soft/symbolic links (`ln -s` command in UNIX)
- a soft link only makes a symbolic pointer from one file to another ($B \rightarrow A$)
  - removing B does not affect A
  - removing A leaves the name B in the directory, but its contents no longer exists
- when the file system encounters a symbolic link it automatically translates it if possible

Referential naming

- soft/symbolic links (`ln -s` command in UNIX)
- a soft link only makes a symbolic pointer from one file to another ($B \rightarrow A$)
  - removing B does not affect A
  - removing A leaves the name B in the directory, but its contents no longer exists
- when the file system encounters a symbolic link it automatically translates it if possible
- **Problem**: circular links can create infinite loops (e.g., trying to list all the files in a directory and its subdirectories)
- **Solution**: limit the number of links traversed

Protection

- OS must allow users to control sharing of their files, and subsequently grant or deny access to file operations depending on access control information
  - associate with each file/directory an access-control list (ACL) specifying user names and the types of access allowed for each users
  - problems:
    - constructing ACL is a tedious task when there are many users
    - need to know all users in advance
    - the size of the directory entry is no longer fixed complicating space management

Protection

- OS must allow users to control sharing of their files, and subsequently grant or deny access to file operations depending on access control information
  - associate with each file/directory an access-control list (ACL) specifying user names and the types of access allowed for each users
  - problems:
    - constructing ACL is a tedious task when there are many users
    - need to know all users in advance
    - the size of the directory entry is no longer fixed complicating space management
- how to condense the length of ACL?
  - define a small number of user classes

Protection

- OS must allow users to control sharing of their files, and subsequently grant or deny access to file operations depending on access control information
  - associate with each file/directory an access-control list (ACL) specifying user names and the types of access allowed for each users
  - problems:
    - constructing ACL is a tedious task when there are many users
    - need to know all users in advance
      - the size of the directory entry is no longer fixed complicating space management
- how to condense the length of ACL?
  - define a small number of user classes
- access control in UNIX
  - three categories of users (owner, group, public)
  - a field of 3 bits for 3 access privileges (read, write, execute)
    - keep a separate field for each category of users
  - maintain a bit for each privilege

```
111101100
rwx          7: user
   r-x       5: group
      r--    4: public
```

Protection

- OS must allow users to control sharing of their files, and subsequently grant or deny access to file operations depending on access control information
  - associate with each file/directory an access-control list (ACL) specifying user names and the types of access allowed for each users
  - problems:
    - constructing ACL is a tedious task when there are many users
    - need to know all users in advance
    - the size of the directory entry is no longer fixed complicating space management
- how to condense the length of ACL?
  - define a small number of user classes
- access control in UNIX
  - three categories of users (owner, group, public)
  - a field of 3 bits for 3 access privileges (read, write, execute)
    - keep a separate field for each category of users
  - maintain a bit for each privilege

```
chmod 774 sample.tex
111111100
rwx           7: user
   rwx        7: group
      r--     4: public
```