File System Calls

<table>
<tr><th colspan="7">File System Calls</th></tr>
<tr><th>Return File Desc</th><th colspan="2">Use of namei</th><th>Assign inodes</th><th>File Attributes</th><th>File I/O</th><th>File Sys Structure</th><th>Tree Manipulation</th></tr>
<tr><td>open
(creat)
dup
pipe
close</td><td colspan="2">open
(creat)
chdir
chroot
chown
chmod
stat
link
unlink
mknod
mount
umount</td><td>(creat)
mknod
link
unlink</td><td>chown
chmod
stat</td><td>read
write
lseek</td><td>mount
umount</td><td>chdir
chown</td></tr>
<tr><td colspan="8">Lower Level File System Algorithms</td></tr>
<tr><td colspan="2">namei
iget iput</td><td colspan="2">ialloc ifree</td><td colspan="4">alloc free bmap</td></tr>
<tr><td colspan="8">Buffer Allocation Algorithms</td></tr>
<tr><td colspan="8">getblk brelse bread breada bwrite</td></tr>
</table>

U Area

- *U area* is the working area of a process.
- Each process has its own *u area*.
  - user file descriptor table
  - I/O parameters for read/write system calls
  - etc.

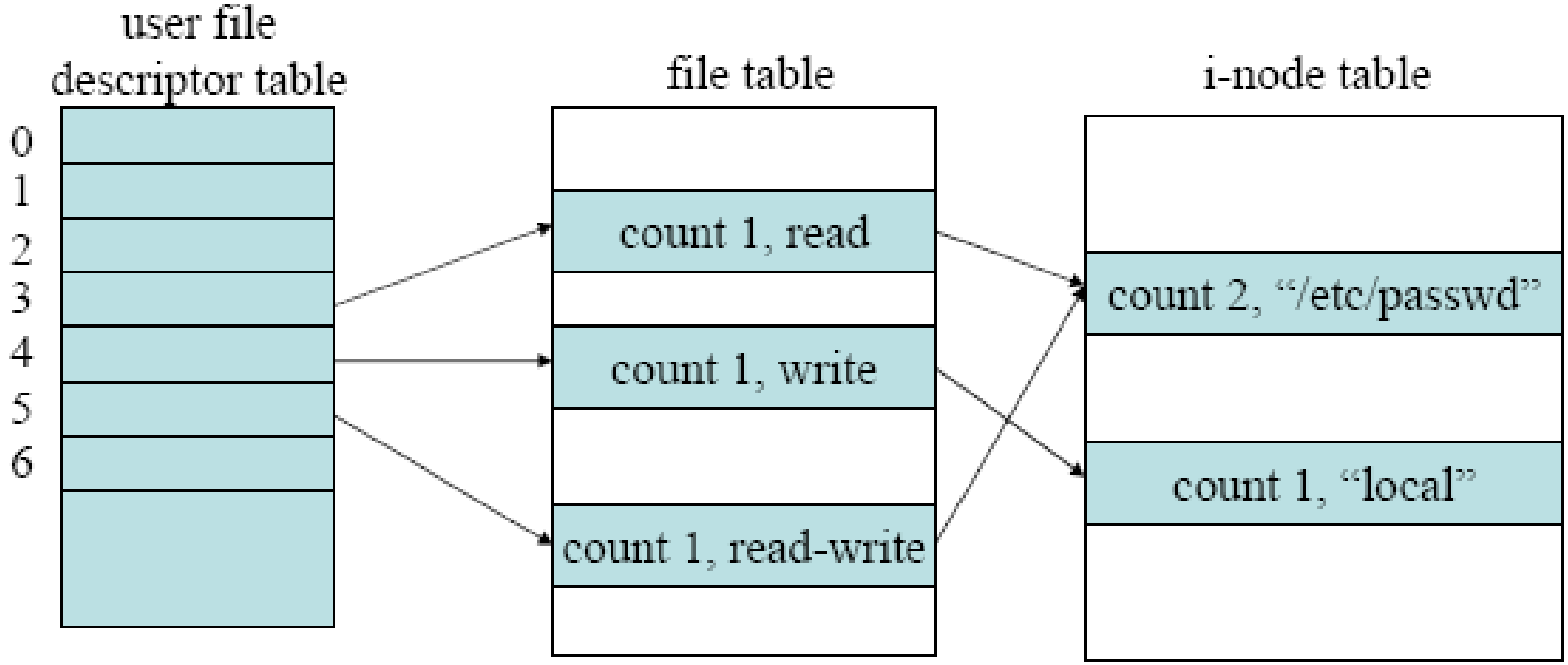
Open

- Syntax: fd = open(pathname, flags, modes)
- The *open* system call returns an integer called the user *file descriptor*.
- The kernel searches the file system for the file by *namei*.
- It checks permissions for opening the file after it finds the in-core i-node and allocates an entry in the file table for the open file
  - The file table entry contains a pointer to the i-node of the open file and a field that indicates the byte offset in the file.
- The kernel allocates an entry in a private table in the process *u area*, called the user file descriptor table.

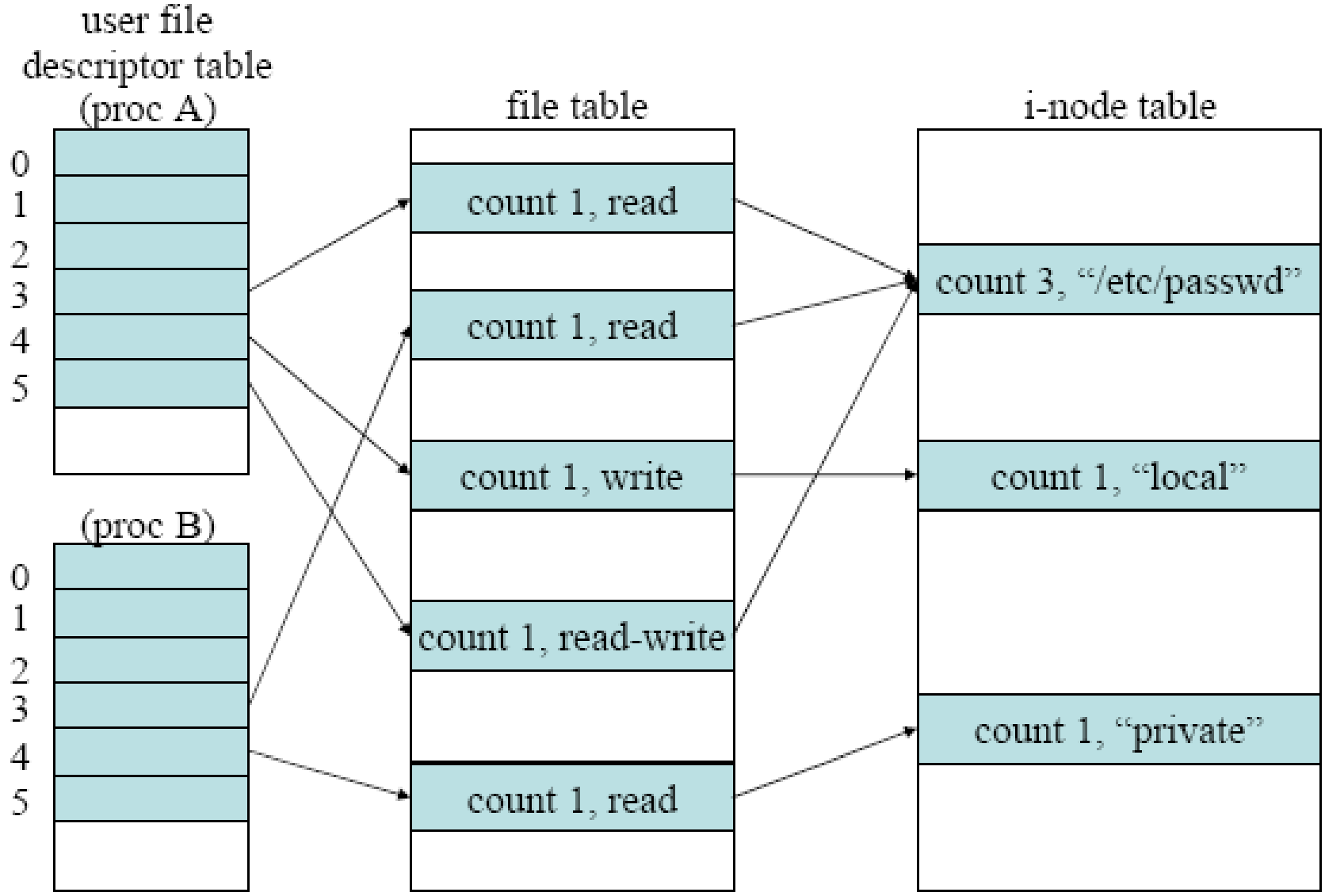
```
algorithm open
inputs: file name
        type of open
        file permissions
{
   convert file name to inode (namei);
   if (file does not exist or not permitted access)
      return (error);
   allocate file table entry for inode, initialize count, offset;
   allocate user file descriptor entry, set pointer to file table entry;
   if (type of open specifies truncate file)
      free all file blocks (free);
   unlock(inode);       /* locked above in namei */
   return (user file descriptor);
}
```

Opening a File

Data Structures after Open



fd1 = open("/etc/passwd", O_RDONLY);
fd2 = open("local", O_WRONLY);
fd3 = open("/etc/passwd", O_RDWR);



proc B: fd1 = open("/etc/passwd", O_RDONLY);
fd2 = open("private", O_RDONLY);

Data Structures

- The user file descriptor table entry could conceivably contain the file offset of the next I/O operation and point directly to the in-core entry, eliminating the need for a separate kernel file table.
- This structure allows sharing of the offset pointer between several user file descriptors.
- The *dup* and *fork* system calls manipulate the data structures to allow such sharing.
- The first three user file descriptors (0, 1, and 2) are called the *standard input*, *standard output*, and *standard error* file descriptors.

Read (1/2)

- Syntax: number = read(fd, buffer, count);
- The kernel gets the file table entry that corresponds to the user file descriptor.
- The kernel sets several I/O parameters in the *u area*, eliminating the need to pass them as function parameters.
  - I/O mode to indicate that a read is being done
  - a flag to indicate that the I/O will go to user address space
  - a count field to indicate the number of bytes to read
  - the target address of the user data buffer
  - an offset field (from the file table) to indicate the byte offset into the file where the I/O should begin
- The kernel locks the i-node before it reads the file.

Read (2/2)

- The kernel converts the file byte offset into block number by *bmap*.
- After reading the block into a buffer by *bread* and *breada*, it copies the data from the block to the target address in the user space.
- The kernel updates the I/O parameters in the *u area* according to the number of bytes it read.
- If the user request is not satisfied, the kernel repeats the entire cycle.
- The cycle completes either when the kernel completely satisfies the user request, or if the kernel encounters an error in reading the data from disk or in copying the data to user space.
- The kernel updates the offset in the file table according to the number of bytes it actually read.

```
algorithm read
input: user file descriptor
        address of buffer in user space
        number of bytes to read
output: count of bytes copied into user space
{
   get file table entry from user file descriptor;
   check file accessibility;
   set parameters in u area for user address,
         byte count, I/O to user;
   get inode from file table;
   lock inode;
   set byte offset in u area from file table offset;
   while (count not satisfied) {
      convert file offset to disk block (bmap);
      calculate offset into block, number of
            bytes to read;
      if (number of bytes to read is 0)
         break; /* trying to read end of file */
      read block (breada or bread);
      copy data from system buffer
            to user address;
      update u area fields for file
            byte, offset, read count,
            address to write into user
            space;
      release buffer;
            /* locked in bread */
   } /* while */
   unlock inode;
   update file table offset for next
         read;
   return (total number of bytes
            read);
}
```

Reading a file

Sample Program for Reading a File

```
#include <fcntl.h>
main( )
{
   int fd;
   char buf1[20], buf2[1024];

   fd = open("/etc/passwd", O_RDONLY);
   read(fd, buf1, 20);
   read(fd, buf2, 1024);
   read(fd, buf1, 20);
}
```

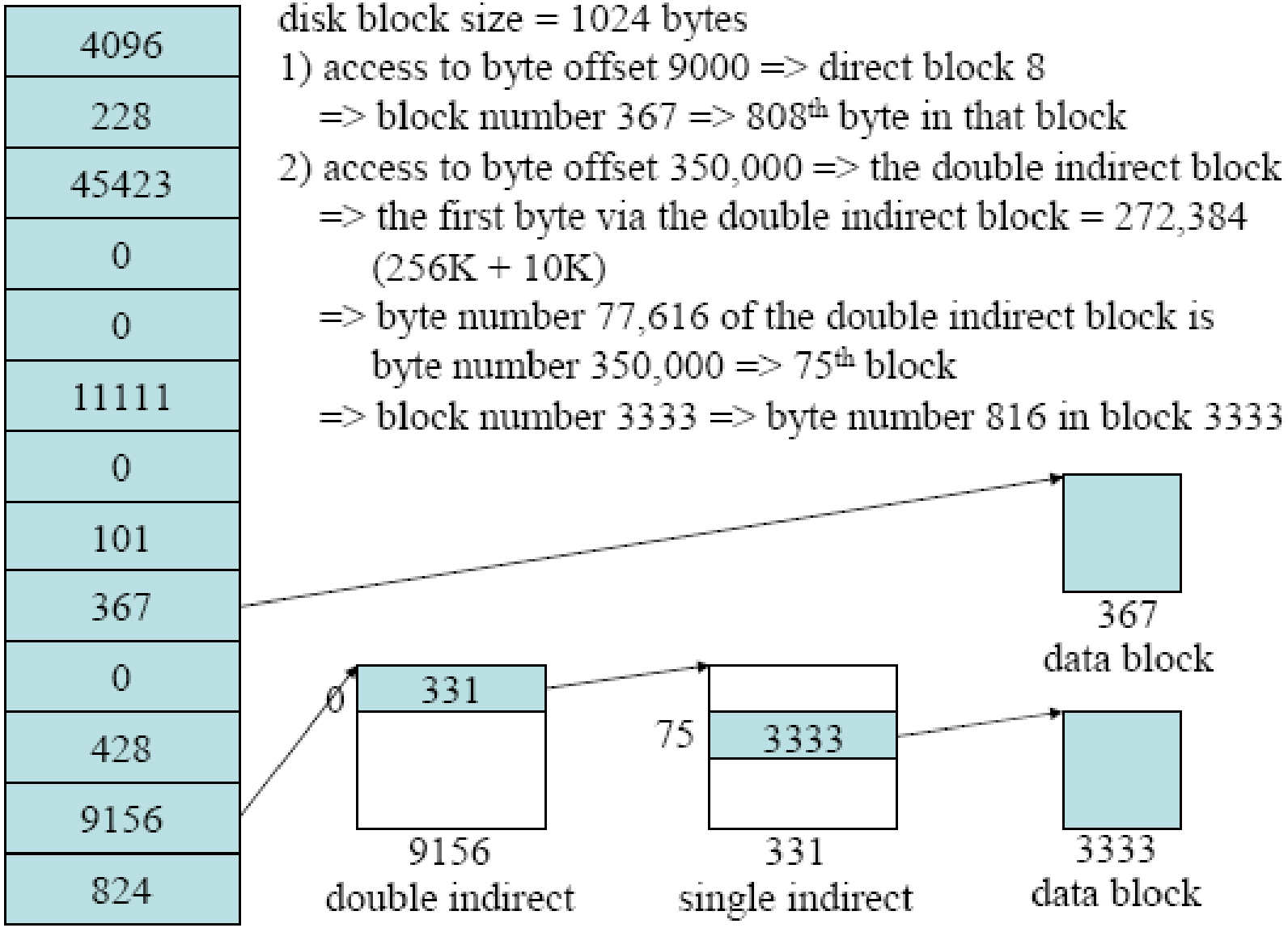
- the first "read" reads the first block of the file but copies only 20 bytes to the user address *buf1*.
- the second "read" may find the buffer cache for the first block of the file, and then copies 1004 bytes to the user address *buf2*.
- it cycles the read loop and read the second block of the file. 20 bytes are copied to the user address.
- the third "read" may find the buffer cache and it copies 20 bytes to the user address.
- This example shows how advantageous it is for I/O requests to start on file system block number boundaries.

Read Ahead

- As the kernel goes through the *read* loop, it determines whether a file is subject to read-ahead: if a process *read*s two blocks sequentially, the kernel assumes that all subsequent *read*s will be sequential until proven otherwise.
- During each iteration through the loop, the kernel saves the next logical block number in the in-core i-node and, during the next iteration, compares the current logical block number to the value previously saved.
- If they are equal, the kernel calculates the physical block number for read-ahead and saves its value in the *u area* for use in *breada*.

Reading Block Number 0

- It is possible for some block numbers in an i-node or in indirect blocks to have the value 0, even though later blocks have nonzero value.
- If a process attempts to read data from such a block, the kernel satisfies the request by allocating an arbitrary buffer in the read loop, clearing its contents to 0, and copying it to the user address.
- This case is different from the case where a process encounters the end of file, meaning that no data was ever written to any location beyond the current point.
- When encountering end of file, the kernel returns no data to the process.


4096
228
45423
0
0
11111
0
101
367
0
428
9156
824
disk block size = 1024 bytes
1) access to byte offset 9000 => direct block 8
=> block number 367 => 808th byte in that block
2) access to byte offset 350,000 => the double indirect block
=> the first byte via the double indirect block = 272,384
(256K + 10K)
=> byte number 77,616 of the double indirect block is
byte number 350,000 => 75th block
=> block number 3333 => byte number 816 in block 3333
0
331
9156
double indirect
75
3333
331
single indirect
367
data block
3333
data block

Read & i-node Lock

- When a process invokes the *read* system call, the kernel locks the i-node for duration of the call.
  - afterwards, it could go to sleep reading buffer associated with data or with indirect blocks.
- If another process were allowed to change the file while the first process was sleeping, *read* could return inconsistent data.
- The i-node is left locked for the duration of the *read* call, affording the process a consistent view of the file as it existed at the start of the call.

Read & i-node Lock

- Since the i-node is unlocked at the end of a system call, nothing prevents other process from accessing the file and changing its contents.
- It would be unfair for the system to keep an i-node locked from the time a process *open*ed the file until it *close*d the file, because one process could keep a file open and thus prevent other process from ever accessing it.
- To avoid such problems, the kernel unlocks the i-node at the end of each system call that uses it.
- If another process changes the file between the two *read* system calls by the first process, the first process may read unexpected data, but the kernel data structures are consistent.

```
#include  <fcntl.h>
/* process A */
main( )
{
   int fd; char buf[512];
   fd = open("/etc/passwd", O_RDONLY);
   read(fd, buf, sizeof(buf));    /* read1 */
   read(fd, buf, sizeof(buf));    /* read2 */
}

/* process B */
main ( )
{
   int fd, i; char buf[512];
   for (i = 0; i < sizeof(buf); i++)
      buf[i] = 'a';
   fd = open("etc/passwd", O_WRONLY);
   write(fd, buf, sizeof(buf));    /* write1 */
   write(fd, buf, sizeof(buf));    /* write2 */
}
```

- assume both processes complete their *open* calls before either one starts its *read* or *write*, the kernel could execute the *read* and *write* calls in any of six sequences.
- the data that process A *read*s depends on the order that the system executes the system calls of the two processes.
- Use of the *file and record locking* feature allows a process to guarantee file consistency while it has a file *open*.

```
#include  <fcntl.h>
main( )
{
   int fd1, fd2;
   char buf1[512], buf2[512];

   fd1 = open("/etc/passwd", O_RDONLY);
   fd2 = open("/etc/passwd", O_RDONLY);
   read(fd1, buf1, sizeof(buf1));
   read(fd2, buf2, sizeof(buf2));
}
```

- the program shown above shows how a process can open a file more than once and read it via different file descriptors.
- the kernel manipulates the file table offsets associated with the two file descriptors independently, and hence, the arrays buf1 and buf2 should be identical when the process completes, assuming no other process writes "/etc/passwd" in the meantime.

Write (1/2)

- syntax: number = write(fd, buffer, count);
- *writ*ing a regular file is similar to that of *read*ing a regular file.
- If the file does not contain a block that corresponds to the byte offset to be written, the kernel allocates a new block by *alloc* and assigns the block number to the correct position in the i-node's table of contents.
- If the byte offset is that of an indirect block, the kernel may have to allocate several blocks for use as indirect blocks and data blocks.
- The i-node is locked for the duration of the *write*, because the kernel may change the i-node when allocating new blocks.
- When the *write* is complete, the kernel updates the file size entry in the i-node if the file has grown larger.

Write (2/2)

- The kernel goes through an internal loop, as in *read*.
- During each iteration, it determines whether it will write the entire block or only part of it.
- If it writes only part of a block, it must first read the block from disk so as not to overwrite the parts that will remain the same, but if it writes the whole block, it need not read the block.
- The kernel uses a *delayed write* to write the data to disk, caching it in case another process should read or write it soon and avoiding extra disk operations.
- Delayed write is probably most effective in pipes.
- Even for regular files, delayed write is effective if the file is created temporarily and will be read soon.

Lseek

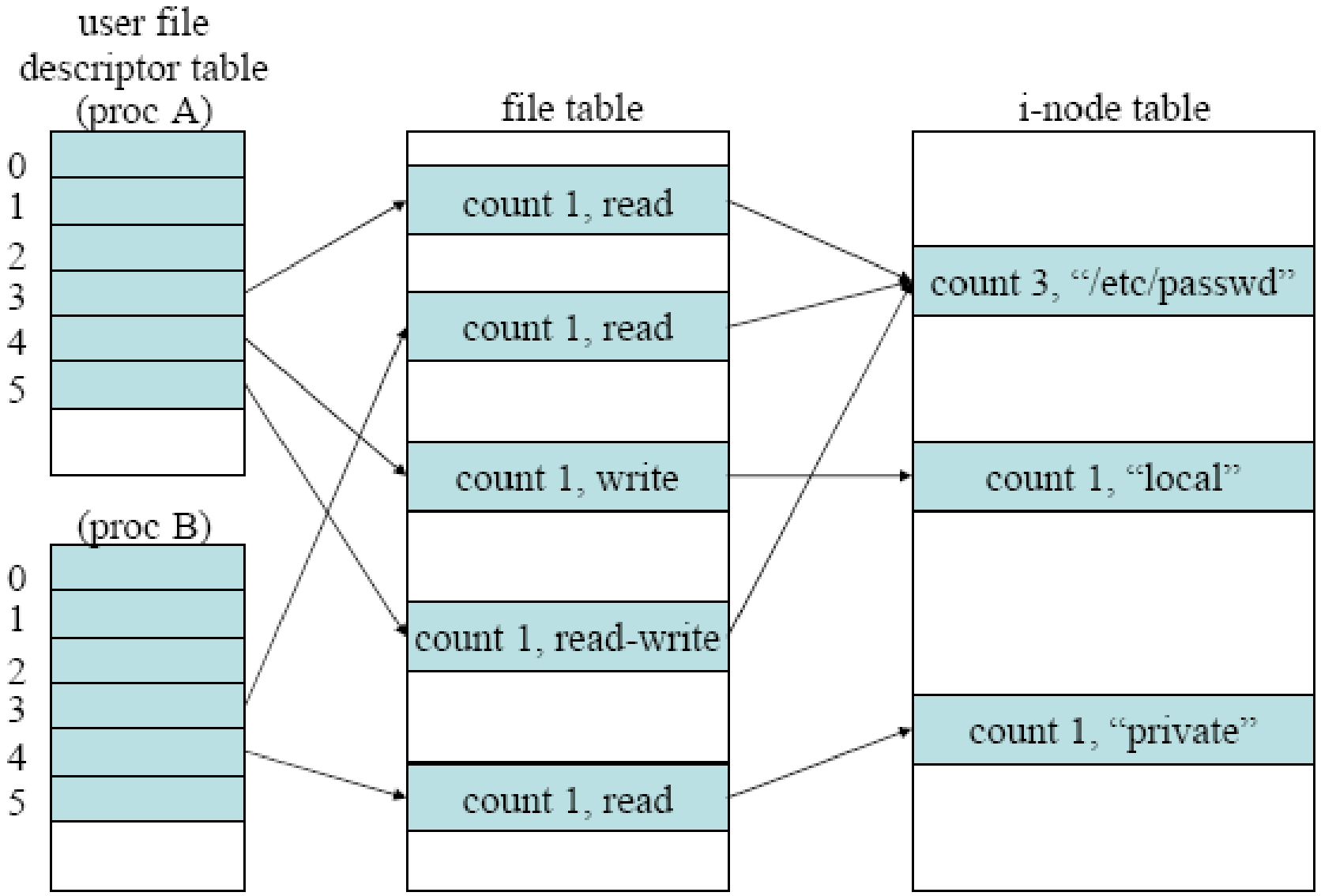
- syntax: position = lseek(df, offset, reference);
- The ordinary use of read and write system calls provides sequential access to a file, but processes can use the *lseek* system call to position the I/O and allow random access to a file.
- The *lseek* system call has nothing to do with seek operation that positions a disk arm over a particular disk sector.
- To implement *lseek*, the kernel simply adjusts the offset value in the file table; subsequent *read* or *write* system calls use the file table offset as their starting byte offset.

Close (1/2)

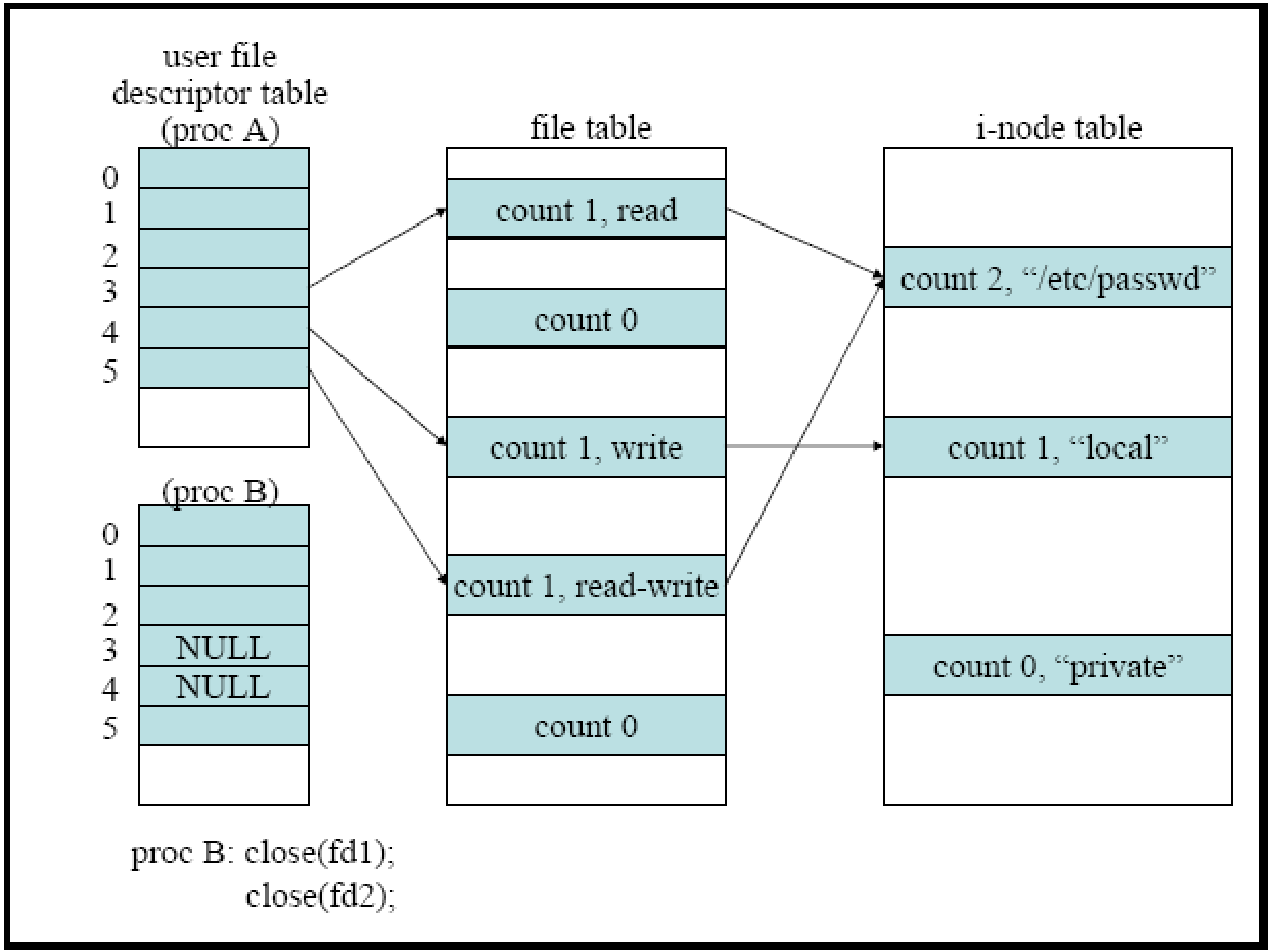
- syntax: close(fd);
- The kernel does the *close* operation by manipulating the file descriptor and the corresponding file table and i-node table entries.
- If the reference count of the file table entry is greater than 1 because of *dup* or *fork* calls, then other user file descriptor reference the file table entry; the kernel decrements the count and the *close* completes.
- If the file table reference count is 1, the kernel frees the entry and releases the in-core i-node originally allocated in the *open* system call (by *iput*).

Close (2/2)

- If other process still reference the i-node, the kernel decrements the i-node reference count but leaves it allocated; otherwise, the i-node is free for reallocation because its reference count is 0.
- When the close system call completes, the user file descriptor table entry is empty.



proc B: fd1 = open("/etc/passwd", O_RDONLY);
fd2 = open("private", O_RDONLY);


user file
descriptor table
(proc A)
0
1
2
3
4
5
(proc B)
0
1
2
3
NULL
4
NULL
5
file table
count 1, read
count 0
count 1, write
count 1, read-write
count 0
i-node table
count 2, "/etc/passwd"
count 1, "local"
count 0, "private"
proc B: close(fd1);
close(fd2);

File Creation (1/3)

- syntax: fd = creat(pathname, modes);
  - same as: open(pathname, O_CREAT | O_TRUNC | O_WRONLY, mode);
- The kernel parses the path name by *namei*.
- If the kernel does not find the path name component in the directory, it will eventually write the name into the empty slot just found.
- If the directory has no empty slots, the kernel remembers the offset of the end of the directory and creates a new slot there.
- It also remembers the i-node of the directory being searched in its *u area* and keeps the i-node locked; the directory will become the parent directory of the new file.
- The kernel does not write the new file name into the directory yet, so that it has less to undo in event of later errors.

File Creation (2/3)

- Assuming no file by the given name, the kernel assigns an i-node for the new file by *ialloc*.
- It then writes the new file name component and the i-node number of the newly allocated i-node in the parent directory at the byte offset saved in the *u area*.
- The kernel writes the newly allocated i-node to disk by *bwrite* before it write the directory with the new name to disk.
- If the system crashes between the write operations for the i-node and the directory, there will be an allocated i-node that is not referenced by any path name in the system but the system will function normally.

File Creation (3/3)

- If, on the other hand, the directory were written before the newly allocated i-node and the system crashed in the middle, the file system would contain a path name that referred to a bad i-node.

```
algorithm creat
input: file name
        permission settings
output: file descriptor
{
   get inode for file name (namei);
   if (file already exists) {
      if (not permitted access) {
         release inode (iput);
         return (error);
      }
   } else {
      assign free inode from file system
                                    (ialloc);
      create new directory entry in parent
            directory; include new file name
            and newly assigned inode number;
   }.
   allocate file table entry for
         inode, initiailize count;
   if (file did exist at time of
         creat)
      free all file blocks (free);
   unlock(inode);
   return (user file descriptor);
}
```

Creating a File

Creation of Special Files

- syntax: mknod(pathname, type and permissions, dev);
- *mknod* creates special files including named pipes, device files, and directories.
- It is similar to *creat* in that the kernel allocates an i-node for the file.
- If the file does not yet exist, the kernel assigns a new i-node on the disk and writes the new file name and i-node number into the parent directory.
- The kernel sets the file type field in the i-node to indicate that the file type is a pipe, directory or special file.
- If the file is a *character special* device file or *block special* device file, it writes the major and minor numbers into the i-node.

```
algorithm make new node
input: node (file name)
        file type
        permissions
        major, minor device number
output: none
{
   if (new node not named pipe and user
           not super user)
      return (error);
   get inode of parent of new node (namei);
   if (new node already exists) {
      release parent inode (iput);
      return (error);
   }
   assign free inode from file system
        for new node (ialloc);
   create new directory entry in
        parent directory: include new
        node name and newly
        assigned inode number;
   release parent directory inode
        (iput);
   if (new node is block or character
           special file)
      write major, minor numbers into
           inode structure;
   release new node  inode (iput);
}
```

Algorithm for Making New Node

Change Directory

- When a new process is created via *fork*, the new process inherits the current directory of the old process in its *u area*, and the kernel increments the i-node reference count accordingly.
- syntax: chdir(pathname);
- The kernel releases the lock to the new i-node but keeps the i-node allocated and its reference count incremented, release the i-node of the old current directory stored in the *u area*.
- *chdir* is similar to *open*, because both system calls access a file and leave its i-node allocated.

```
algorithm change directory
input: new directory name
output: none
{
    get inode for new directory name (namei);
    if (inode not that of directory or process not permitted access to file) {
        release inode (iput);
        return (error);
    }
    unlock inode;
    release "old" current directory inode (iput);
    place new inode into current directory slot in u area;
}
```

Algorithm for Changing Current Directory

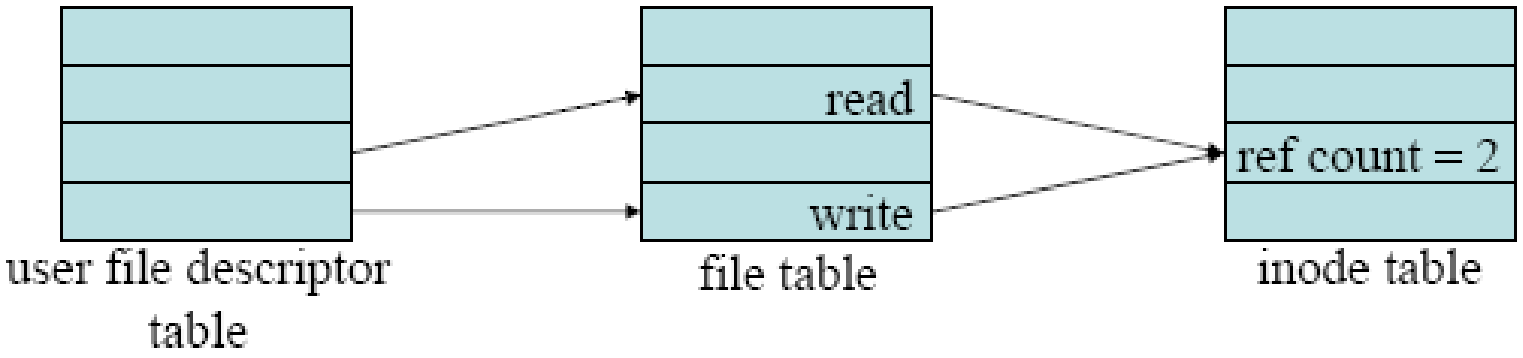
chown, chmod, stat, fstat

- chown(pathname, owner, group);
- chmod(pathname, mode);
- stat(pathname, statbuffer);
- fstat(fd, statbuffer);
- These system calls set or get information in the i-node of the specified file.

Pipes

- Pipes allow transfer of data between processes in a first-in-first-out manner (*FIFO*), and they also allow synchronization of process execution.
- The implementation uses the file system for data storage.
- There are two kinds of pipes: they are identical except for the way that a process initially accesses them.
  - *named pipes* (=> UNIX domain socket):
    processes use *open* to create name pipes.
  - *unnamed pipes*:
    processes use *pipe* to create an unnamed pipes.
- Afterwards, processes use regular system calls for files, such as *read, write*, and *close* when manipulating pipes.

```
algorithm pipe
input: none
output: read file descriptor
        write file descriptor
{
   assign new inode from pipe device (ialloc);
   allocate file table entry for reading, another for writing;
   initialize file table entries to point to new inode;
   allocate user file descriptor for reading, another for writing,
        initialize to point to respective file table entries;
   set inode reference count to 2;
   initialize count of inode readers, writers to 1;
}
```



Reading and Writing Pipes

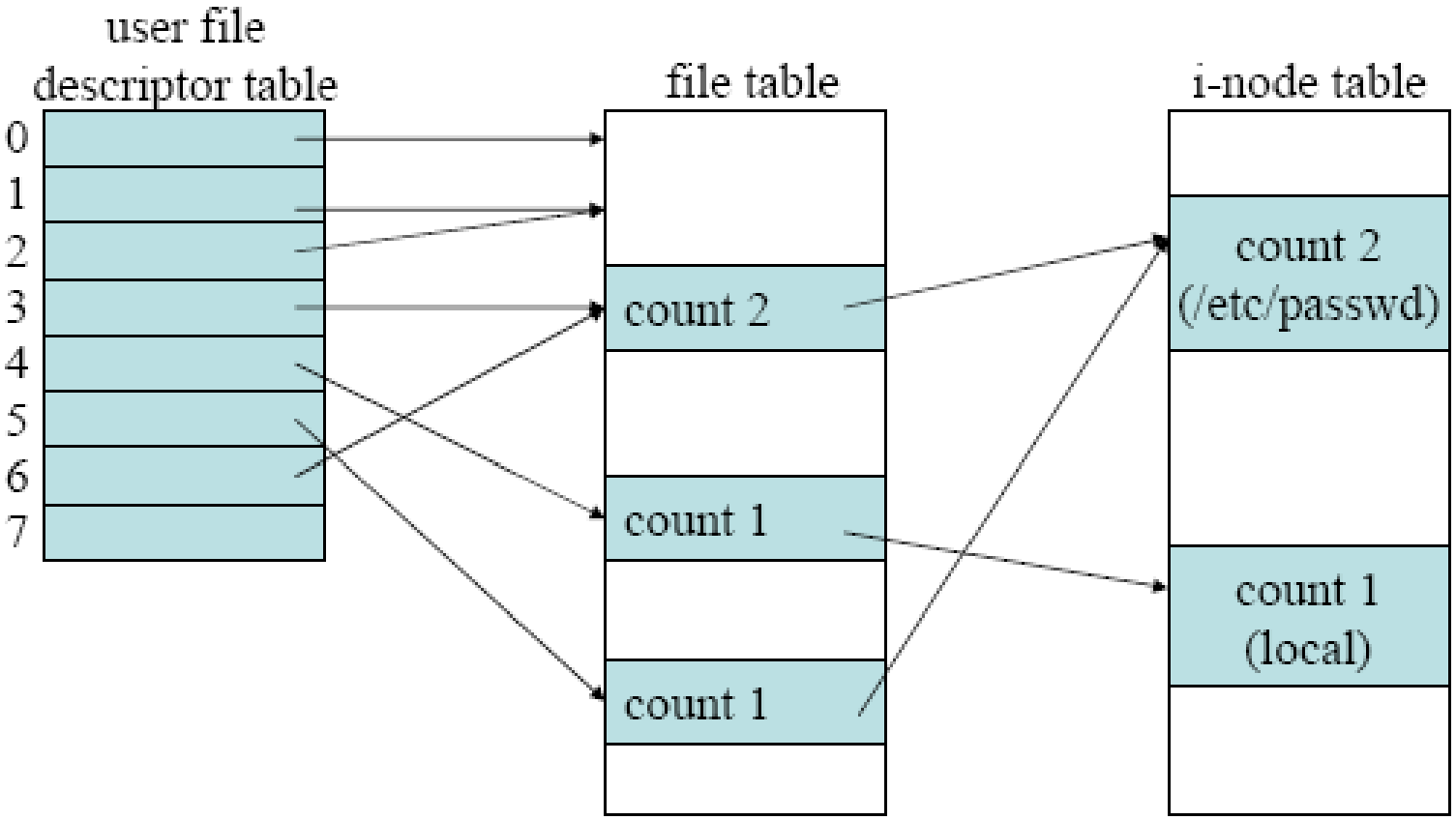
- A pipe should be viewed as if processes *write* into one end of the pipe and *read* from the other end.
- The kernel accesses the data for a pipe exactly as it accesses data for a regular file: it stores data on the pipe device and assigns blocks to the pipe as needed during *write* calls.
- The difference between storage allocation for a pipe and a regular file is that a pipe uses only the direct blocks of the i-node for greater efficiency, although this places a limit on how much data a pipe can hold at a time.
- The kernel manipulates the direct blocks of the i-node as a circular queue, maintaining read and write pointers internally to preserve the FIFO order.

Closing Pipes

- When closing a pipe, a process follows the same procedure it would follow for closing a regular file, except that the kernel does special processing before releasing the pipe's i-node.
- If the count of writer processes drops to 0 and there are processes asleep to read data from the pipe, the kernel awakens them, and they return from their *read* calls without reading any data.
- If the count of reader processes drops to 0 and there are processes asleep to write data to the pipe, the kernel awakens them and sends them a signal to indicate an error condition.

Dup

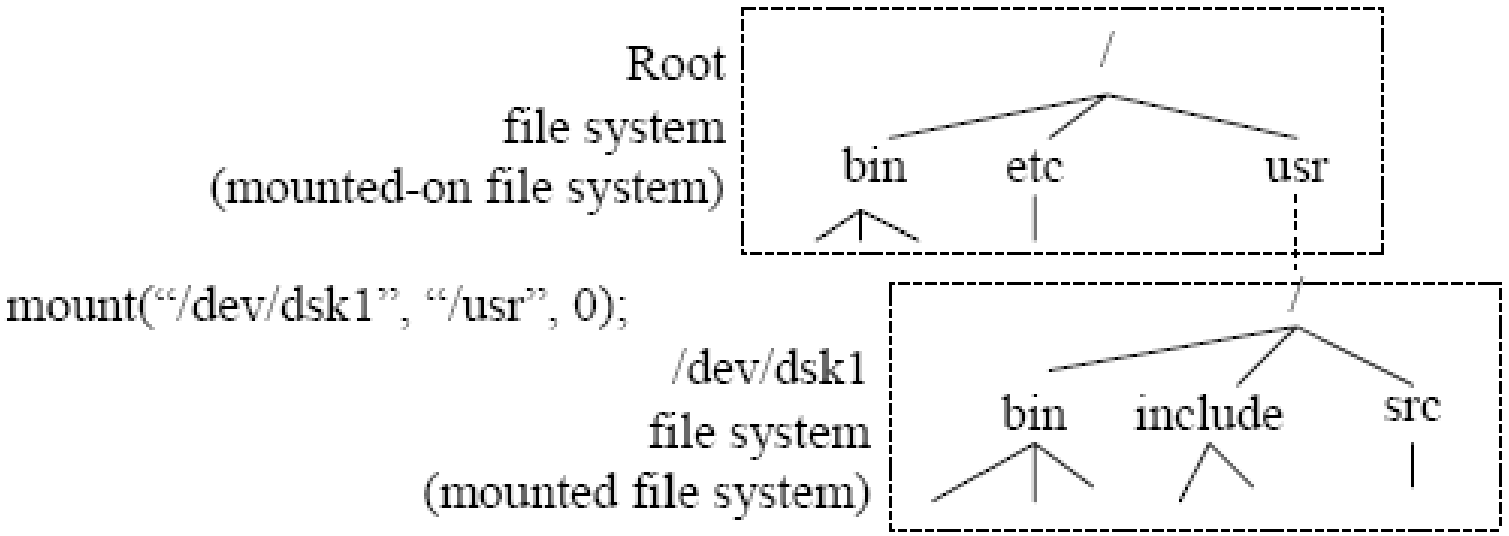
- syntax: newfd = dup(fd);
- The *dup* system call copies a file descriptor into the first free slot of the user file descriptor table, returning the new file descriptor to the user.
- Because *dup* duplicates the file descriptor, it increments the count of the corresponding file table entry, which now has one more file descriptor entry that points to it.
- *dup* serves as an important purpose in building sophisticated programs simpler, building-block programs, as exemplified in the construction of shell pipelines.



1) fd3 = open("/etc/passwd", …);
2) fd4 = open("local", …);
3) fd5 = open("/etc/passwd", …);
4) fd6 = dup(fd3);

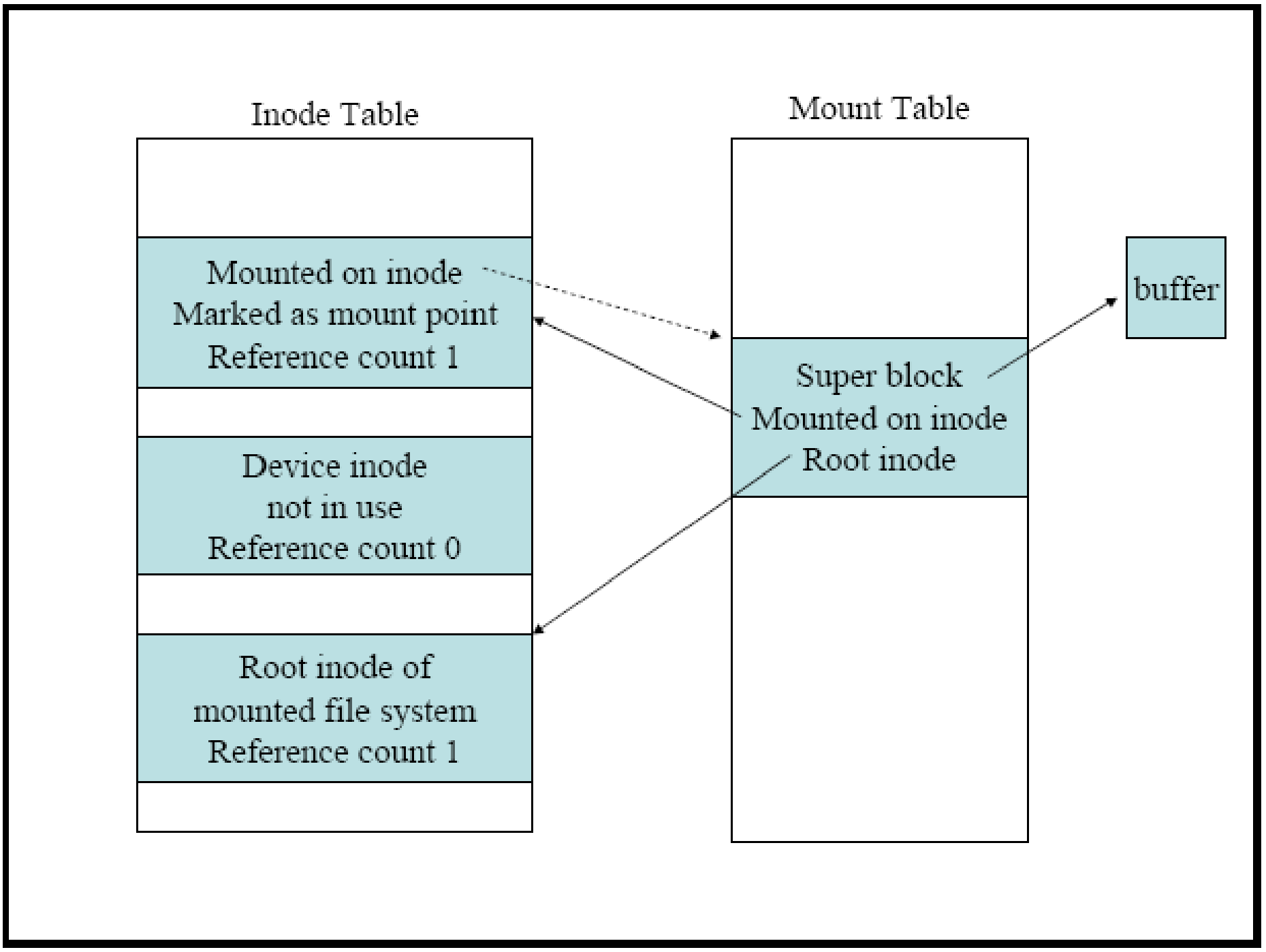
Mount

- syntax: mount(special pathname, directory pathname, options);
- The *mount* system call connects the file system in a specified section of a disk to the existing file system hierarchy; *mount* thus allows users to access data in a disk section as a file system instead of a sequence of disk blocks.



Mount Table

- The kernel has a mount table with entries for every mounted file system; each mount table entry contains
  - a device number that identifies the mounted file system (logical file system number)
  - a pointer to a buffer containing the file system super block
  - a pointer to the root i-node of the mounted file system ("/" of the "/dev/dsk1" file system in the example)
  - a pointer to the i-node of the directory that is the mount point ("usr" of the root file system in the example)
- Association of the mount point i-node and the root i-node of the mounted file system, set up during *mount*, allows the kernel to traverse the file system hierarchy.


Inode Table
Mount Table
Mounted on inode
Marked as mount point
Reference count 1
Device inode
not in use
Reference count 0
Root inode of
mounted file system
Reference count 1
Super block
Mounted on inode
Root inode
buffer

```
algorithm mount
input: file name of block special file
       directory name of mount point
       options
output: none
{
   if (not super user)
      return (error);
   get inode for block special file
         (namei);
   make legality checks;
   get inode for "mounted on"
         directory name (namei);
   if (not directory, or ref count > 1) {
      release inodes (iput);
      return (error);
   }
   find empty slot in mount table;
   invoke block device driver open
         routine;
   get free buffer from buffer cache;
   read super block into free buffer;
   initialize super block fields;
   get root inode of mounted device
         (iget), save in mount table;
   mark inode of "mounted on"
         directory as mount point;
   release special file inode (iput);
   unlock inode of mount point
         directory;
}
```

Algorithm for
Mount a File System

Crossing Mount Point

- Consider the following case:
  - mount /dev/dsk1 /usr
    cd /usr/src/uts (from mounted-on file system to mounted file system)
    cd ../../.. (from mounted file system to mounted-on file system)
- For the case of crossing the mount point from the mounted-on file system to the mounted file system, consider the revised algorithm for *iget*, which checks if the inode is a mount point.
- It finds the mount table entry whose mounted-on i-node is the one just accessed and notes the device number of the mounted file system.
- Using the device number, it then accesses the root i-node of the mounted device and returns that i-node.

```
algorithm iget
input: inode number
output: locked inode
{
   while (not done) {
      if (inode in inode cache) {
         if (inode locked) {
            sleep(event inode becomes
                                  unlocked);
            continue;
         }
         /* special processing for mount
               points */
         if (inode on inode free list)
            remove from free list;
         increment inode reference count;
         return (inode);
      }
      /* inode not in inode cache */
      if (no inodes on free list)
         return (error);
      remove new inode from free
                                  list;
      reset inode umber and file
                                  system;
      remove inode from old hash
          queue, place on new one;
      read inode from disk;
         (algorithm bread)
      initialize inode;
      return (inode);
   }
}
```

algorithm for allocation of in-core i-nodes (AdvOS-03)

```
algorithm iget
input: file system inode number
output: locked inode
{
   while (not done) {
      if (inode in inode cache) {
         if (inode locked) {
            sleep(event inode becomes
                  unlocked);
            continue;
         } /* if (inode locked) */
         /* processing for mount */
         if (inode a mount point) {
            find mount table entry for
                  mount point;
            get new file system number
                  from mount table;
            use root inode number in
                  search;
            continue;  /* loop again */
         }  /* if (mount point) */
         if (inode on inode free list)
            remove from free list;
         increment inode reference count;
         return (inode);
      } /* if (inode in inode cache) */

      /* inode not in inode cache */
      remove new inode from free list;
      reset inode number and file system;
      remove inode from old hash queue,
            place on new one;
      read inode from disk (bread);
      initialize inode (e.g. reference count
            to 1);
      return (inode);
   } /* while (not done);
}
```

Revised Algorithm for Accessing an inode

Crossing Mount Point

- For the second case of crossing the mount point from the mounted file system to the mounted-on file system, consider the revised algorithm for *namei*.
- After finding the i-node number for a path name component in a directory, the kernel checks if the i-node number is the root i-node of a file system.
- If it is, and if the i-node of the current working i-node is also root, and the path name component is dot-dot (“..”), the kernel identifies the i-node as a mount point.
- It finds the mount table entry whose device number equals the device number of the last found i-node, gets the i-node of the mounted-on directory, and continues its search for dot-dot (“..”) using the mounted-on i-node as the working i-node.

```
algorithm namei
input: path name
output: locked inode
{
   if (path name starts from root)
      working inode = root inode;
                  (algorithm iget)
   else
      working inode = current dir inode;
                  (algorithm iget)
   while (there is more path name) {
      read next path name component;
      verify that working inode is of dir,
            access permissions OK;
      if (working inode is of root and
                  component is "..")
         continue;
      read directory by repeated use of
        algorithms bmap, bread, and brelse;
      if (component matches an
            entry in directory
            (working inode)) {
         get inode number for matched
            component;
         release working inode;
            (algorithm iput);
         working inode = inode of
            matched component;
               (algorithm iget)
      } else /* component not in dir */
         return (no inode);
   } /* while */
   return (working inode);
}
```

Conversion of a Path Name to an I-node (AdvOS-03)

```
algorithm namei
input: path name
output: locked inode
{
   if (path name starts from root)
      working inode  = root inode (iget);
   else
      working inode = current directory
            inode (iget);
   while (there is more path name) {
      read next path name component;
      verify that inode is of directory;
      if (inode is of changed root and
            component is ".." )
         continue; /* loop */
component search:
      read inode (dir) (bmap, bread, brelse);
      if (component matches a dir entry) {
         get inode number for matched
               component;
         if (found inode of root and
               working inode is root
               and component is "..") {
            /* crossing mount point */
            get mount table entry for
                  working inode;
            release working inode (iput);
            working inode = mounted on
                  inode;
            lock mounted on inode;
            increment reference count of
                  working inode;
            go to component search
                  (for "..");
         }
         release working inode (iput);
         working inode = inode for new
            inode number (iget);
      } else /* component not in dir */
         return (no inode);
   } /* while */
   return (working inode);
}
```

Unmounting a File System (1/2)

- syntax: umount(special filename);
- Before the kernel actually *unmounts* a file system, it makes sure that no files on that file system are still in use by searching the i-node table.
- Active files have a positive reference count and include files that are the current directory of some process, files with shared text that are currently being executed, and open files that have not been closed.
- The buffer pool may still contain "delayed write" blocks that were not written to disk, so the kernel flushes them from the buffer pool.

Unmounting a File System (2/2)

- The kernel removes shared text entries that are in the region table but not operational, write out all recently modified super blocks to disk, and updates the disk copy of all i-nodes that need updating.
- The kernel then release the root i-node of the mounted file system and invokes the driver of the device that contains the file system to close the device.
- Afterwards, the kernel invalidates buffers for blocks on the now *unmount*ed file system; it moves the buffers to beginning of the buffer free list, so that valid blocks remain in the buffer cache longer.
- The kernel clears the "mounted-on" flag in the mounted-on i-node and release the i-node.

```
algorithm umount
input: special file name of file system
        to be unmounted
output: none
{
   if (not super user)
      return (error);
   get inode for special file (namei);
   extract major, minor number of
        device being unmounted;
   get mount table entry, based on
        major, minor number for
        unmounting file system;
   release inode of special file (iput);
   remove shared text entries from
        region table for files belonging
        to the file system;
   updates super block, inodes, flush
        buffers;
   if (files from file system still in use)
      return (error);
   get root inode of mounted file system
        from mount table;
   lock inode;
   release inode (iput);
   invoke close routine for special device;
   invalidate buffers in pool from
        unmounted file system;
   get inode of mount point from mount
        table;
   lock inode;
   clear flag marking it as mount point;
   release inode (iput);
   free buffer userd for super block;
   free mount table slot;
}
```

Algorithm for
Unmounting a File System

Link

- syntax: link(source file name, target file name);
- The kernel first locates the i-node for the source file by *namei*, increments its link count, update the disk copy of the inode, and unlocks the i-node.
- It then searches for the target file; if the file is present, the link call fails, and the kernel decrements the link count incremented earlier.
- Otherwise, it notes the location of an empty slot in the parent directory of the target file, writes the target file name and the source file i-node number into that slot.
- The link count keeps count of the directory entries that refers to the file and is thus distinct from the i-node reference count.

```
algorithm link
input: existing file name
       new file name
output : none
{
   get inode for existing file name
       (namei);
   if (too many links on file or linking
       directory without super user
       permission) {
      release inode (iput);
      return (error);
   }
   increment link count on inode;
   update disk copy of inode;
   unlock inode;
   get parent inode for directory to
       contain new file name (namei);
   if (new file already exists or existing
       file, new file on different file
       systems) {
      undo update done above;
      return (error);
   }
   create new directory entry in parent
       directory of new file name:
       include new file name, inode
       number of existing file name;
   release parent directory inode (iput);
   release inode of existing file (iput);
}
```

Algorithm for Linking Files

Deadlock in Link - 1

- If the kernel did not unlock the i-node of the source file after incrementing its link count, two processes could deadlock by executing the following system calls simultaneously.
  - process A: link("a/b/c/d", "e/f/g");
  - process B: link("e/f", "a/b/c/d/ee");
- Suppose process A finds the i-node for file "a/b/c/d" at the same time that process B finds the i-node for "e/f".
  - "at the same time" means that the system arrives at a state where each process has allocated its inode.
- When process A new attempts to find the i-node for directory "e/f", it would sleep awaiting the event that the i-node for "f" becomes free; but when process B attempts to find the i-node for directory "a/b/c/d", it would sleep awaiting the event that the inode for "d" becomes free.

Deadlock in Link - 2

- A single process could also deadlock itself.
  - link("a/b/c", "a/b/c/d");
- It would allocate the i-node for file "c" in the first part of the algorithm; if the kernel did not release the i-node lock, it would deadlock when encountering the i-node "c" in searching for the file "d".
- If two processes, or even one process, could not continue executing because of deadlock, what would be the effect on the system?
- Since i-nodes are finitely allocatable resources, recipient of a signal cannot the deadlock without rebooting.