INTERNAL REPRESENTATION OF FILES

- Every file on a UNIX system has a unique inode (contains information necessary for a process to access a file)

File System Algorithms

Lower Level File system Algorithms

<table>
<tr><td colspan="2">namei

iget iput bmap</td><td>alloc free</td><td>ialloc ifree</td></tr>
<tr><td colspan="4"></td></tr>
<tr><td colspan="4">buffer allocation algorithms</td></tr>
<tr><td colspan="4">getblk brelse bread breada bwrite</td></tr>
</table>

Lower Level File System Algorithms

- The algorithm *iget* returns a previously identified inode, possible reading it from disk via the buffer cache and the algorithm *iput* releases this inode. The algorithm *bmap* sets kernel parameter for accessing a file. The algorithm *namei* converts a user-level path name to an inode, using the algorithms iget, iput, and bmap. Algorithms *alloc* and *free* allocate and free disk blocks for files and algorithms *ialloc* and *ifree* assign and free inodes for files.

INODES

- Inodes exist in a static form on disk
- The kernel reads them into an in-core inode to manipulate them.

Disk Inodes

- File Owner identifier
  - Individual owner
  - “Group” owner
  - Set of users who have access rights to a file
- File Type
  - File
    - Regular, directory, character or block special
  - FIFO (pipe)
- File Access permissions
  - To protect by three classes(owner, group, other)
  - Read, write, execute

Disk Inodes (2/2)

- Access times
  - Last modified
- Number of links to the file
- Table of contents for the disk address of data in a file
  - Kernel saves the data in discontiguous disk blocks
  - The Inodes identifies the disk blocks that contain the file's data
- Size
  - Data in a file is addressable by the number of bytes

Disk Inodes – Sample file

Owner mjb

Group os

Type regular file

Perms rwxr-xr-x

Accessed Oct 23 1984 1:45 P.M.

Modified Oct 22 1984 10:30 A.M.

Inode Oct 23 1984 1:30 P.M.

Size 6030 bytes

Disk addresses

- The contents of an inode change when changing the contents of a file or when changing its owner ,permission or link settings.
- Changing the contents of a file automatically implies a change to the inode, but changing the inode does not imply that the contents of the file change.

in-core inode

- Contents in addition to the fields fo the disk inode
  - status of the in-core inode
    - The inode is Locked
    - A Process is waiting for the inode to become unlocked
    - In-core representation of the inode differs from the disk copy as a result of change to the data in the inode.
    - In-core representation of the file differs from the disk copy as a result of change to the file data .
    - The file is a mount point.

in-core inode

- logical device number of the file system that contains the file
- **inode number**
  - Inodes : Its position in the linear array on disk
- pointers to other in-core inodes
- **reference count**
  - Number of instances of the active file

algorithm iget

1. The kernel finds the inode in inode cache and it is on inode free list

   remove from free list

   increment inode reference count

2. The kernel cannot find the inode in inode cache so it allocates a inode from inode free list

   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(algorithm bread)

   initialize inode

Accessing inodes

- The kernel identifies particular inodes by their file system and inode number and allocates in-core inodes at the request of higher level algorithms.
- The algorithm *iget* allocates an in-core copy of an inode. The kernel maps the device number and inode number into a hash queue and searches the queue for the inode.
- If it cannot find the inode, it allocates one from the free list and locks it. The kernel then prepares to read the disk copy of the newly accessed inode into the in-core copy.

- It already knows the inode number and the logical device and computes the logical disk block that contains the inode according to how many disk inodes fit into a disk block. The computation follows the formula:

Block number = ((inode number –1)/ number of inodes per block) + start block of inode list

Example:

Assuming that block 2 is the beginning of the inode list and that there are 8 inodes per block, then inode number 8 is in disk block 2 and

inode number 9 is in disk block 3

Reading *disk inode* into *in-core inode*

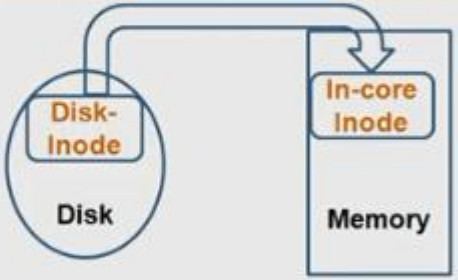
- Find the logical disk block number

Block num = ((inode number -1) / number of inodes per block) + start block of inode list

•Consider,

Start disk block number: 2

number of inodes per block = 8



- inode number **9** is on **disk block 3**
- **Block Num** = ((9-1) / 8) + 2 = (8 / 8) + 2 = 1 + 2 = **3**

Block no 2	Block no 3	Block no 4	So on..
1 to 8	9 to 16	17 to 24	So on...

When kernel knows the device and disk block number, it reads the block using algorithm bread and uses formula to compute the byte offset of the inode in the block:

Byte offset = ((inode number –1) modulo (number of inodes per block))*size of disk inode

Example:

If each disk inode occupies 64 bytes and there are 8 inodes per disk block, then inode number 8 starts at byte offset 448 in the disk block!

Reading *disk inode* into *in-core inode*

Byte offset = ((inode number-1) modulo (number of inodes per block))
* size of disk inode

e.g.

size of disk inode = 64 bytes
number of inodes per disk block =8

Find the byte offset for inode number 8

Byte offset = ((8-1) modulo 8) * 64 = 7 * 64 = **448**

algorithm iget

3. The kernel cannot find the inode in inode cache but finds the free list empty

error

: process have control over the allocation of inodes at user level via execution of open and close system calls and consequently the kernel cannot guarantee when an inode will become available.

4. The kernel finds the inode in inode cache but it was locked

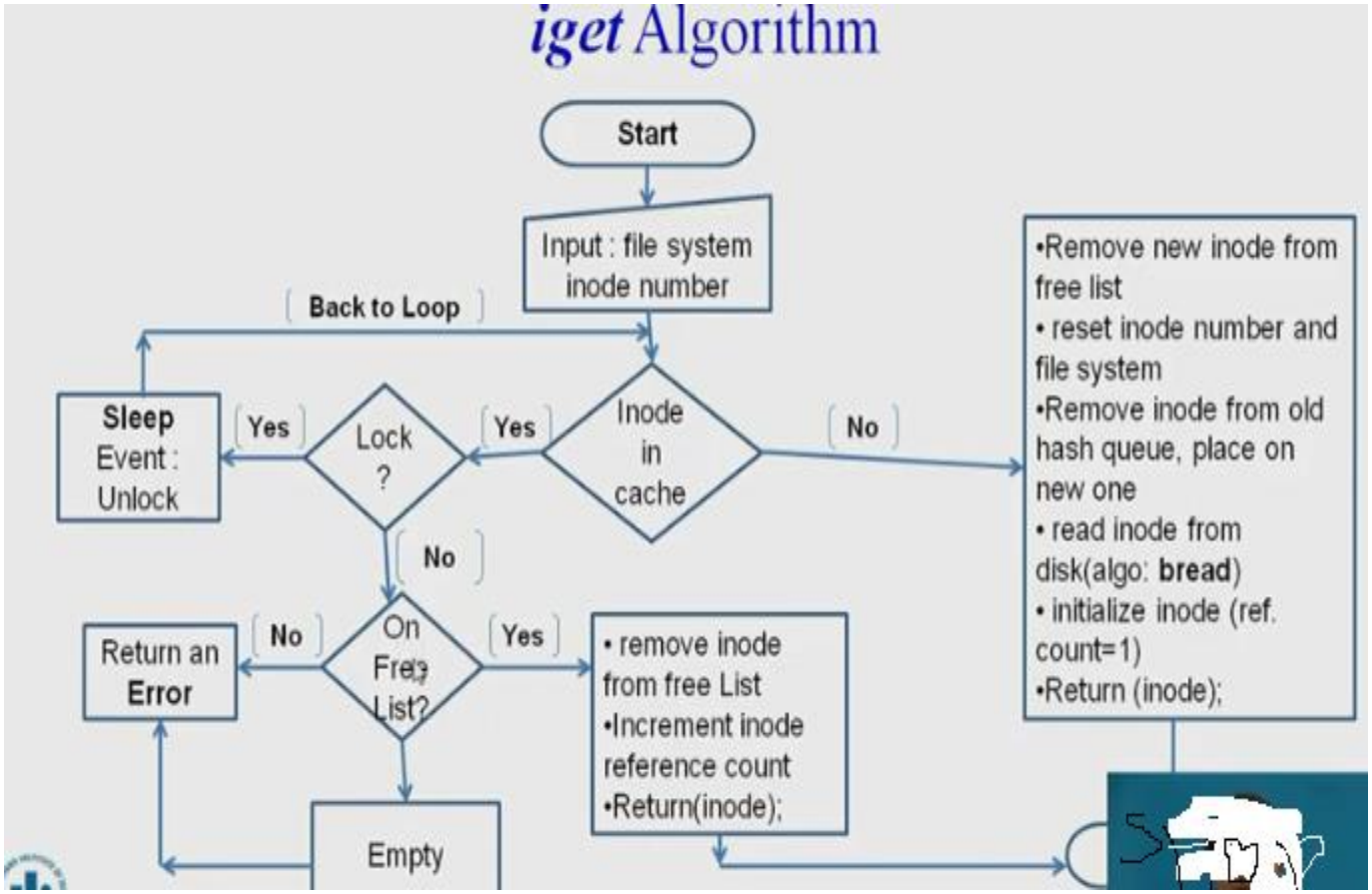
sleep until inode becomes unlocked

Algorithm for allocation of in-core inodes

```
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;           /*loop back to while */
            }
            /*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 number and file system;
        remove inode from old hash queue, place on new one;
        read inode from disk (algorithm bread);
        initialise inode (e.g. reference count to 1);
        return (inode);
}
} /end of main()
```


U
N
I
X


iget Algorithm
Start
Input : file system inode number
Back to Loop
Inode in cache
Yes
Lock ?
Yes
Sleep Event : Unlock
No
•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(algo: bread)
• initialize inode (ref. count=1)
•Return (inode);
No
On Free List?
No
Return an Error
Yes
• remove inode from free List
•Increment inode reference count
•Return(inode);
Empty

Releasing Inodes

- When the kernel releases an inode (algorithm *iput*), it decrements its in-core reference count.
- If the count drops to 0, the kernel writes the inode to disk if the in-core copy differs from the disk copy.
- They differ if the file data has changed, if the file access time has changed or if the file owner or access permissions have changed.
- The kernel places the inode on the free list of inodes, effectively caching the inode in case it is needed again soon.
- The kernel may release all data blocks associated with the file and free the inode if the number of links to the file is 0.

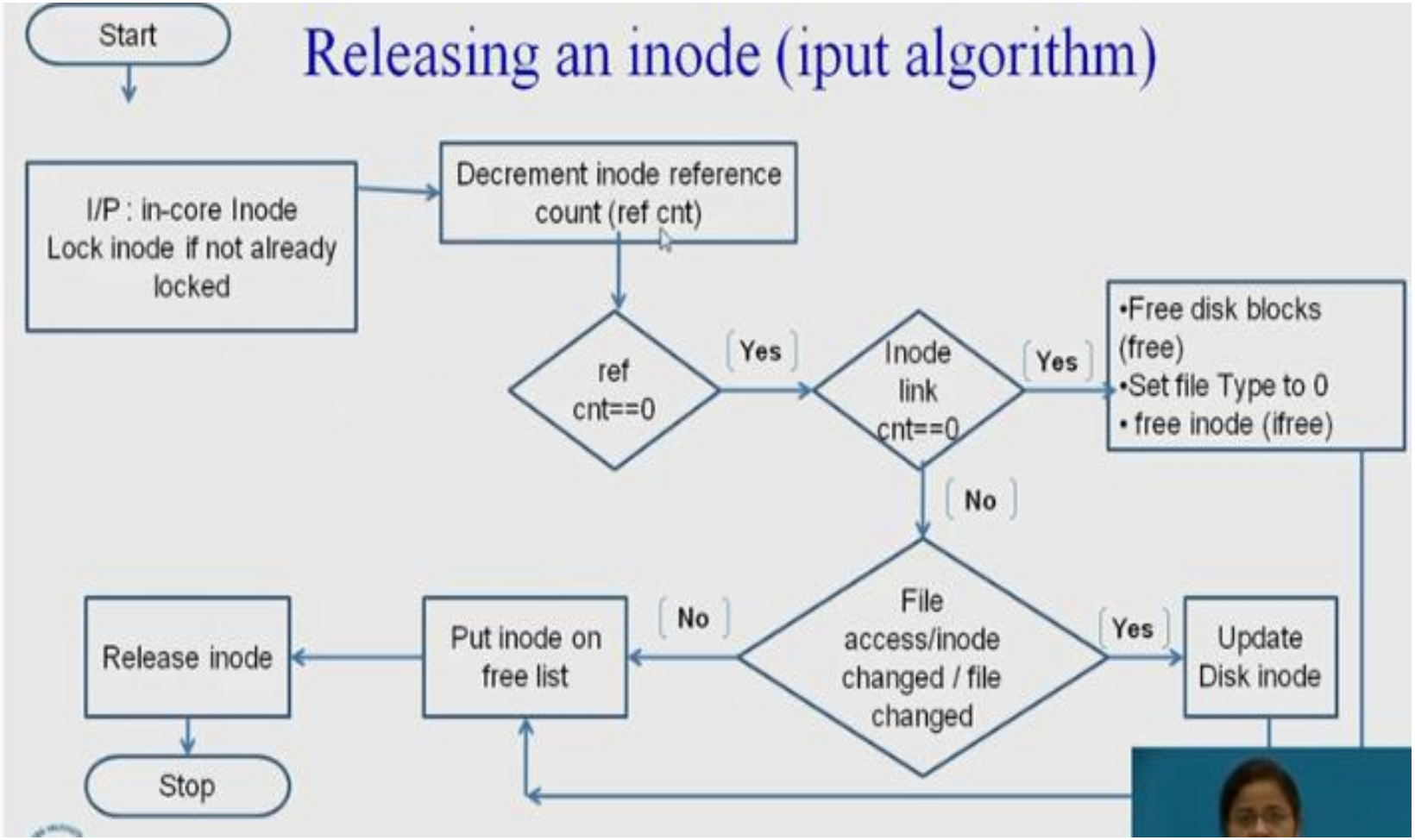
algorithm iput

- - The kernel locks the inode if it has not been already locked
  - The kernel decrements inode reference count
  - The kernel checks if reference count is 0 or not
  - If the reference count is 0 and the number of links to the file is 0, then the kernel releases disk blocks for file(algorithm free), free the inode(algorithm ifree)
    - If the file was accessed or the inode was changed or the file was changed , then the kernel updates the disk inode
    - The kernel puts the inode on free list
  - If the reference count is not 0, the kernel releases the inode lock

iput (inode_no) //releaseIncoreInode

- lock inode if not locked
- decrement inode refernece count
- if (refernce count==0)
  - if (inode link==0)
    - free disk block
    - set file type to 0
    - free inode
  - if (file accessed or inode changed or file changed)
    - update disk inode
  - put inode on free list
- Release inode lock

Releasing an inode (iput algorithm)



Structure of a Regular File

- The inode contains the table of contents to locate a file's data on disk.

Disk block number	Address on Disk

- Since each block on a disk is addressable by number, the table of contents consists of a set of disk block numbers.
- If the data in a file were stored in a contiguous section of the disk (that is, the file occupied a linear sequence of disk blocks), then storing the start block address and the file size in the inode would suffice to access all the data in the file.

Start block addr	File size

Structure of a regular file

However, such an allocation strategy would not allow for simple expansion and contraction of files in the file system without running the risk of fragmenting free storage area on the disk.

Structure of a regular file

…….	File A	File B	File C	…….
Block address	40	50	60	70

…….	File A	Free	File C	File B	…….
Block address	40	50	60	70	85

Allocation of Contiguous Files and Fragmentation of Free Space

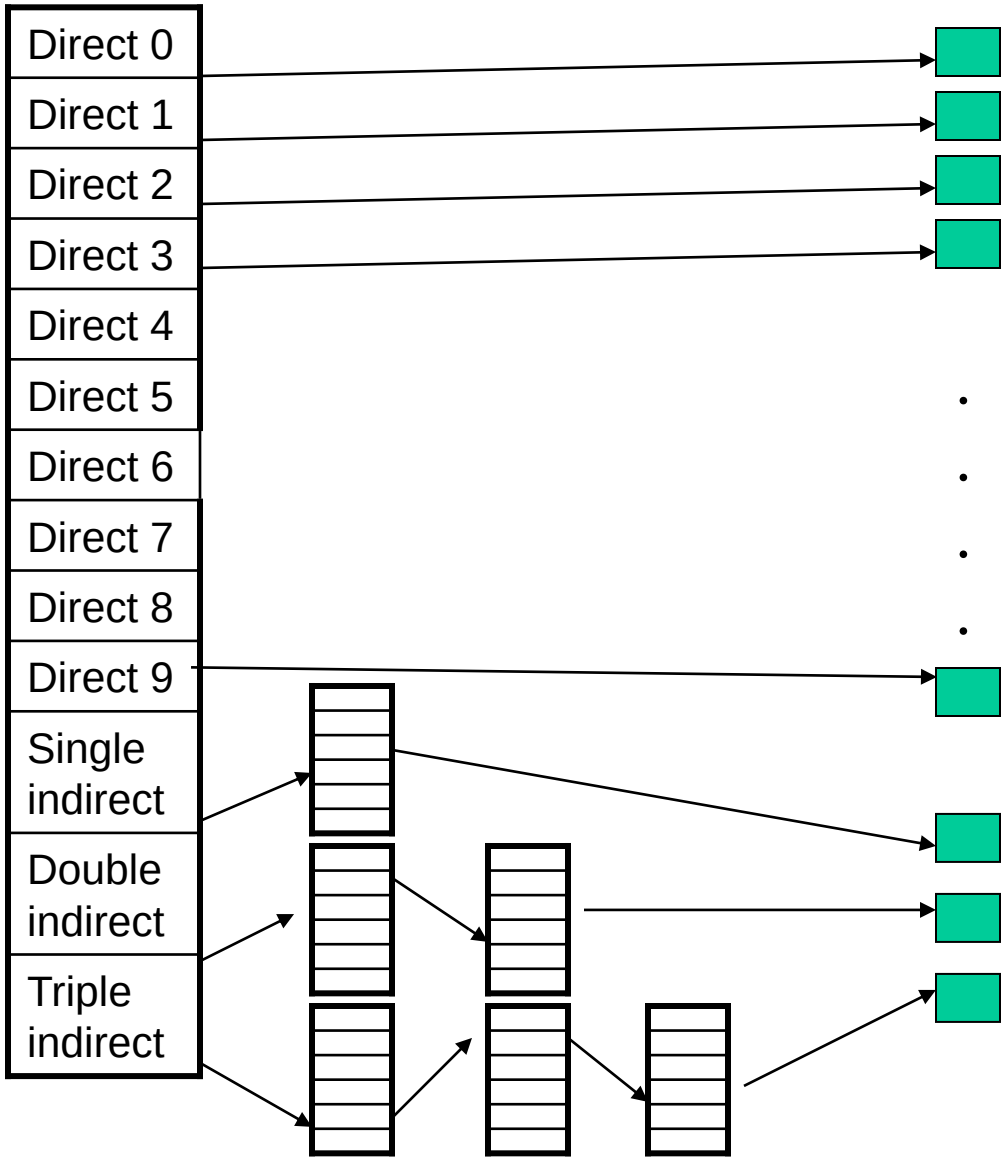
The kernel could minimise fragmentation of storage space by periodically running garbage collection procedures to compact available storage, but that would place an added drain on processing power!!

For greater flexibility, the kernel allocates file space one block at a time and allows the data in a file to be spread throughout the file system! But this allocation scheme complicates the task of locating the data.

To keep the inode structure small yet still allow large files, the table of contents of disk blocks conforms to that shown next→

Direct and Indirect Blocks in Inode

inode



File Allocation (more details about direct-indirect blocks)

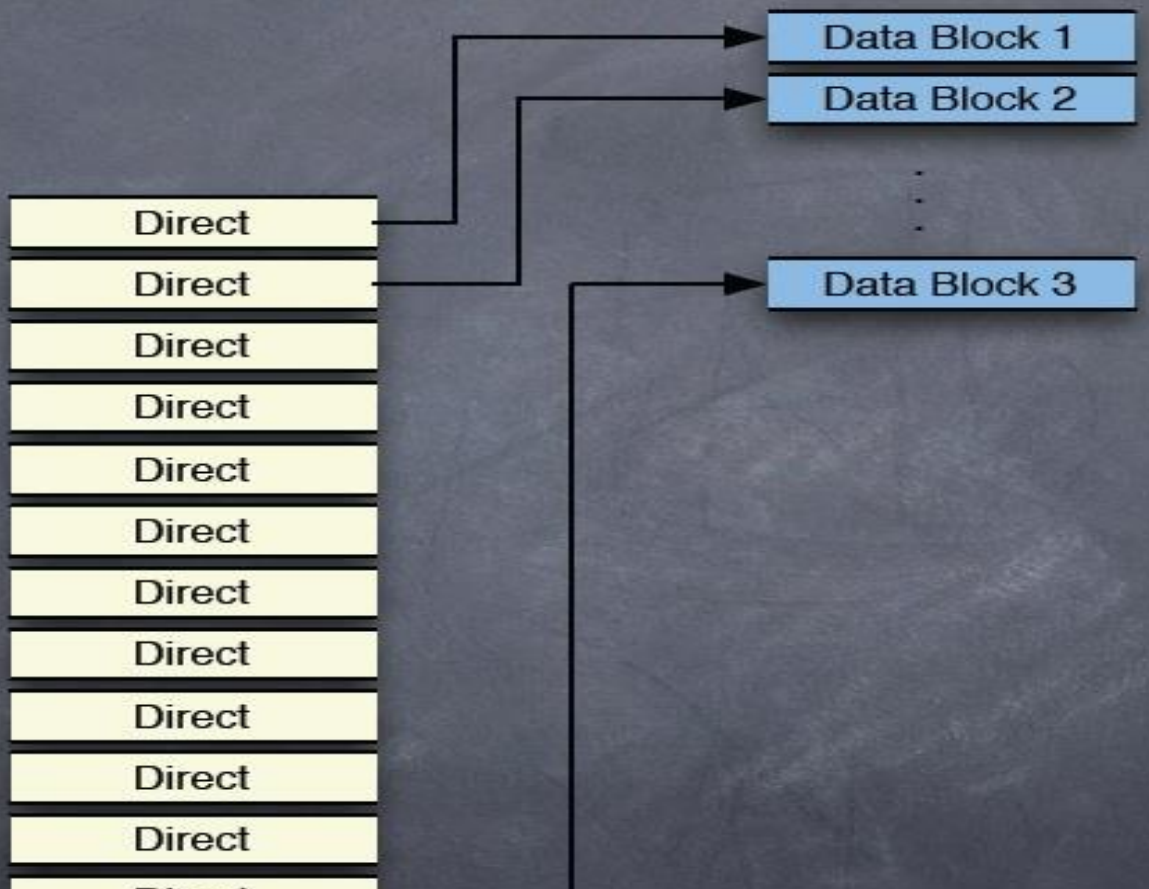
- Files are allocated on a block basis. Allocation is dynamic as needed. Hence, the blocks of a file on disk are not necessarily contiguous.
- An indexed method is used to keep track of each file, with part of the index stored in the inode for the file.
- The first 10 addresses point to the first 10 data blocks of file. If the file is longer than 10 blocks, then one or more levels of information are used as follows:

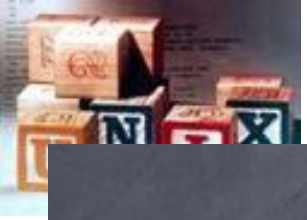
Levels of indirection:

- The eleventh address in the inode points to a block on disk that contains the next portion of the index. This block is referred to as the *single indirect block*. This block contains the pointers to succeeding data blocks in the file.
- If the file contains more blocks, the twelfth address in the inode points to a double indirect block. This block contains a list of addresses of additional single indirect blocks. Each of the single indirect blocks in turn contains pointers to file data blocks.
- If the file contains still more blocks, the thirteenth address in the inode points to a triple indirect block that is a third level of indexing. This block points to additional double indirect blocks.

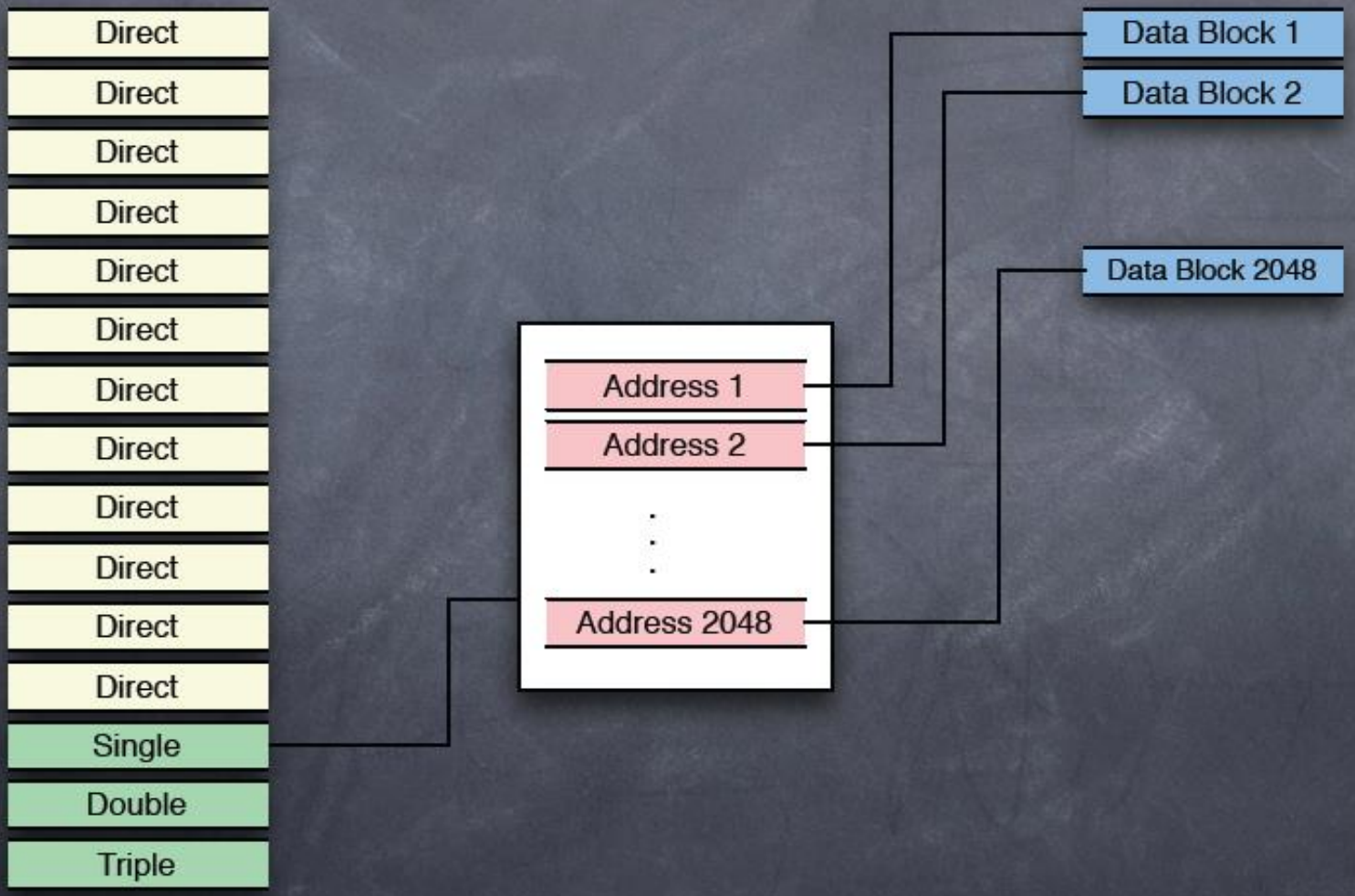

UNIX

Direct


Data Block 1
Data Block 2
Data Block 3
Direct
Direct
Direct
Direct
Direct
Direct
Direct
Direct
Direct
Direct
Direct
Direct
Direct
Single
Double
Triple

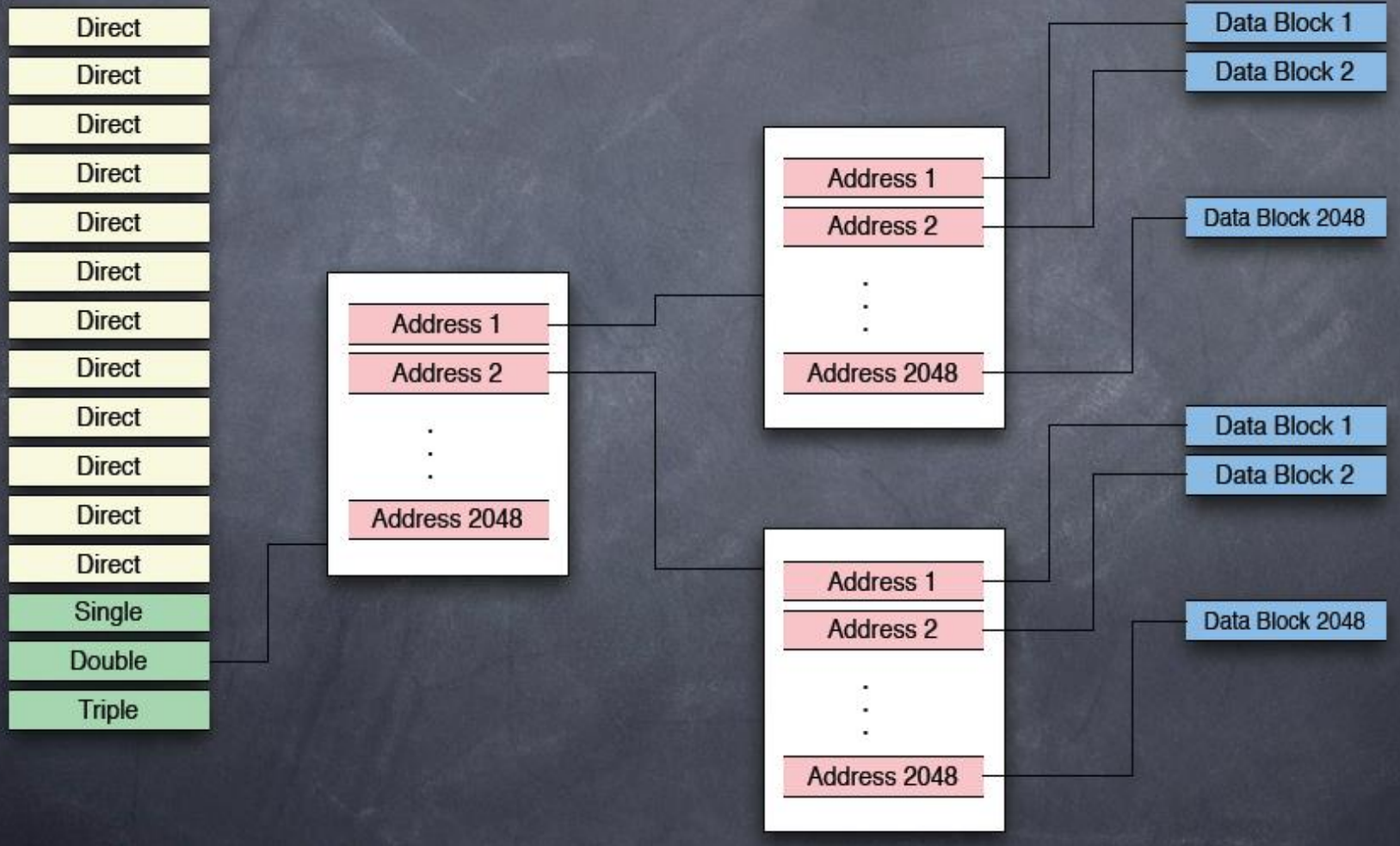


Single

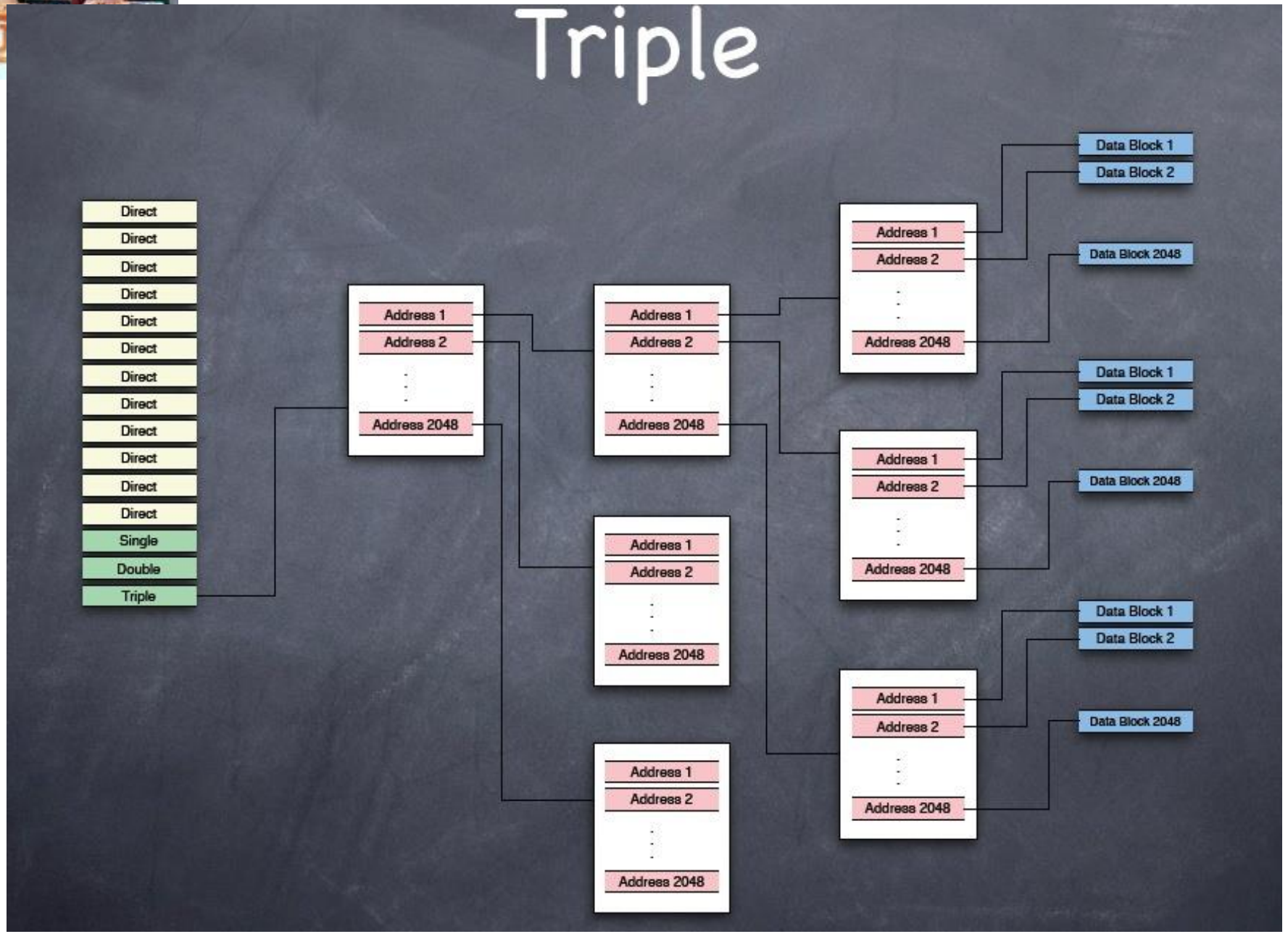

Direct
Direct
Direct
Direct
Direct
Direct
Direct
Direct
Direct
Direct
Direct
Direct
Single
Double
Triple
Address 1
Address 2
Address 2048
Data Block 1
Data Block 2
Data Block 2048

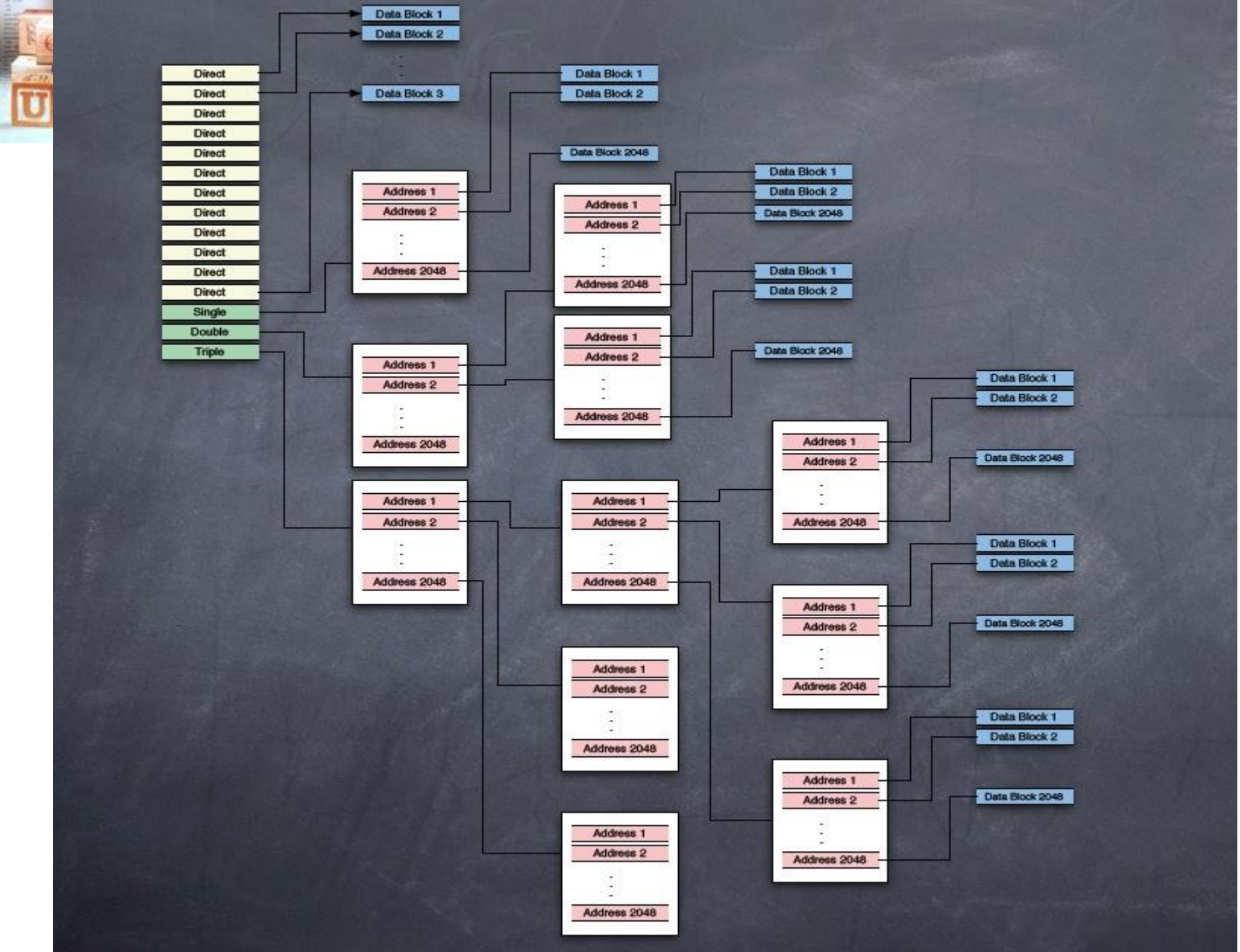

U
N
I
X

Double

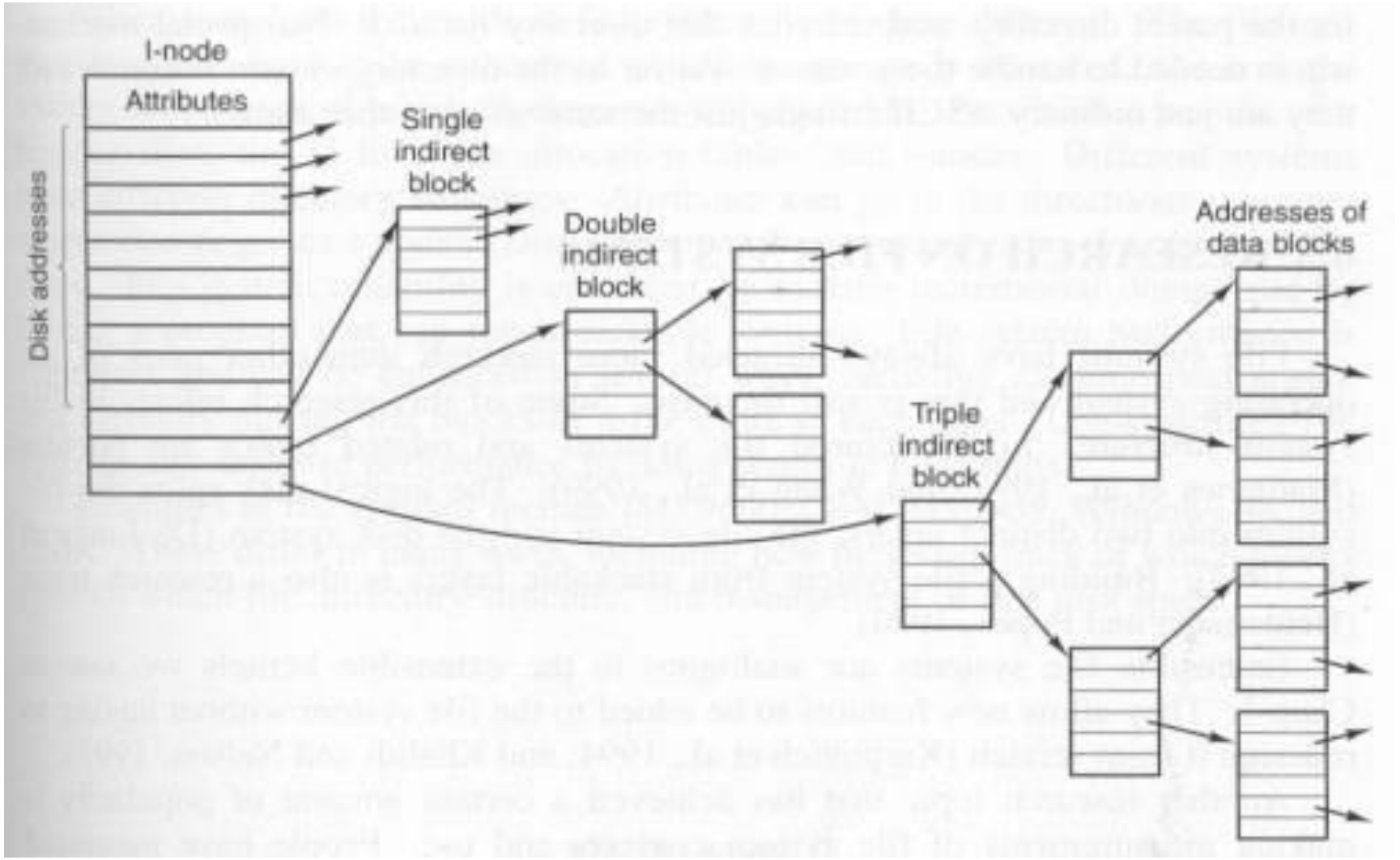

Direct
Direct
Direct
Direct
Direct
Direct
Direct
Direct
Direct
Direct
Direct
Direct
Single
Double
Triple
Address 1
Address 2
Address 2048
Address 1
Address 2
Address 2048
Address 1
Address 2
Address 2048
Data Block 1
Data Block 2
Data Block 2048
Data Block 1
Data Block 2
Data Block 2048

Triple


Direct
Direct
Direct
Direct
Direct
Direct
Direct
Direct
Direct
Direct
Direct
Direct
Direct
Single
Double
Triple
Address 1
Address 2
Address 2048
Address 1
Address 2
Address 2048
Address 1
Address 2
Address 2048
Address 1
Address 2
Address 2048
Address 1
Address 2
Address 2048
Data Block 1
Data Block 2
Data Block 2048
Data Block 1
Data Block 2
Data Block 2048
Data Block 1
Data Block 2
Data Block 2048


Direct
Direct
Direct
Direct
Direct
Direct
Direct
Direct
Direct
Direct
Direct
Single
Double
Triple
Data Block 1
Data Block 2
Data Block 3
Address 1
Address 2
Address 2048
Data Block 2048


U
N
I
X


I-node
Attributes
Disk addresses
Single indirect block
Double indirect block
Triple indirect block
Addresses of data blocks

Structure of a Regular File

- Processes
  - access data in a file by byte offset
  - view a file as a stream of bytes
- The kernel
  - accesses the inode
  - converts the logical file block into the appropriate disk block

Structure of a Regular File – UNIX System V

13 entries in the inode table of contents

- 10 direct, 1 indirect, 1 double indirect, 1 triple indirect block

- Assume
  - a logical block = 1K bytes
  - a block number is addressable by a 32 bit (4 bytes) integer
  - a block can hold up to 256 block numbers

Structure of Regular File

Consider , System V UNIX

Assume that ,

- A logical block on the file system holds 1K bytes
- Block number is addressable by a 32 bit integer, then
- A block can hold up to 256 block numbers, then maximum number of bytes

10 direct blocks with 1K bytes each=10K bytes

1 indirect block with 256 direct blocks= 1K*256=256K bytes

1 double indirect block with 256 indirect blocks=256K*256=64M bytes

1 triple indirect block with 256 double indirect blocks=64M*256=16G bytes

Byte Capacity of a File – 1K Bytes Per Block

- 10 direct blocks with 1K bytes each = 10K bytes
- 1 indirect block with 256 direct blocks = 256K bytes
- 1 double indirect block with 256 indirect blocks = 64M bytes
- 1 triple indirect block with 256 double indirect blocks = 16G bytes

Direct and Indirect Blocks in Inode – UNIX System V

- Assume that a logical block on the file system holds 1K bytes and that a block number is addressable by a 32 bit integer, then a block can hold up to 256 block numbers.

- Assume that a disk block contains 1024 bytes. If a process wants to access byte offset 9000,
- the kernel calculates that the byte is in which direct block and starting byte offset ?

1 disk block = 1024 bytes

- If it wants the 350000 in the file?

Block Layout of a Sample File and its Inode

4096
228
45423
0
0
11111
0
101
367
0
428
9156
824

0 : meaning that the logical block entries contain no data.
If no process ever wrote data into the file at any byte offsets corresponding to those blocks

808th

367 Data block

816th

3333

Data block

0

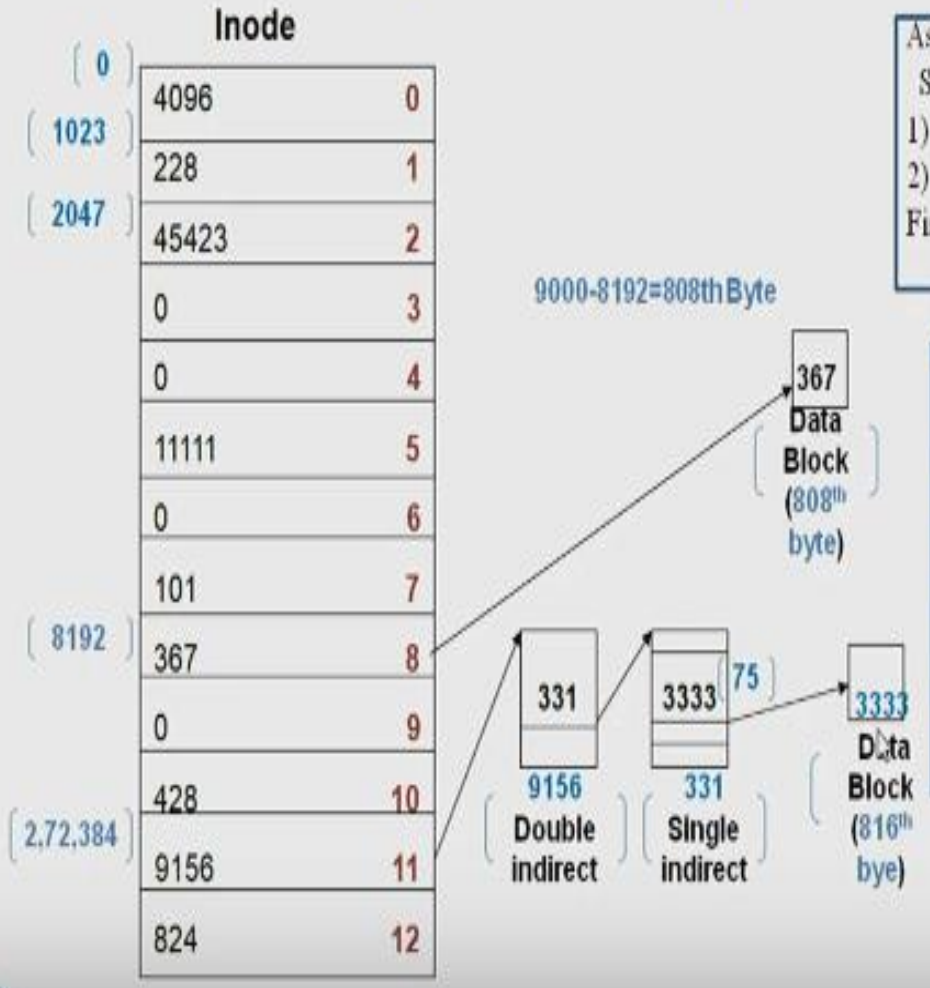
75 3333

331 Single indirect

0 331

9156 Double indirect

Block Layout of Sample File and its Inode



Assume that,

Size of Disk Block = 1024 Bytes

1) Process wants to access byte offset 9000,
2) Process wants to access byte offset 3,50,000

Find how process access its data block?

1. Each block has capacity = 256K
There are , Total 10 blocks
So, 256 + 10 = 266 blocks
Starting at, 266 * 1024 = 2,72,384
2. Byte offset: 3, 50,000 – 2,72,384 = 77,616 of Double indirect which has direct block no 331
3. So, block no of Single indirect block= 77,616/1024 = 75 which point to data block 3333
4. As, 75 * 1024 = 76,800
Now, 77,616 – 76800 = 816th byte in Data block 3333

Assume that,

Size of Disk Block = 1024 Bytes

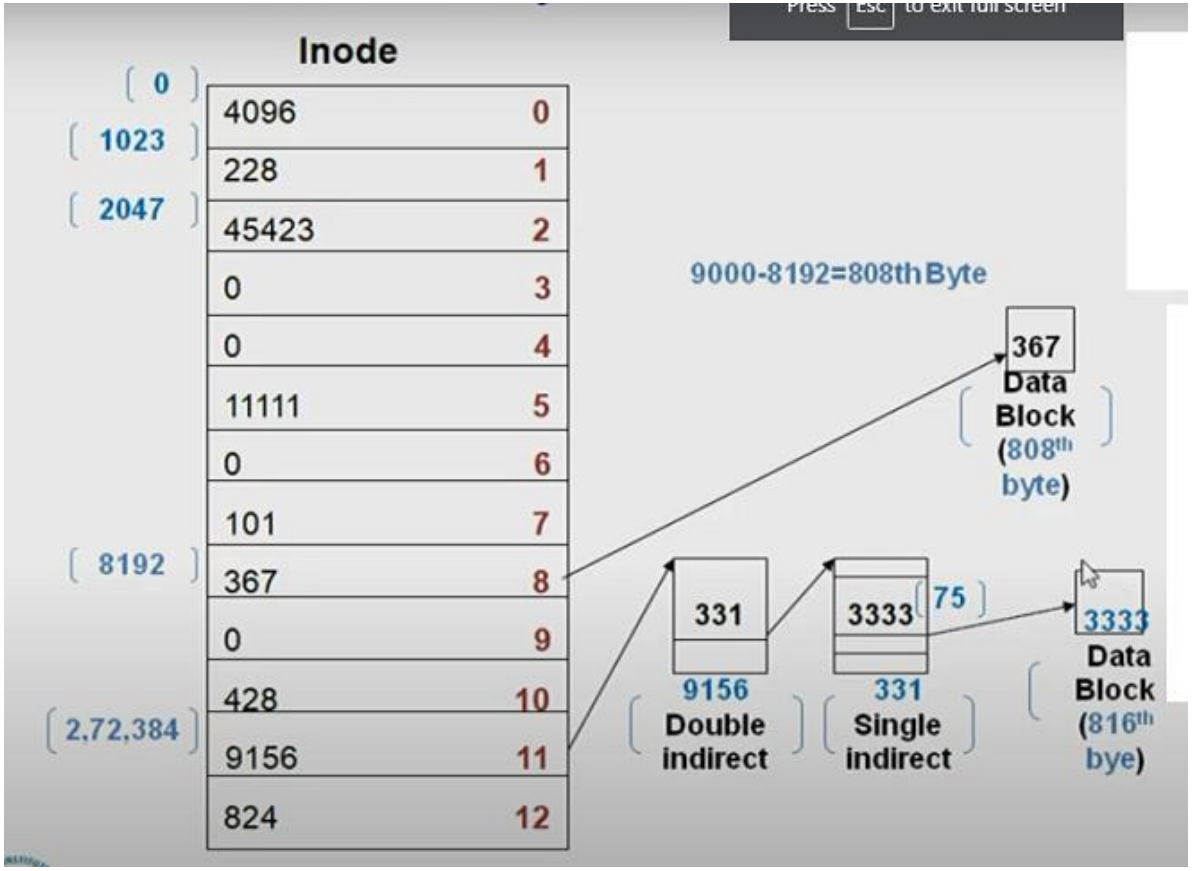
1) Process wants to access byte offset 9000,

2) Process wants to access byte offset 3,50,000

Find how process access its data block?

1. Each block has capacity = 256K
There are , Total 10 blocks
So, 256 + 10 = 266 blocks
Starting at, 266 * 1024 = 2,72,384
2. Byte offset: 3, 50,000 – 2,72,384 = 77,616 of Double indirect which has direct block no 331
3. So, block no of Single indirect block= 77,616/1024 = 75 which point to data block 3333
4. As, 75 * 1024 = 76,800
Now, 77,616 – 76800 = 816th byte in Data block 3333


U
N
I
X


Press Esc to exit full screen
Inode
0
1023
2047
8192
2,72,384
4096 0
228 1
45423 2
0 3
0 4
11111 5
0 6
101 7
367 8
0 9
428 10
9156 11
824 12
9000-8192=808th Byte
367
Data Block (808th byte)
331
9156
Double indirect
3333
75
331
Single indirect
3333
Data Block (816th bye)

Advantages of this scheme:

- The inode is of fixed size and relatively small and hence may be kept in main memory for long periods
- Smaller files may be accessed with little or no indirection, reducing processing and disk access time
- The theoretical maximum size of a file is large enough to satisfy virtually all applications!

Algorithm bmap

Input: (1) inode

(2) byte offset

Output : (1) block number in file system

(2) byte offset into block

(3) bytes of I/O in block

(4) read ahead block number

{

calculate logical block number in file from byte offset;

calculate start byte in block for I/O;

calculate number of bytes to copy to user;

check if read-ahead applicable, mark inode;

determine level of indirection;


```
while (not at necessary level of indirection)
    {
        calculate index into inode or indirect block from logical block
    number in file;
        get disk block number from inode or indirect block;
        release buffer from previous disk read, if any (algorithm brelse);
        if ( no more levels of indirection )
            return (block number);
        read indirect disk block ( algorithm bread);
        adjust logical block number in file according to level of
    indirection;
    }
}
```

Directories (1/2)

- A directory is a file
  - Its data is a sequence of entries
  - Contents of each entries
    - an inode number and the name of a file
- Path name is a null terminated character string divided by slash (“/”)
- UNIX System V
  - Maximum of component name : 14 characters
  - Inode # : 2 bytes
  - Size of a directory : 16 bytes

Directories (2/2)

- Directory layout for /etc

	Byte Offset in Directory	Inode Number	File Name
	0	83	.
	16	2	..
	32	1798	init
	48	1276	fsck
. Current directory	…	…	…
	224	0	crash
. . Parent directory	…	…	…
	256	188	inittab

Directory Layout for /etc

Byte Offset in Directory	Inode Number (2 bytes)	File Names
0	83	.
16	2	..
32	1798	init
48	1276	fsck
64	85	clri
80	1268	motd
96	1799	mount
112	88	mknod
128	2114	passwd
144	1717	umount
160	1851	ckecklist
176	92	fsdb1b
192	84	config
208	1432	getty
224	0	crash
240	95	mkfs
256	188	inittab

Every directory contains the file names dot and dot-dot whose inode numbers are those of the directory and its parent directory, respectively. The number of “.” in /etc is allocated at offset 0, and its value is 83. Directory entries may be empty, indicated by an inode number 0. For instance, the entry at address 224 in /etc is empty, although it once contained an entry for a file named “crash”.

Conversion of a Path Name to an Inode

- The initial access to a file is by its path name
  - Open, chdir, link system calls
- The kernel works internally with inodes rather than with path name
  - Converting the path names to inodes
  - Using *algorithm namei*
    - parse the path name one component at a time
    - convert each component into an inode
    - finally return the inode of the input path name

CONVERSION OF PATHNAME TO AN INODE

- Initial access to a file is through its pathname. The kernel needs to translate a pathname to inodes to access files.
- The algorithm *namei* parses the pathname one component at a time, converting each component into an inode based on its name and the directory being searched and eventually returns the inode of the input path name.

Namei ALGORITHM

- if pathname is absolute, then search starts from the root inode
- if pathname is relative, search is started from the inode corresponding to the current working directory of the process.(kept in the process *u area*)
- the components of the pathname are then processed from left to right. Every component, except the last one, should either be a directory or a symbolic link. **Let's call the intermediate inodes the working inodes.**

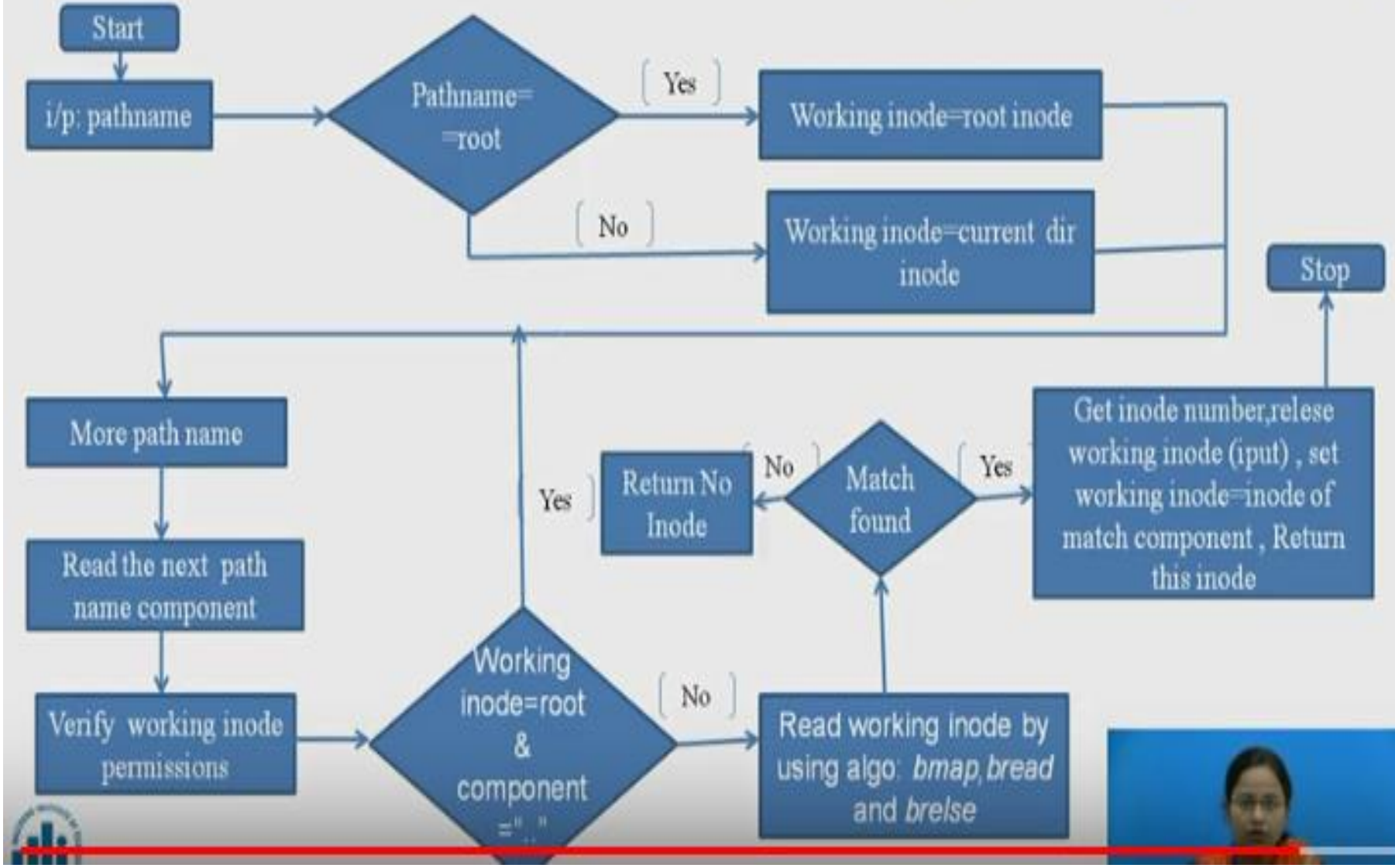
Namei algorithm (cont’)

– If the working inode is a directory, the current pathname component is looked for in the directory corresponding to the working inode. If it is not found, it returns an error, otherwise, the value of the working inode number becomes the inode number associated with the located pathname component.

Namei (cont’)

- If the working inode corresponds to a symbolic link, the pathname up to and including the current path component is replaced by the contents of the symbolic link, and the pathname is reprocessed.
- The inode corresponding to the final pathname component is the inode of the file referenced by the entire pathname

Conversion of Pathname to an inode (*namei* algorithm)



Algorithm namei /* convert path-name to inode */

Input : path name

Output: locked inode

```
{
    if (path name starts from root)
            working inode = root inode (algorithm iget);
    else
        working inode = current directory inode ( algorithm iget);
```


```
while (there is more path name)
    {
        read next path name component from input;
        verify that working inode is of directory, access permissions OK;
    if (working inode is of root and component is "..")
                continue;          /* loop back to while */
    read directory ( working inode ) 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
     return ( no inode);
}
   return (working inode);
}//end of main
```

Sample-namei(/etc/passwd)

1. Encounters “/” and gets the system root inode
2. Current working inode = root
3. Permission check
4. Search root for a file – “etc”
    1. Access data in the root directory block by block
    2. Search each block one entry-”etc”
5. Finding
    1. Release the inode for root(iput)
    2. Allocate the inode for etc(iget) by inode # found
6. Permission check for “etc”

7. Search “etc” block by block for a directory structure entry for “passwd”
8. Finding
    1. Relase the inode for “etc”
    2. Allocate the inode for “passwd”
9. Returns that inode

CONVERSION OF A PATH NAME TO AN INODE

- Example(to open the file"/etc/passwd")

Root inode

Current working inode
(gather string etc)

Check current working inode ➔ process has permission to search

etc

Release root inode
Allocation inode for "etc"

Check current working inode ➔ process has permission to search

passwd

Release "etc" inode
Allocation inode for "passwd"
(It is the ninth entry of the directory in Figurre)

Super block

- File System

boot block	super block	inode list	data blocks

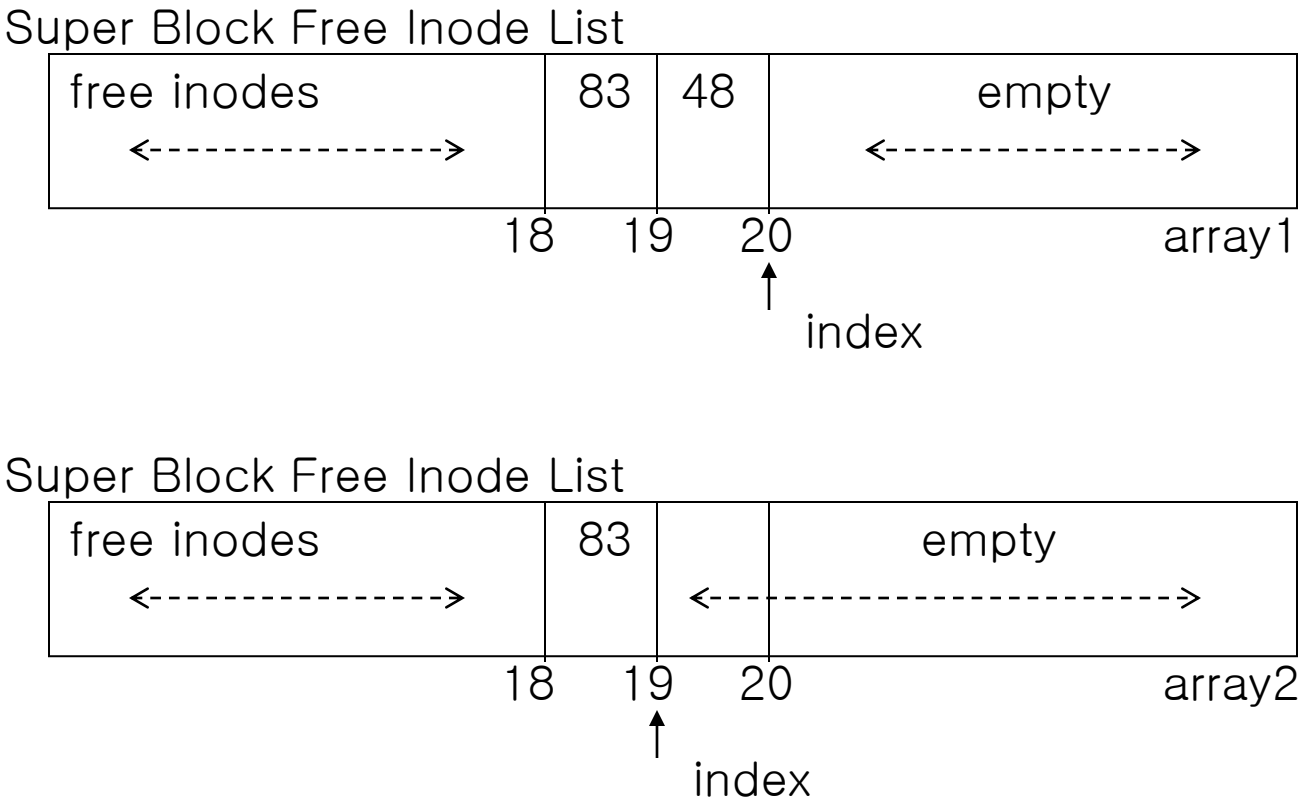
- consists of

- the size of the file system
- the number of free blocks in the file system
- a list of free blocks available on the file system
- the index of the next free block in the free block list
- the size of the inode list
- the number of free inodes in the file system
- a list of free inodes in the file system
- the index of the next free inode in the free inode list
- lock fields for the free block and free inode lists
- a flag indicating that the super block has been modified

Inode Assignment to A New File

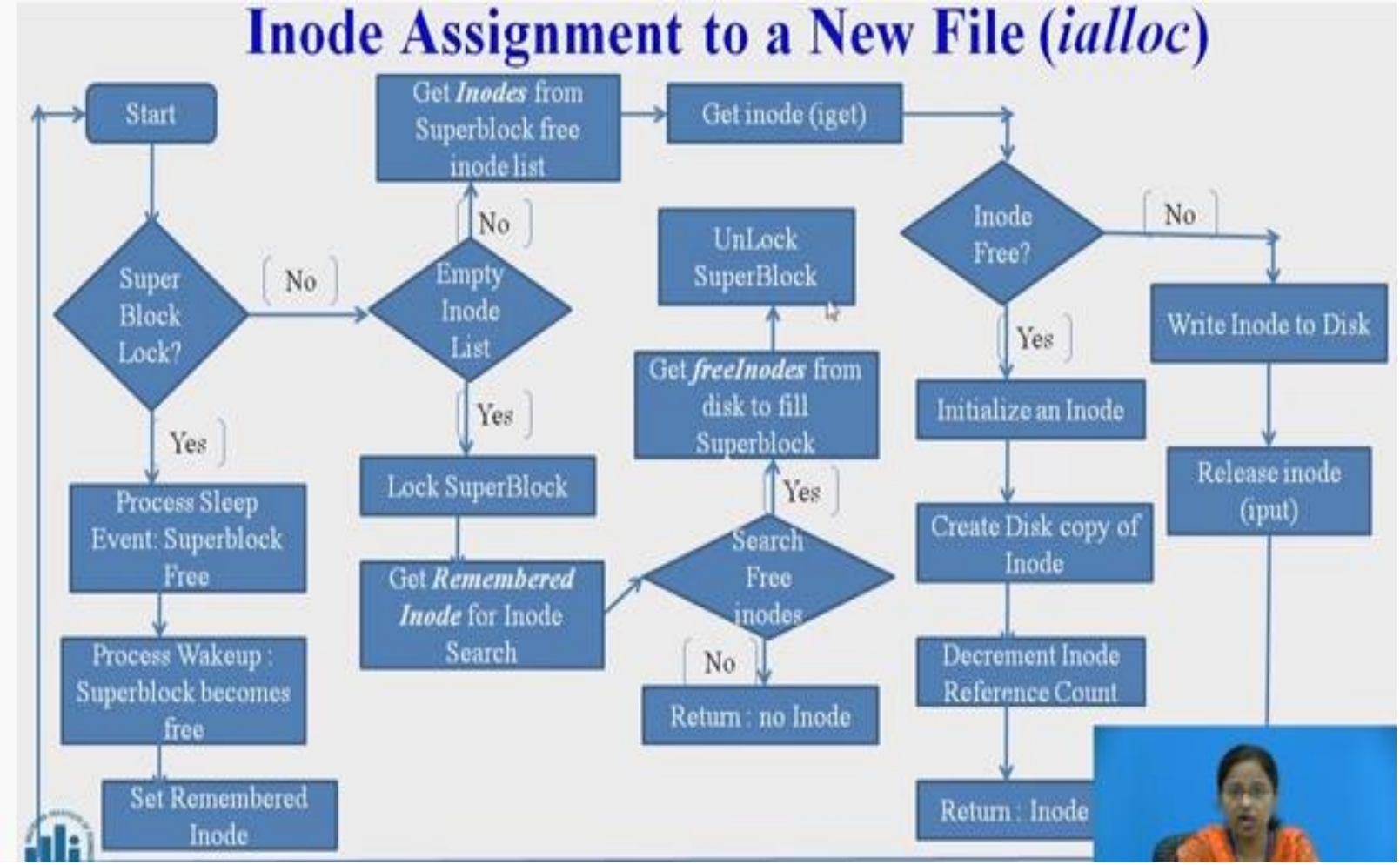
- a known inode
  - Algorithm iget : to allocate an known inode
  - Algorithm namei : to determine inode No. by matching pathname component to a name in directory
- Algorithm ialloc
  - To assign a disk inode to a newly created file

Inode assignment to a new file



Assigning Free Inode from Middle of List

Inode Assignment to a New File (*ialloc*)



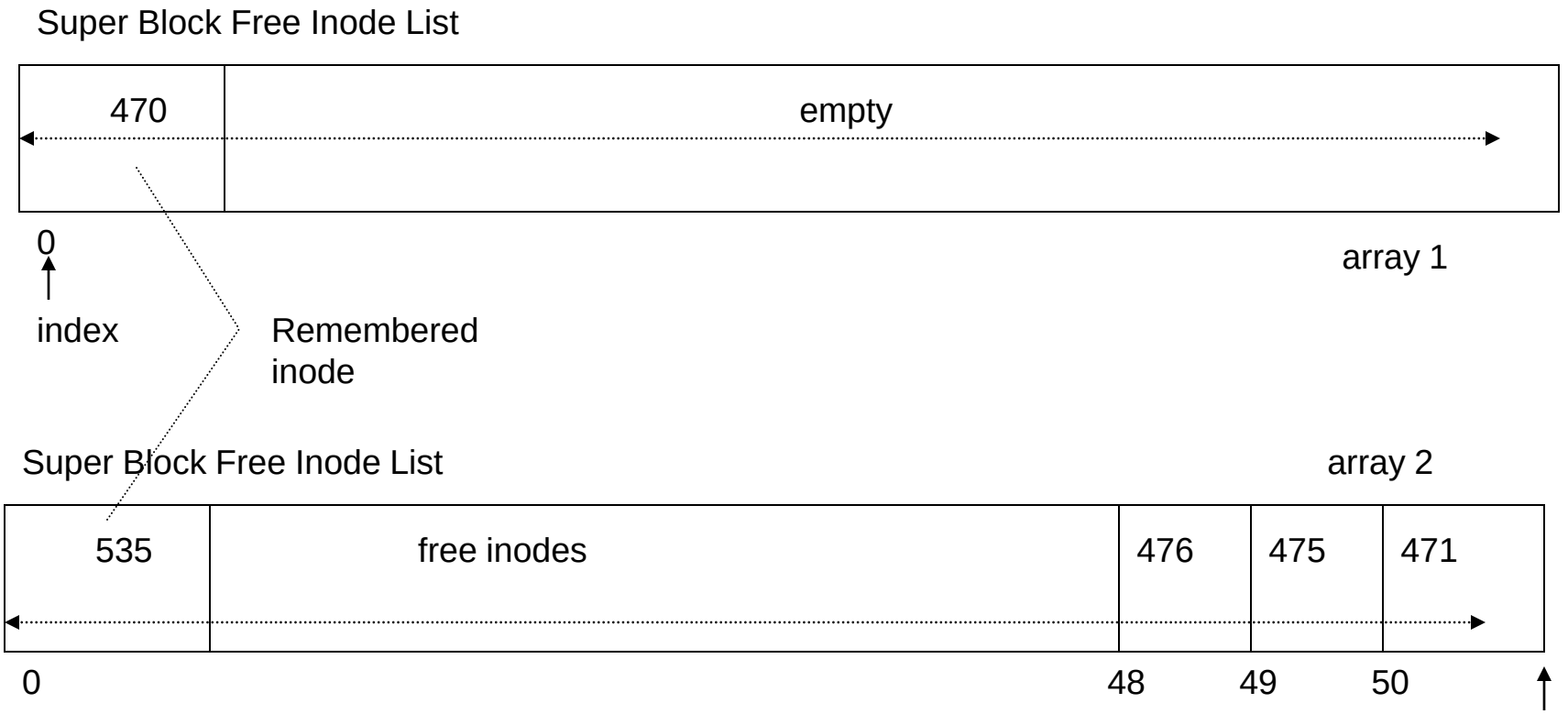
ialloc

- Algorithm *ialloc* assigns a disk inode to a newly created file
- The file system contains a linear list of inodes. An inode is free if its type field is zero. When a process needs a new inode, the kernel could theoretically search the inode list for a free inode
- Such a search would be expensive, requiring at least one read operation for every inode! To improve performance, the file system super block contains an array to cache the numbers of free inodes in the file system
- Super block contains an array
  - To improve performance of searching a free inode
  - To cache the number of free inodes

Remembered inodes:

- If the super block list of free inodes is empty,
  - the kernel searches the disk and places as many free inode numbers as possible into the super block.
  - The kernel reads the inode list on disk, block by block, and fills the super block list of inode numbers to capacity, remembering the highest-numbered inode that it finds.
  - Call that inode the **“remembered”inode; it is the last one saved in the super block.**
  - The next time the kernel searches the disk for free inodes, it uses the remembered inode as its starting point, thereby assuring that it wastes no time reading disk blocks where no free inodes should exist.

Assigning Free Inode – Super Block List Empty



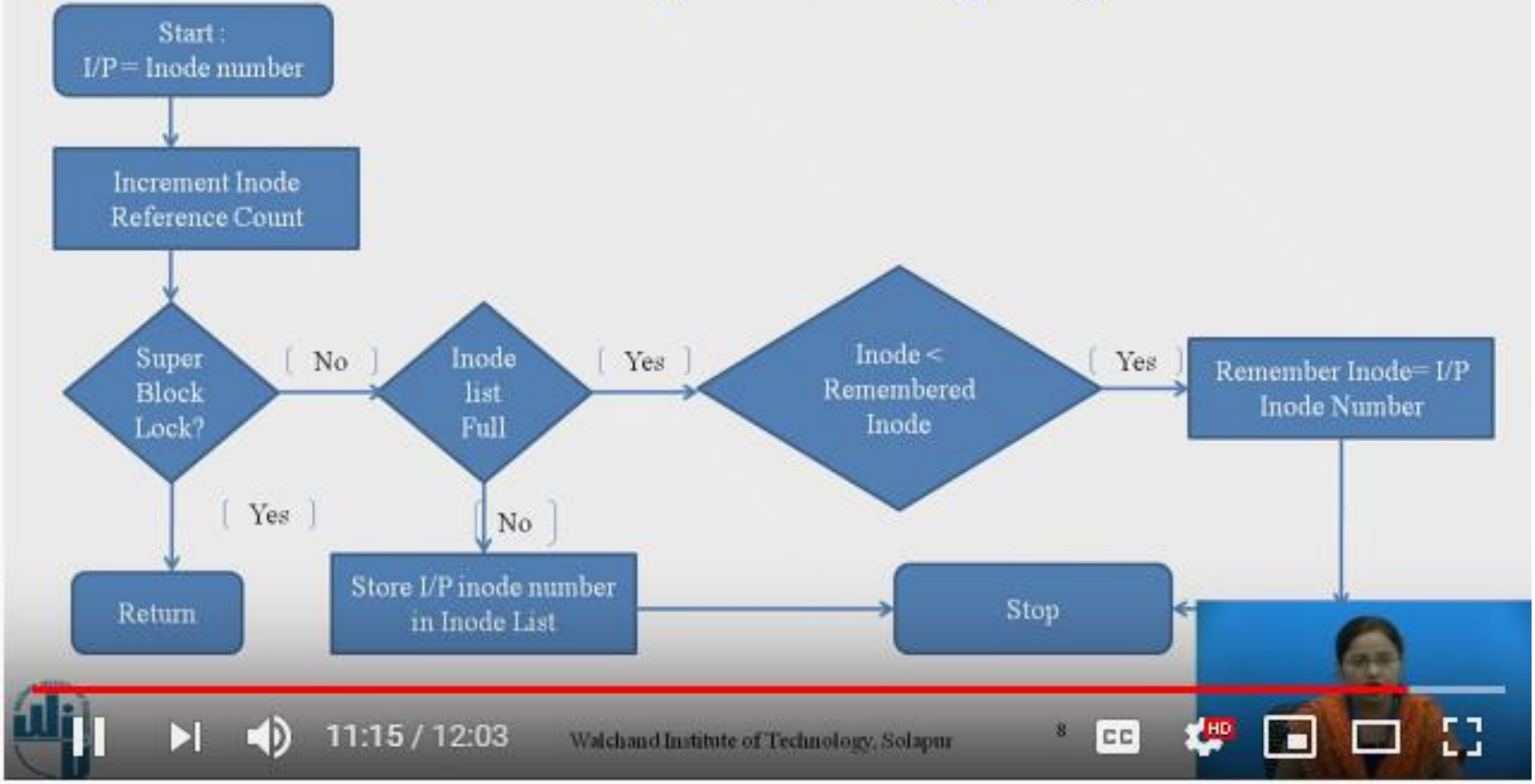
If the list of free inodes in the super block is like the first one above, it will notice that the array is empty and search the disk for free inodes, starting from inode number 470, the remembered inode. When the kernel fills the super block free list to capacity, it remembers the last inode as the start point for the next search of the disk. The kernel assigns an inode it just took from the disk and continues whatever it was doing.


```
Algorithm ialloc        /* allocate inode */
Input   : file system
Output : locked inode
{
    while (not done)
    {
        if (super block locked)
        {
            sleep (event super block become free);
            continue;                   /* while loop */
        }
```


```
if (inode list in super block is empty)
        {
                lock super block;
                get remembered inode for free inode search;
        search disk for free inodes until super block full,
                or no more free inodes (algorithms bread and
    brelse);
                unlock super block;
                wake up (event super block becomes free);
                if (no free inodes found on disk)
                        return (no inode);
        set remembered inode for next free inode search;
        }
```


```
        /* there are inodes in super block inode list */
        get inode numbers in super block inode list;
        get inode (algorithm iget);
        if (inode not free after all)      /* !!! */
        {
                write inode to disk;
                release inode (algorithm iput);
                continue;        /* while loop */
        }
        /* inode is free */
        initialize inode;
        write inode to disk;
        decrement file system free inode count;
        return (inode);
    }
}
```

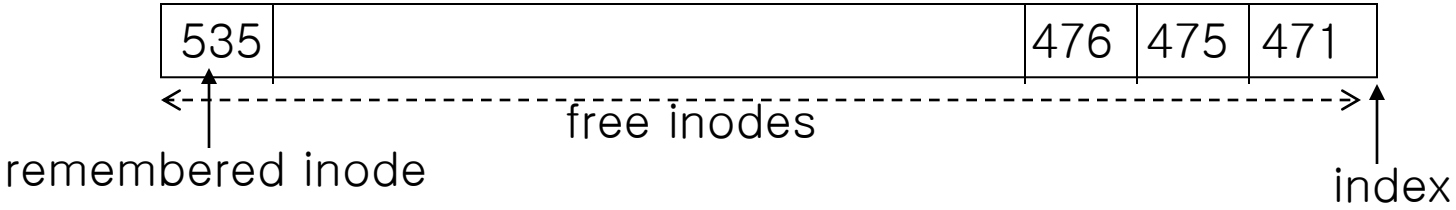
Releasing an Inode (*ifree*)



ialloc - ifree

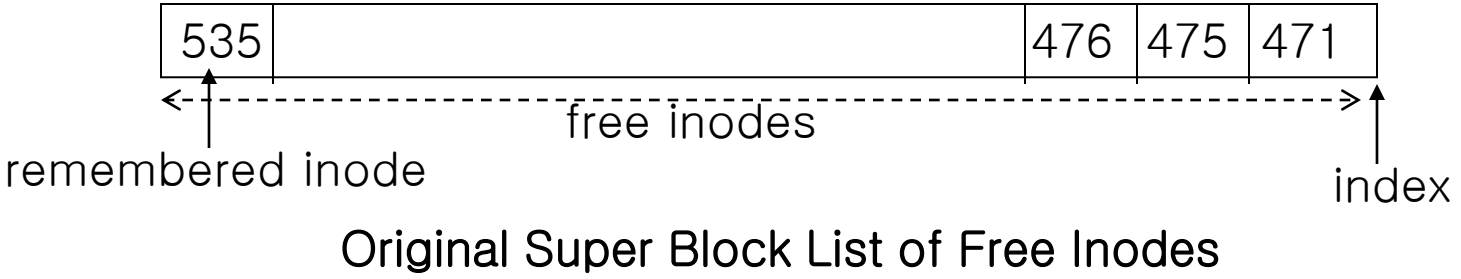
- ifree: freeing an inode

Inode assignment to a new file

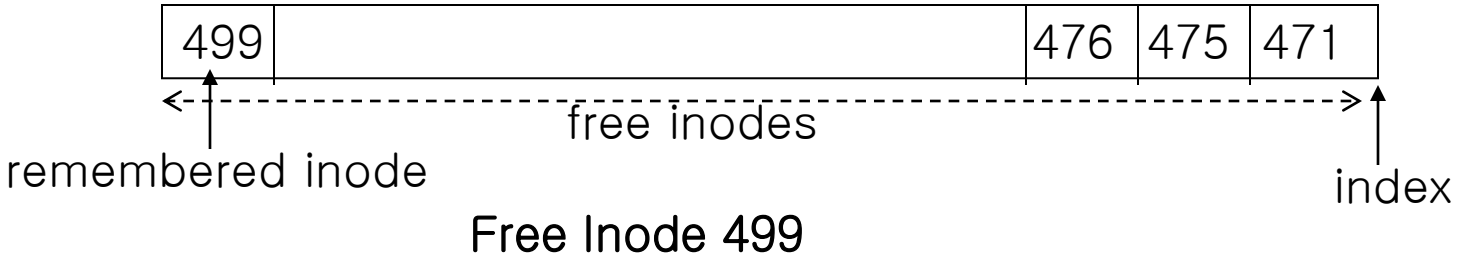


Original Super Block List of Free Inodes

Inode assignment to a new file

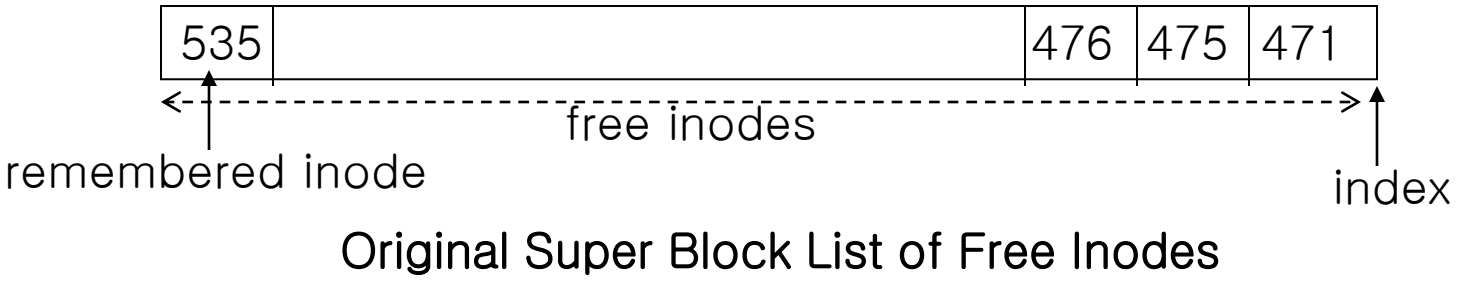


Original Super Block List of Free Inodes

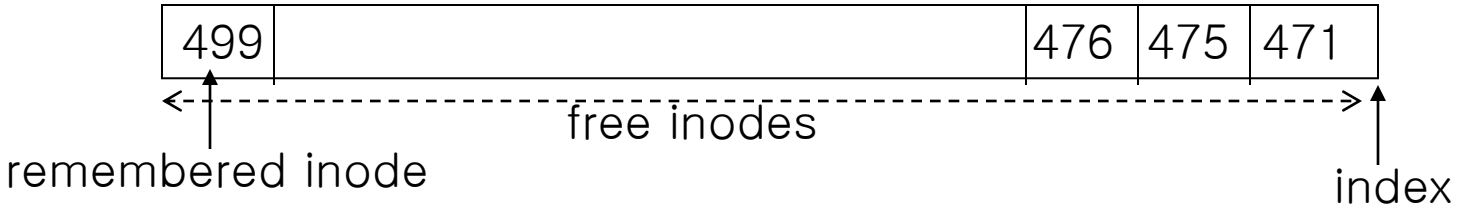


Free Inode 499

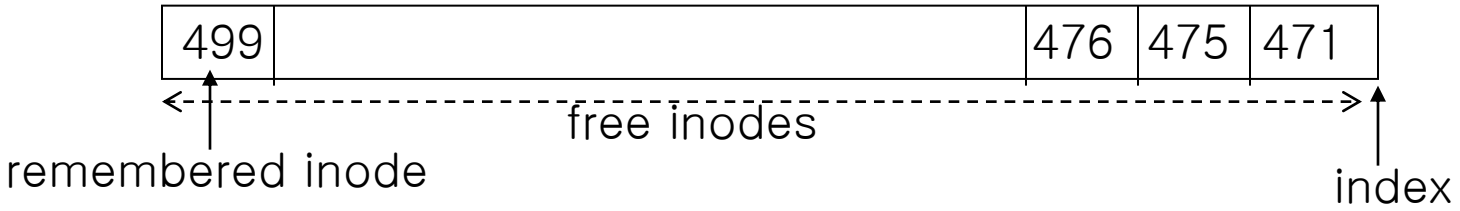
Inode assignment to a new file



Original Super Block List of Free Inodes



Free Inode 499



Free Inode 601

INODE ASSIGNMENT TO A NEW FILE

- The algorithm *ialloc*
  - the kernel first verfies that no other processes have locked access to the super block free inode list.

1) If super block is locked,
   - ➔ Sleep
2) If super block isn't locked and Inode list in super block is empty
   - ➔ Lock super block
   - ➔ Get remembered inode for free inode search
   - ➔ Search disk for free inodes until super block full, or no more free inodes(algorithm bread and brelse)

INODE ASSIGNMENT TO A NEW FILE

➔ Unlock super block

➔ Wake up(event super block becomes free)

① If no free inodes found on disk

➔ Return no inode(so to speak, stop)

② Otherwise

➔set remembered inode for next free inode search

INODE ASSIGNMENT TO A NEW FILE

3) If super block isn't locked and Inode list in super block isn't empty

   there are inodes in super block inode list

   ➔ Get inode number from super block inode list

   ➔ Get inode(algorithm iget)

4) If super block isn't locked and inode is not free after all

   ➔ Write inode to disk

   ➔ Release inode(algorithm iput)

   ➔ Go to loop(while)

INODE ASSIGNMENT TO A NEW FILE

- The algorithm *ialloc*

5) If super block isn't locked and inode is free
   - ➔ Initialize inode
   - ➔ Write inode to disk
   - ➔ Decrement file system free inode count

(freeing inode)

- The algorithm *ifree*
  - algroithm for freeing inode
  - The kernel increments file system free inode count

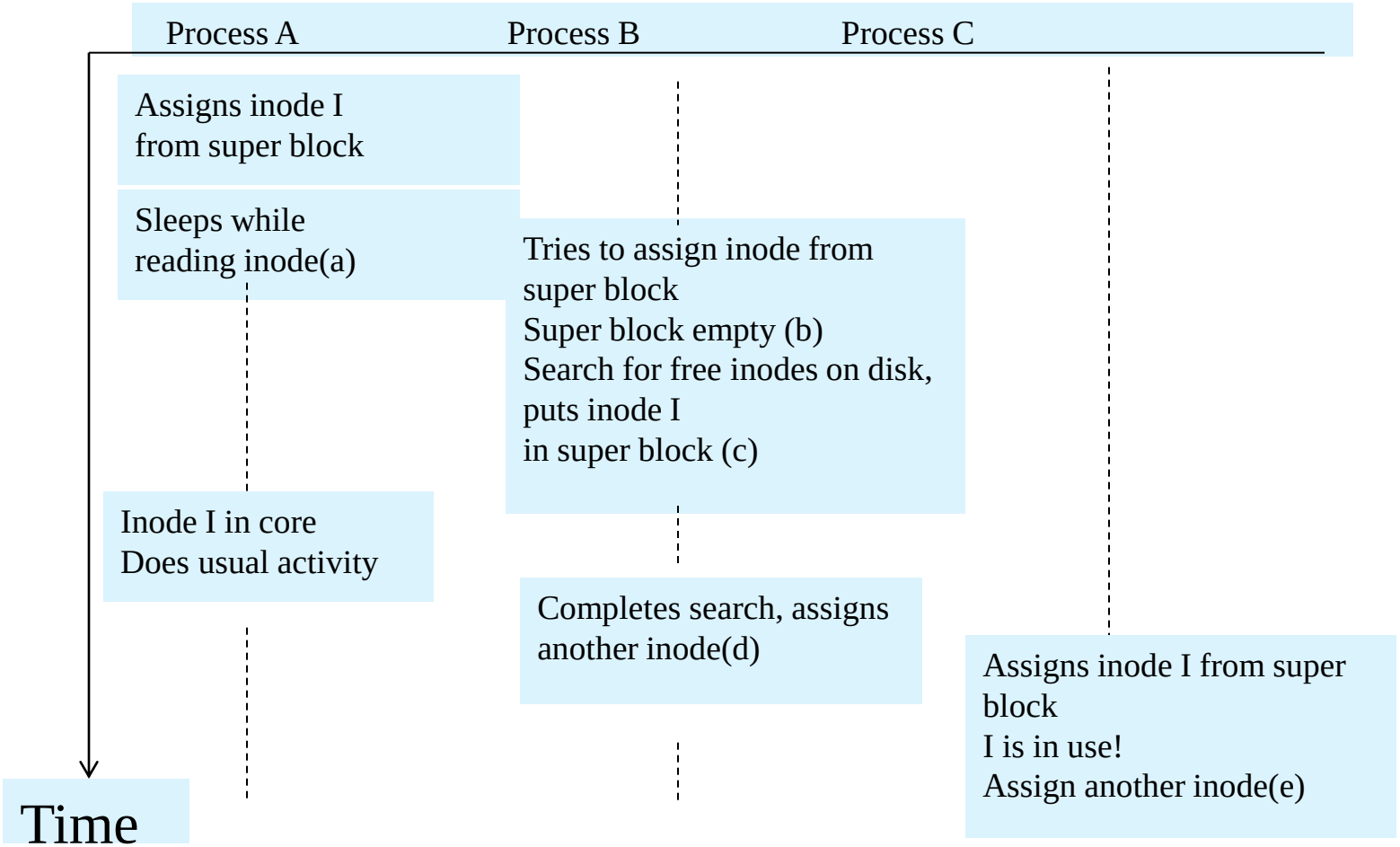
1) If super block is locked
   ➔ avoids race conditions by returning
2) If super block isn't locked , inode list is full ,and inode number is less than remembered inode for search
   ➔ It remembers the newly freed inode number, discarding the old remembered inode number from the super block
3) If super block isn't locked, inode list isn't full
   ➔ Store inode number in inode list

Freeing inode

- ifree (i/p : inode_no ; o/p : none)
  - Increment free inode count
  - If super block locked
    - Return;
  - If (inode list full) //at super block
    - if (inode number <remembered inode)
      - Set remembered inode = input inode number;
  - Else
    - Store inode number in inode list
  - return

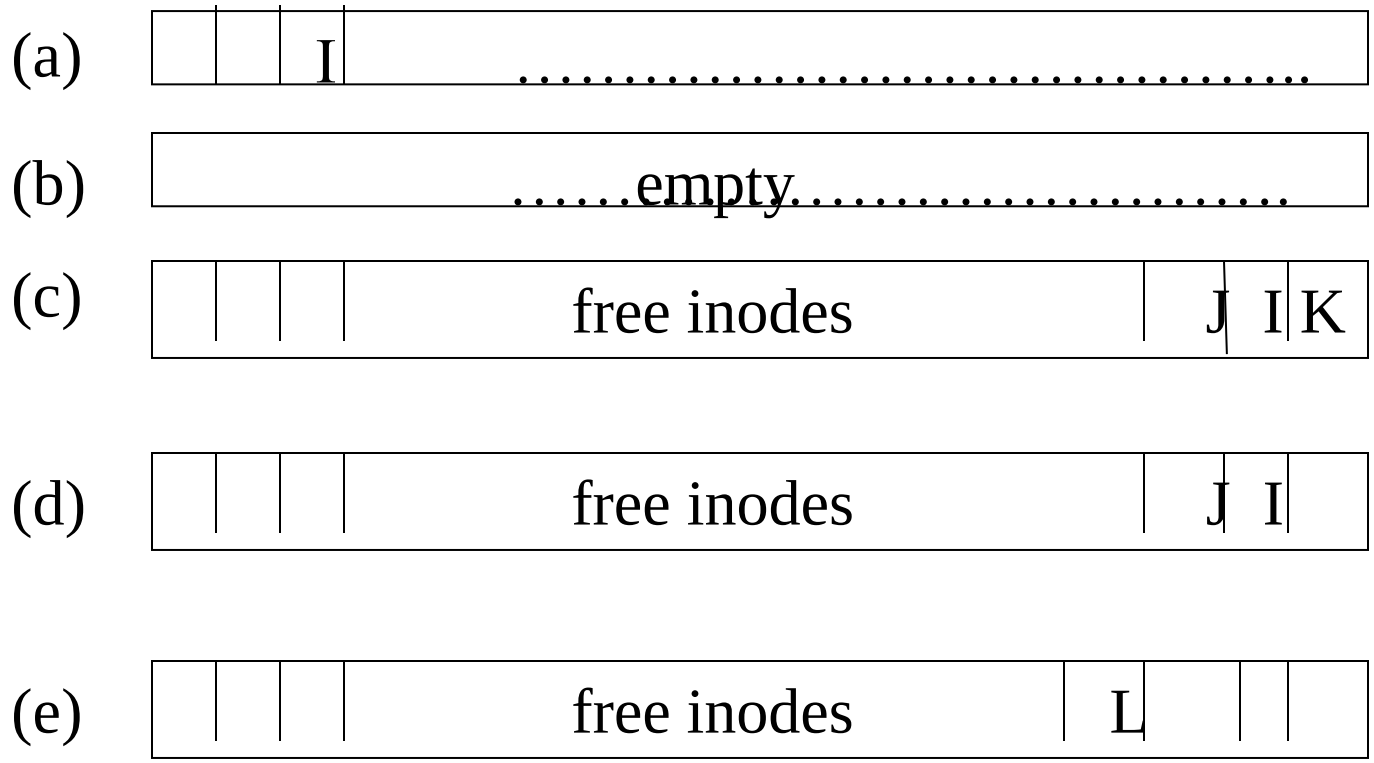
INODE ASSIGNMENT TO A NEW FILE

- Race condition in Assigning inodes



INODE ASSIGNMENT TO A NEW FILE

- Race condition in Assigning inodes



Allocation of disk blocks

- When a process writes data to a file, the kernel must allocate disk blocks from the file system for direct data blocks and sometimes for indirect blocks.
- The file system super block contains an array that is used to cache the numbers of free disk blocks in the file system

ALLOCATION OF DISK BLOCK

1st block is the SBFL and later blocks on the LL contain more free BN.

- Example

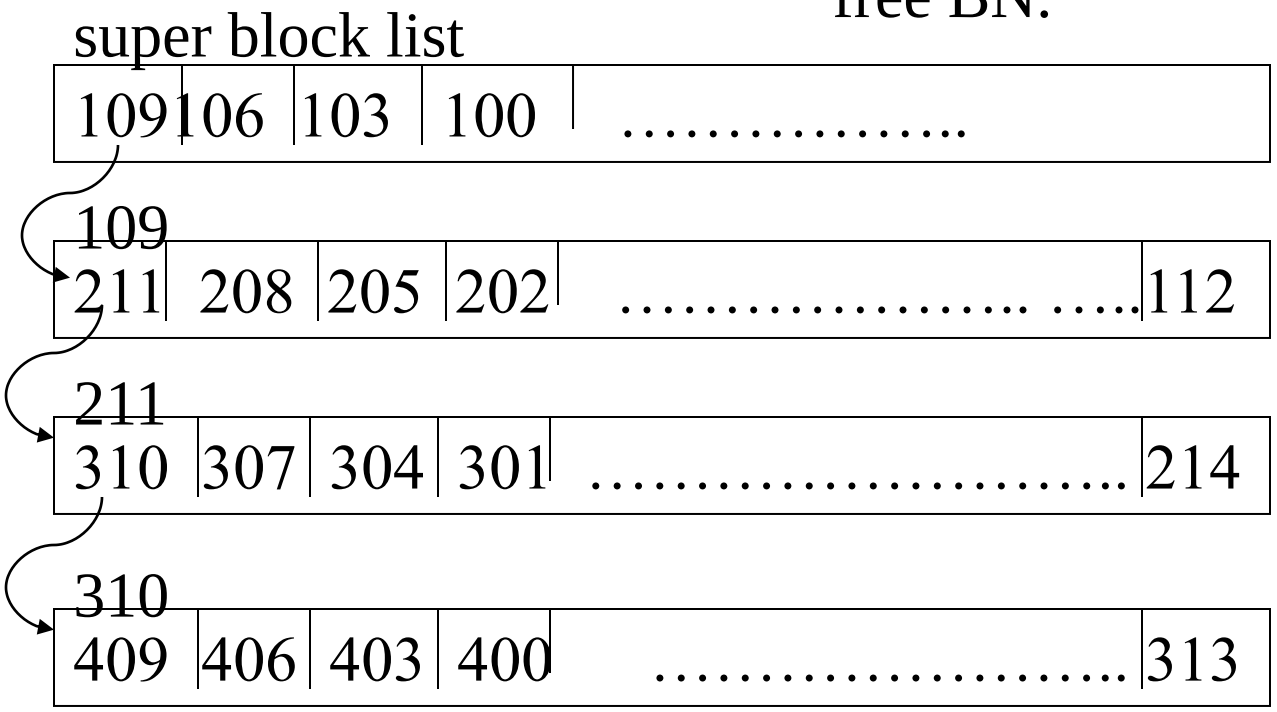
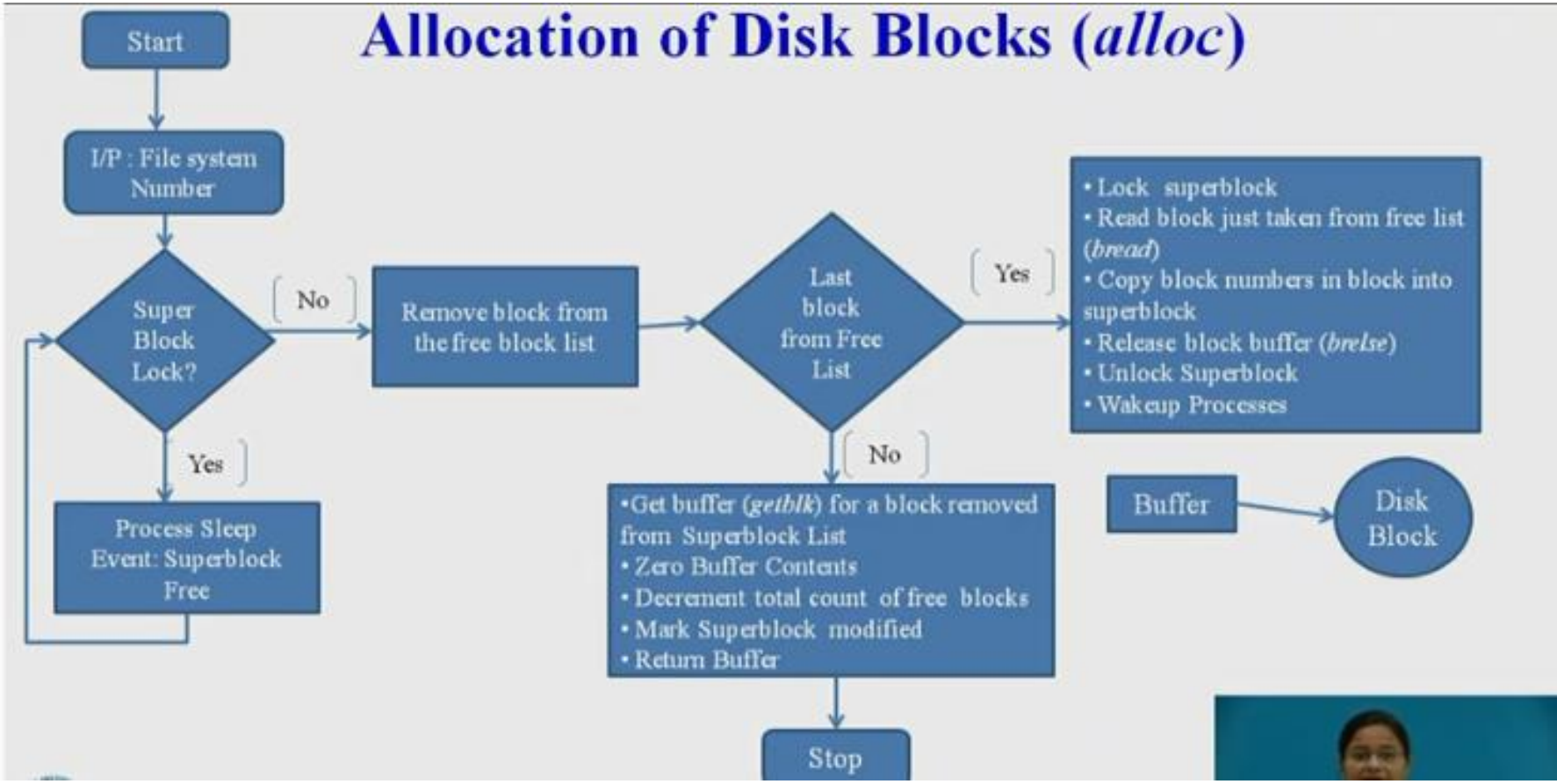


Figure: Linked List of Free Disk Block Numbers

Allocation of Disk Blocks (*alloc*)



ALLOCATION OF DISK BLOCK

- The algorithm *alloc*

  - super block is locked

    ➔ sleep

  1) super block is not locked and removed last block from free list

     ➔ Lock super block

     ➔ Read block just taken from free list(algorithm bread)

     ➔ Copy block numbers in block into super block

     ➔ Release block buffer(algorithm brelse)

     ➔ Unlock super block

     ➔ Wake up process

ALLOCATION OF DISK BLOCK

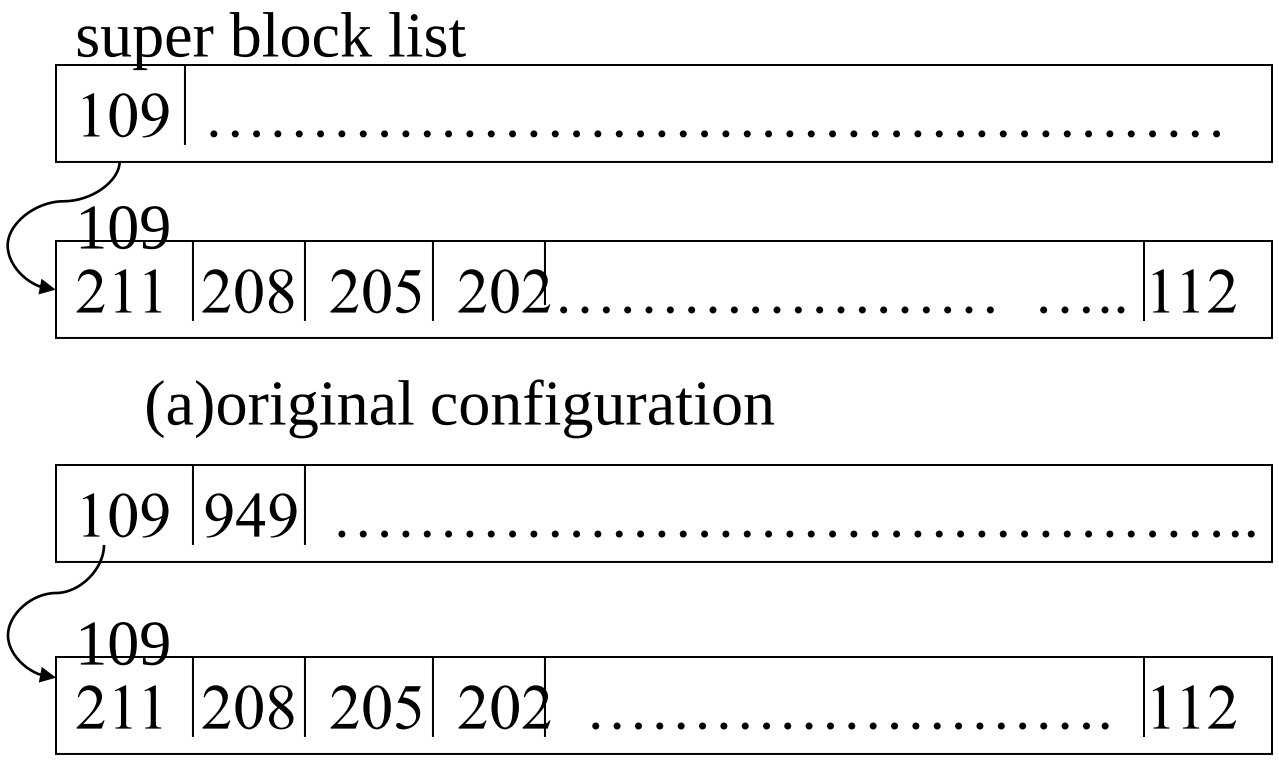
- The algorithm *alloc*

2) super block is not locked

  - ➔ Get buffer for block removed from super block list(algorithm getblk)
  - ➔ Zero buffer contents
  - ➔ Decrement total count of free blocks
  - ➔ Mark super block modified

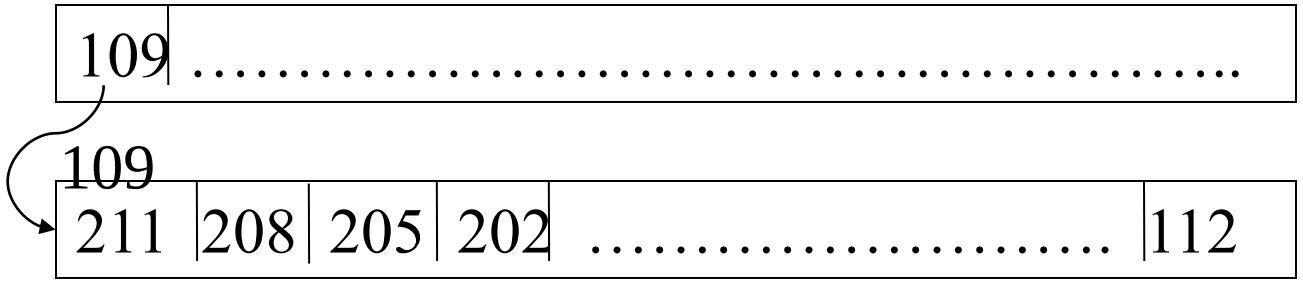
ALLOCATION OF DISK BLOCK

- Example

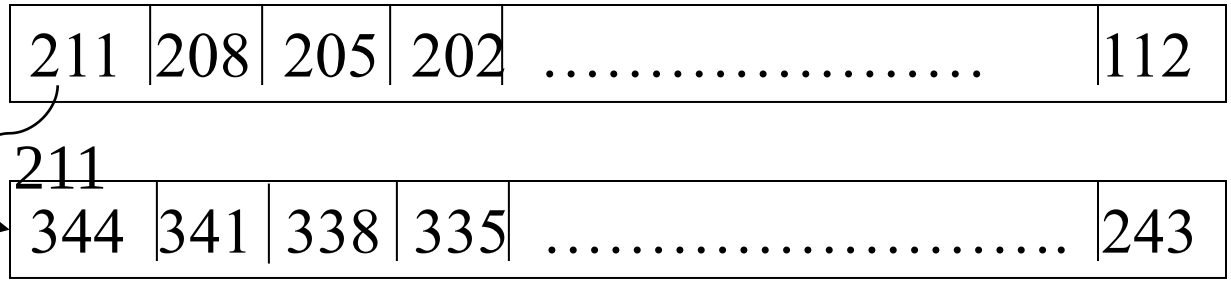


ALLOCATION OF DISK BLOCK

- Example



(c)After assigning block number(949)



(d)After assigning block number(109)
replenish super block free list

ALLOCATION OF DISK BLOCK

- The reason of maintaining a linked list of block number comparing with inodes.
  1. The kernel requires an external method to identify free blocks, but inodes don’t need.
  2. Disk blocks lend themselves to the use of linked lists: a disk block easily holds large lists of free block numbers. But inodes aren’t.
  3. Users tend to consume disk block resources more quickly than they consume inodes.

Other file types: Pipes

- A pipe, sometimes called FIFO, difers from a regular file in that its data is transient: Once data is read from a pipe, it cannot be read again. Also the data is read in order that it was written to the pipe and the system allows no deviation from that order. The kernel stores data in a pipe the same way it stores data in an ordinary file, except that it uses only the direct blocks, not the indirect blocks!

Other file types: special files

- Including block device special files and character device special files. Both types specify devices and therefore the file inodes do not reference any data. Instead, the inode contains two numbers known as the major and minor device numbers: The major number indicates a device type such as terminal or disk, and the minor number indicates the unit number of the device

System Calls

- Provide an interface between a process and the operating system. Generally available as assembly-language instructions

System Calls vs. Library Routines

- A system call is just what its name implies – a request for the OS do something on behalf of the user's program
  - E.g., `read` is a system call which asks the OS to fill a buffer with data stored on a disk drive (or other device). It would be chaotic if everyone were able to access devices whenever they pleased!
- A library routine does not usually need the OS to perform its work
  - E.g., the `sin` function, which computes the sine of an angle

SYSTEM CALLS FOR THE FILE SYSTEM

Return File Desc	Use of namei	Assign inodes	File Attributes	File I/O	File Sys Structure	Tree Manipulation
open creat dup pipe close	open stat creat link chdir unlink chroot mknod chown mount chmod mount	creat mknod link unlink	chown chmod stat	read write lseek	mount umount	chdir chown

System Calls

- System calls that return file descriptors for use in other system calls
- System calls that use the namei algorithm to parse a path name
- System calls that assign and free inodes, using algorithms ialloc and ifree
- System calls that set or change the attributes of a file
- System calls that do I/O to and from a process, using algorithms alloc, free and the buffer application algorithms
- System calls that change the structure of the file system
- System calls that allow a process to change its view of the file system tree.

OPEN

- The open system call is the first step a process must take to access the file

fd = open(pathname,flags, modes);

pathname : is the filename to be opened

flags : read/write

mode : access permission when file is being created

OPEN

Algorithm **open**

Inputs: **file name, type of open, file permissions (for creation type of open)**

Output: **file descriptor**

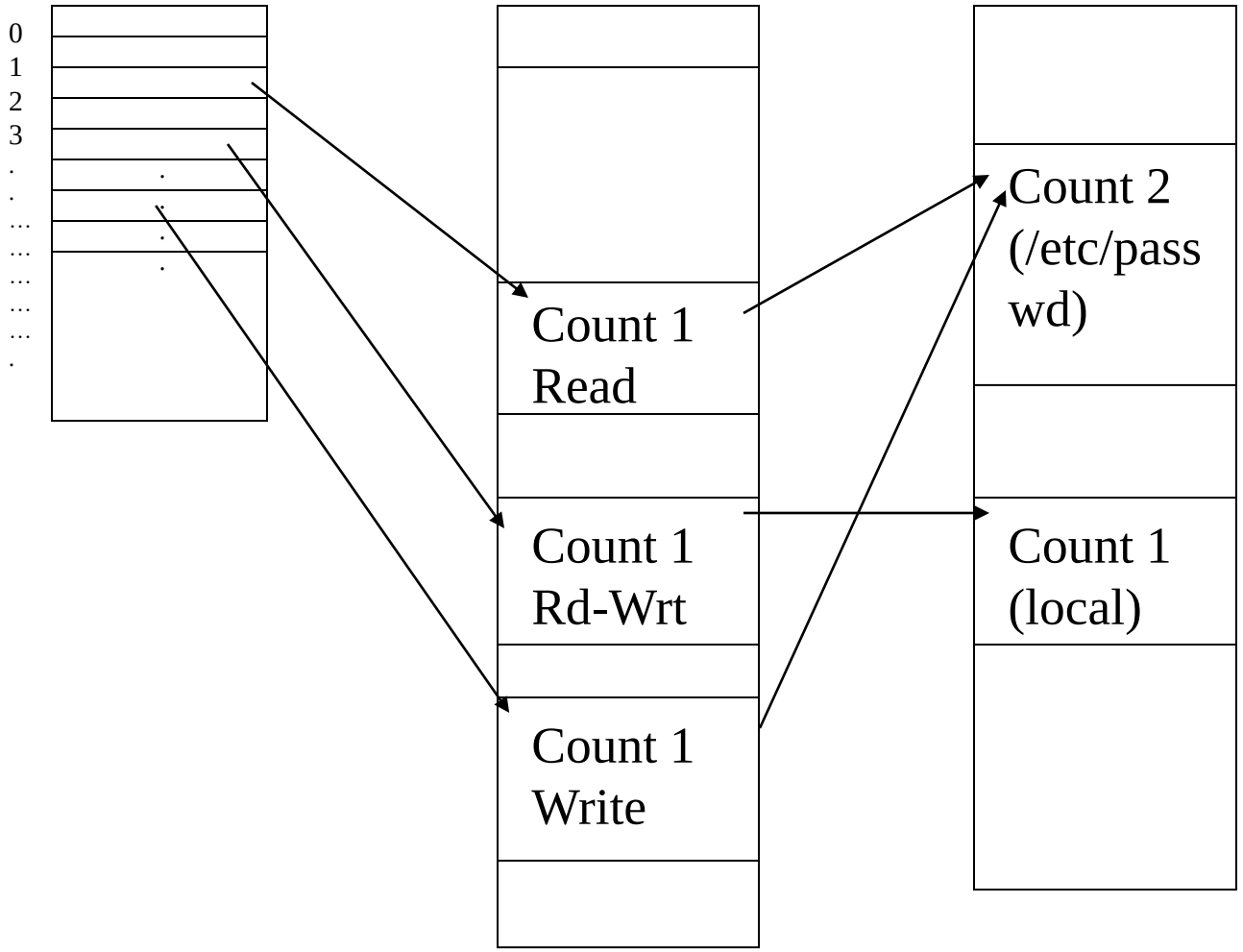
```
{
    convert file name to inode (algorithm namei);
    if (file does not exist or not permitted access) return (error);
    allocate the file table entry for inode, initialise count, offset;
    allocate user file descriptor entry, set pointer to file table entry;
    if (type of open specifies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

Data structures after OPEN

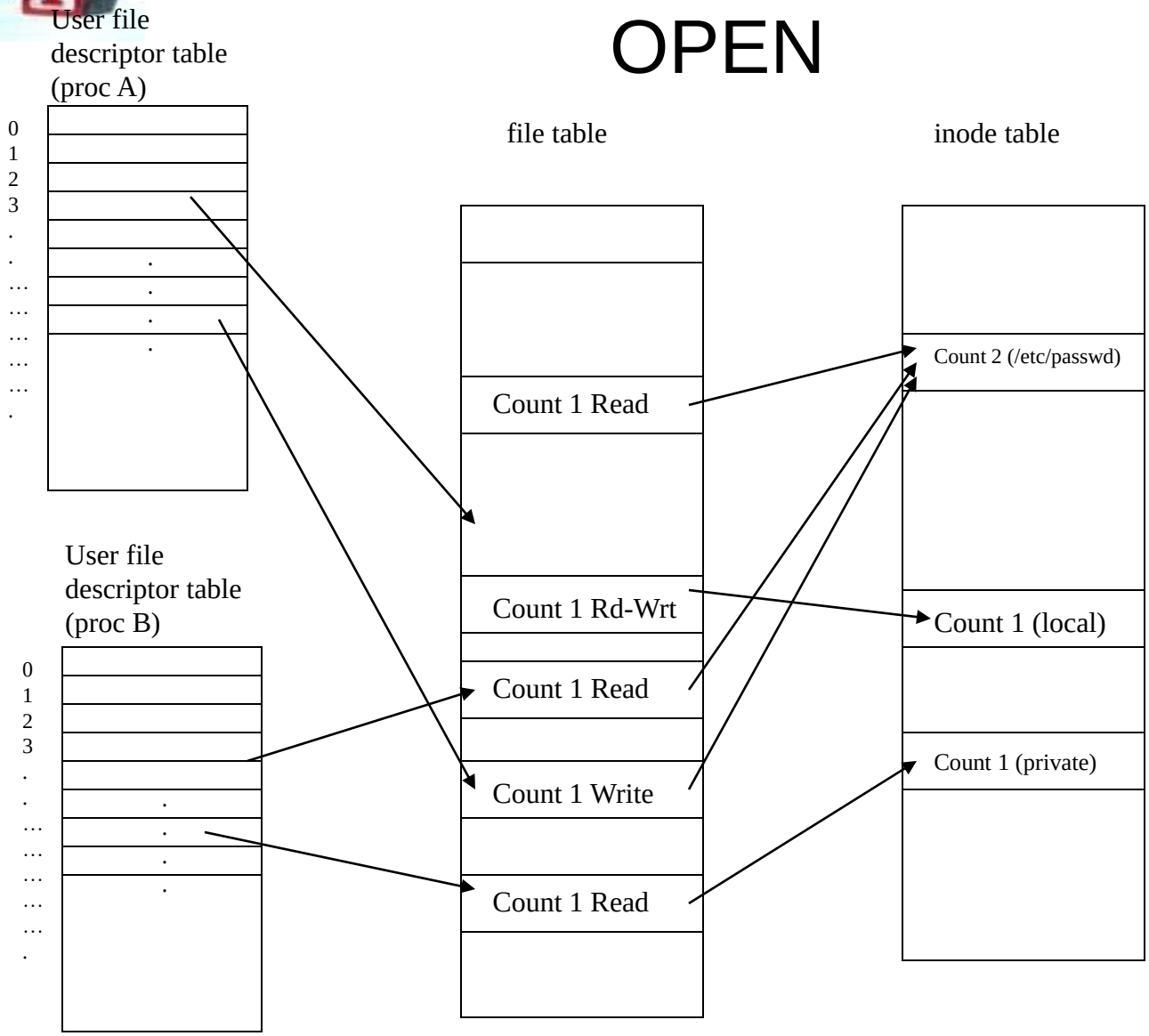
User file descriptor table

file table

inode table



Data structures after two processes OPEN



- number = read(fd, buffer, count);
- fd: file descriptor returned by open
- buffer: address of a data structure in the user process
- count: the number (count) of bytes the user wants to read
- number : no of bytes actually read.

I/O parameters saved in U area

Mode	Indicates read or write
Count	Count of bytes to read or write
Offset	Byte offset in file where the I/O should begin
Address	Target address to copy data ,in user or kernel memory
Flag	Indicates if address is in user address space or kernel memory
Mode	Indicates read or write

READ

algorithm read

inputs: user file descriptor, address of buffer in user process, 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 (algorithm bmap);
        calculate offset into block, number of bytes to read;
        if (number of bytes to read is 0) break;
        read block (algorithm breada if with read ahead, algorithm bread
    otherwise);
        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;
    }
    unlock inode;
    update file table offset for next read;
    return (total number of bytes read);
}
```

- The cycle completes either when the kernel completely satisfy the user request /file contains no more data /if the kernel found an error in reading the data from disk or in copying the data to the user space.
- During each iteration of loop, the kernel saves the next logical block number in the incore inode and during next iteration ,compares the current logical block number to the previously saved.
- If they are equal, the kernel calculate the physical block number for read-ahead and saved its value in the area for use in the breada algorithm.

- If process reaches to read the end of a block then only kernel uses reada for next block.
- When process invokes the read system call ,the kernel locks the inode for the duration of the call.
- The kernel can pre-empt a reading process between system calls in user mode and schedule another process to run.

A reader and writer process

```
//Process A
#include <fcntl.h>
     main() {
            int fd;
            char buf1[512]
fd = open("/etc/passwd", O_RD);
read(fd1,buf1,sizeof(buf1));  //read 1
read(fd2,buf1,sizeof(buf1));  //read2

}
```

```
//Process B
#include <fcntl.h>
     main()
 {
            int fd,i;
           char buf1[512]
        for(i=0;i<sizeof(buf1);i++)
          buf[i]='a';

     fd = open("/etc/passwd", O_WR);
     write(fd1,buf1,sizeof(buf1));      //write 1
     write(fd2,buf1,sizeof(buf1));      //write 2

}
```

Reading a file via two descriptors:

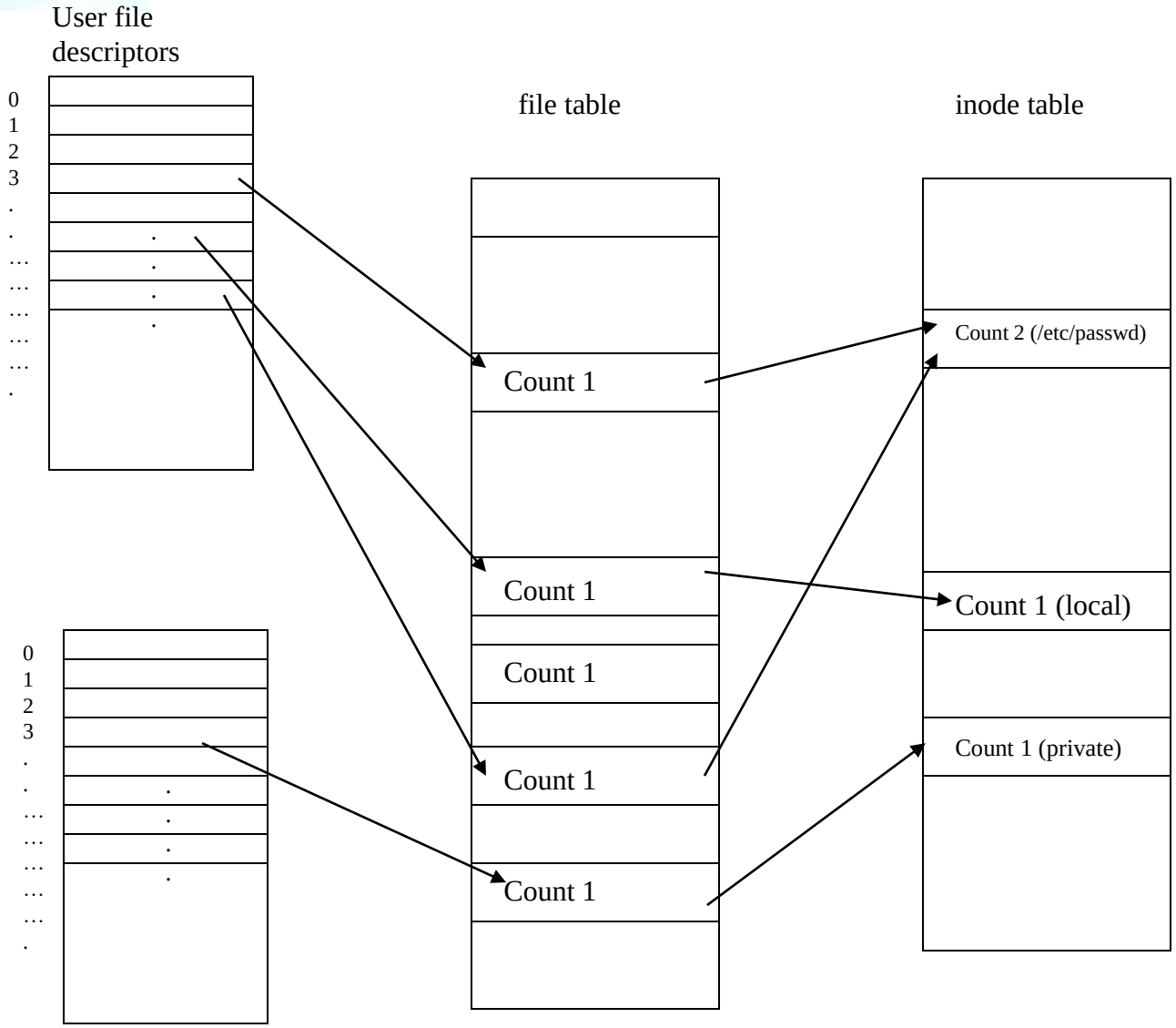
```
#include <fcntl.h>
    main()
    {
        int fd1, fd2;
        char buf1[512], buf2[512];

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

WRITE

write (fd,buffer,count)

Tables after CLOSING a file



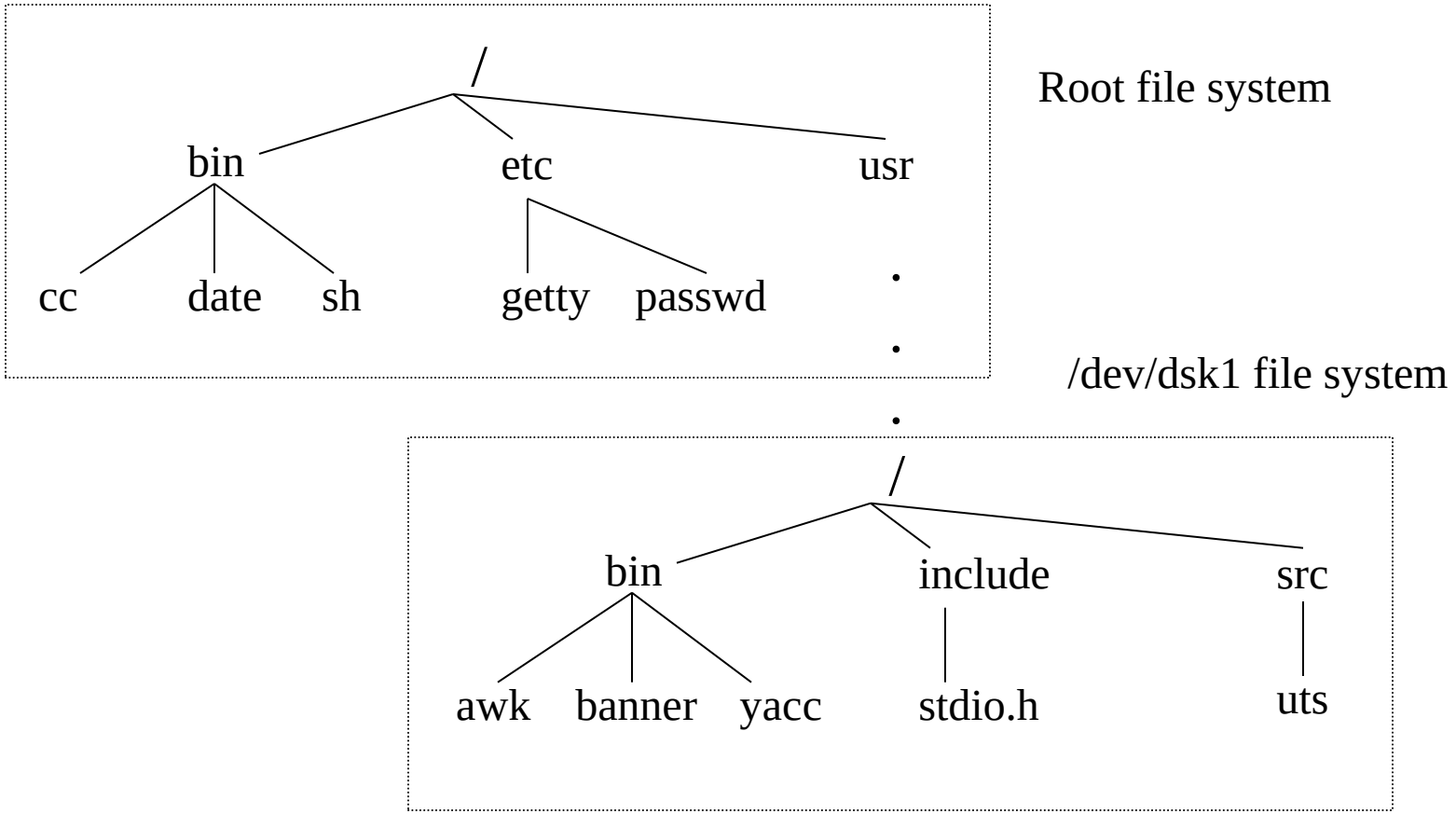
FILE CREATION

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

chdir, chroot

- chdir(pathname)
- chroot(pathname)
- chown(pathname, owner, group)
- chmod(pathname, mode)

File System tree before and after Mount



LINK - UNLINK

- The link system call links a file to a new name in the file system directory structure, creating a new directory entry for an exitsing node
- link(source file name, target file name)
- The unlink system call removes a directory entry for a file
- unlink(pathname)

FILE SYSTEM MAINTENANCE

- The kernel maintains consistency of the file system during normal operation.
- The command fsck checks for consistency and repairs the file system in case of extraordinary circumstances (power failure)

Process

Process Control

- The use and implementation of the system call that Control the Process Context
  - **fork** : create a new process
  - **exit** : terminate process execution
  - **wait** : allow parent process to synchronize its execution with the exit of a child process
  - **exec** : invoke a new program
  - **brk** : allocate more memory dynamically
  - **signal** : inform asynchronous event
  - Process system calls and relation to other algorithms

System Calls Dealing with Memory Management			System Calls Dealing with Synchronization				Miscellaneous	
fork	exec	brk	exit	wait	signal	kill	setpgrp	setuid
dupreg attachreg	detachreg allocreg attachreg growreg loadreg mapreg	growreg	detachreg					

PROCESS STATES AND TRANSITIONS

- In Unix a process undergoes certain Transition during it's execution cycle .
- The lifetime of a process can be conceptually divided into a set of states that describe the process.
- The following list contains the complete set of process states

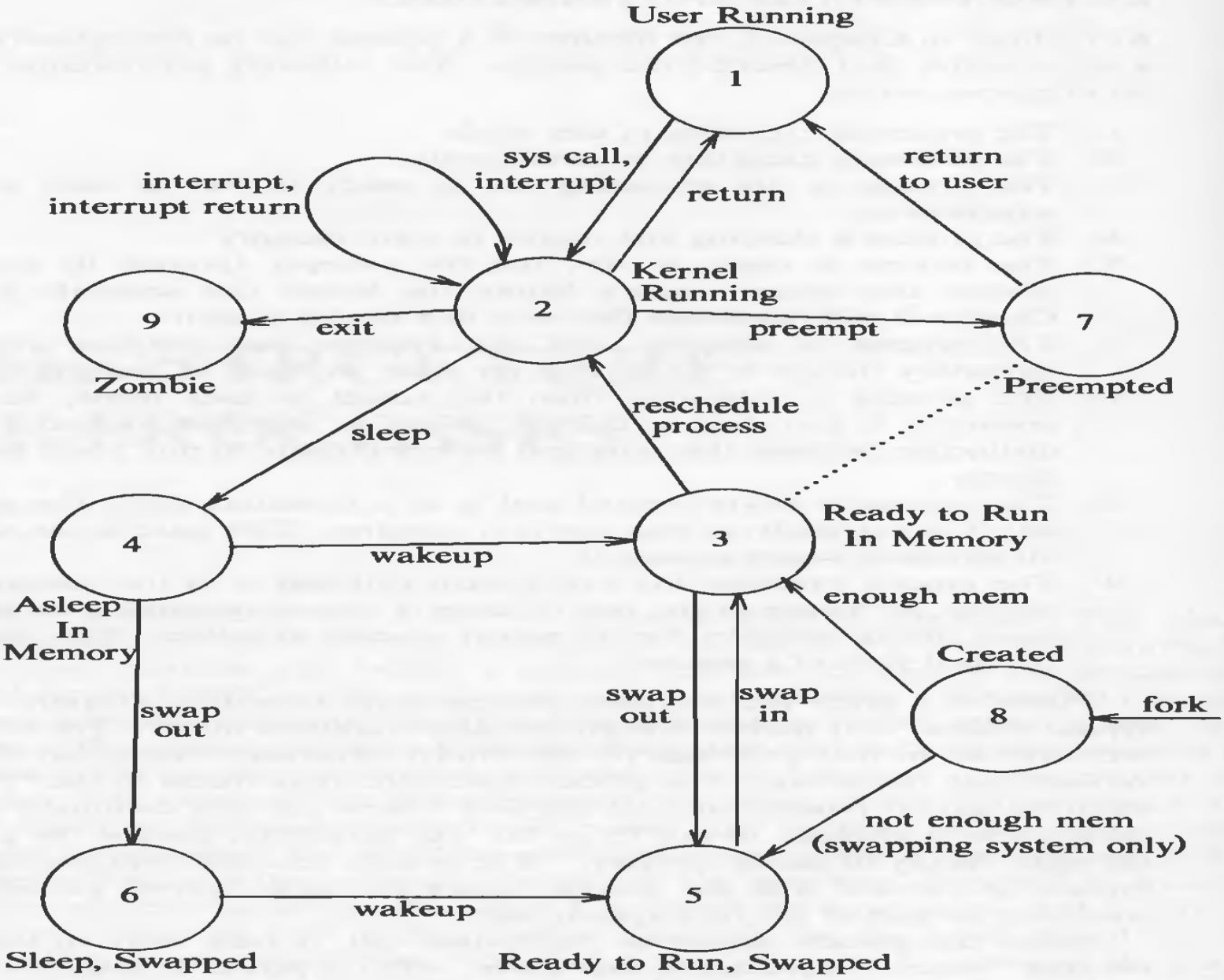


Figure 6.1. Process State Transition Diagram

PROCESS STATES AND TRANSITIONS

- 1. The process is executing in user mode.
- 2. The process is executing in kernel mode.
- 3. The process is not executing but is ready to run as soon as the kernel schedules it.
- 4. The process is sleeping and resides in main memory.
- 5. The process is ready to run, but the swapper (process 0) must swap the process into main memory before the kernel can schedule it to execute, we can reconsider this state as a sort of paging system.

PROCESS STATES AND TRANSITIONS

- 6. The process is sleeping, and the swapper has swapped the process to secondary storage to make room for other processes in main memory.
- 7. The process is returning from the kernel to user mode, but the kernel preempts it and does a context switch to schedule another process.
- 8. The process is newly created and is in a transition state; the process exists, but it is not ready to run, nor is it sleeping. This state is the start state for all processes except process 0.

PROCESS STATES AND TRANSITIONS

- 9. The process executed the exit system call and is in the zombie state.
  - The zombie state is the final state of a process.

User Running	Executing in user mode.
Kernel Running	Executing in kernel mode.
Ready to Run, in Memory	Ready to run as soon as the kernel schedules it.
Asleep in Memory	Unable to execute until an event occurs; process is in main memory (a blocked state).
Ready to Run, Swapped	Process is ready to run, but the swapper must swap the process into main memory before the kernel can schedule it to execute.
Sleeping, Swapped	The process is awaiting an event and has been swapped to secondary storage (a blocked state).
Preempted	Process is returning from kernel to user mode, but the kernel preempts it and does a process switch to schedule another process.
Created	Process is newly created and not yet ready to run.
Zombie	Process no longer exists, but it leaves a record for its parent process to collect.

- Preemption can only occur when the process is about to move from Kernel mode to User mode.
- While the process is running in Kernel mode it may not be preempted.

- The process has control over some state transitions at user level .

1. a process can create another process.(but no control over where to move in state 3 / 5).
2. A process can make system calls to move from state user running to state kernel running (but no control over when it will return from the kernel or directly moves to zombie state).
3. A process can exit of its own volition.

KERNEL DATA STRUCTURES DESCRIBING THE STATE OF A PROCESS

- Two kernel data structures describe the state of a process:
- **the process table entry and the u area.**
- The kernel contains a **process table** with an entry that describes the state of every active process in the system.
- The **u area** contains additional information that controls the operation of a process.
- The **process table** contains fields that must always be accessible to the kernel, but the **u area** contains fields that need to be accessible only to the running process. Therefore, the kernel allocates space for the u area only when creating a process: It does not need u areas, for process table entries that do not have processes.

KERNEL DATA STRUCTURES DESCRIBING THE STATE OF A PROCESS

- The process table entry and the u area are part of the context of a process.

FIELDS OF PROCESS TABLE DESCRIBING THE STATE OF A PROCESS

The fields in the process table are the following.

- **state field**

identifies the process state.

- **The process table entry contains fields that allow the kernel to locate the process and its u area in main memory or in secondary storage.**
- The kernel uses the information to do a context switch to the process when the process moves from state "ready to run in memory" to the state "kernel running" or from the state "preempted" to the state "user running.“
- **The process table entry also contains a field that gives the process size, so that the kernel knows how much space to allocate for the process.**

FIELDS OF PROCESS TABLE DESCRIBING THE STATE OF A PROCESS

- **Several user identifiers (user IDs or UIDs)** determine various process privileges. For example, the user ID fields define the sets of processes that can send signals to each other and **who owns the process.**
- **Process identifiers (process IDs or PIDs)** specify the relationship of processes to each other. These ID fields are set up when the process enters the state "created" in the fork system call.
- The process table entry contains an **event descriptor** when the process is in the "sleep" state.

FIELDS OF PROCESS TABLE DESCRIBING THE STATE OF A PROCESS

- **Scheduling parameters** allow the kernel to determine the order in which processes move to the states "kernel running" and "user running."
- **A signal field** enumerates the signals sent to a process but not yet handled .
- **Various timers** give process execution time and kernel resource utilization, used for process accounting and for the calculation of process scheduling priority. **One field is a user-set timer used to send an alarm signal to a process**


```
algorithm dupreg        /* duplicate an existing region */
input:   pointer to region table entry
output: pointer to a region that looks identical to input region
{
        if (region type shared)
                /* caller will increment region reference count
                 * with subsequent attachreg call
                 */
                return(input region pointer);
        allocate new region (algorithm allocreg);
        set up auxiliary memory management structures, as currently
                exists in input region;
        allocate physical memory for region contents;
        "copy" region contents from input region to newly allocated
                region;
        return(pointer to allocated region);
}
```

Figure 6.28. Algorithm for Dupreg

PROCESS CREATION

- The only way for a user to create a new process in the UNIX operating system is to invoke the fork system call.
- The process that invokes fork is called the **parent process**, and the newly created process is called the **child process**.
- The syntax for the fork system call is

pid = fork();

- On return from the fork system call, the two processes have identical copies of their user-level context except for the return value pid.

PROCESS CREATION

- In the parent process,
  - pid is the child process ID;
- in the child process,
  - pid is 0.

- Process 0, created internally by the kernel when the system is booted, is the only process not created via fork.

PROCESS CREATION

- The kernel does the following sequence of operations for fork.

1. It allocates a slot in the process table for the new process.
2. It assigns a unique ID number to the child process.
3. It makes a logical copy of the context of the parent process. Since certain portions of a process, such as the text region, may be shared between processes, the kernel can sometimes increment a region reference count instead of copying the region to a new physical location in memory.

The kernel does the following sequence of operations for fork

1. It allocates a slot in process table for the new process

2. It assigns a unique ID number to the child process

3. It makes a logical copy of the context of the parent process. Since certain portion of process, such as the text region, may be shared between processes, the kernel can sometimes increment a region reference count instead of copying the region to a new physical location in memory

4. It increments file and inode table counters for files associated with the process.

5. It returns the ID number of child process to the parent process, and a 0 value to the child process.

Algorithm for fork

input : none

output : to parent process, child PID number

to child process, 0

{

check for available kernel resources;

get free process table slot, unique PID number;

check that user not running too many process;

mark child state "being created";

copy data from parent process table to new child slot; (user and super user)

// [the child process inherits the parent process real and effective user id, process group and parent nice value used for calculation of scheduling priority]

increment counts on current directory inode and changed root(if applicable);

increment open file counts in file table;

make copy of parent context(u area, text, data, stack) in memory; [user level context is created]

// **it allocates mem for the child process u area ,regions and auxillary page tables ,duplicate every**

// region in the parent process using algorithm dupreg, and attaches every region to the child process // using algo attachreg.

//In a swapping system , it copies the contents of regions that are not shared into a new area of main memory .

On a swapping system : it needs space either in memory or on disk to hold the child process

On a paging system : it has to allocate memory for auxiliary tables such as page tables

Algorithm for fork(Cont.)

```
// [now kernel creates the dynamic portion of child context ]
    push dummy system level context layer onto child system level context;
                    dummy context contains data allowing child process
                    to recognize itself, and start from here when scheduled;
    if ( executing process is parent )
    {
        change child state to "ready to run";
        return(child ID);                   /* from system to user */
    }
    else    /* executing process is the child process */
    {
        initialize u area timing fields;
        return(0);              /* to user */
    }
}
```

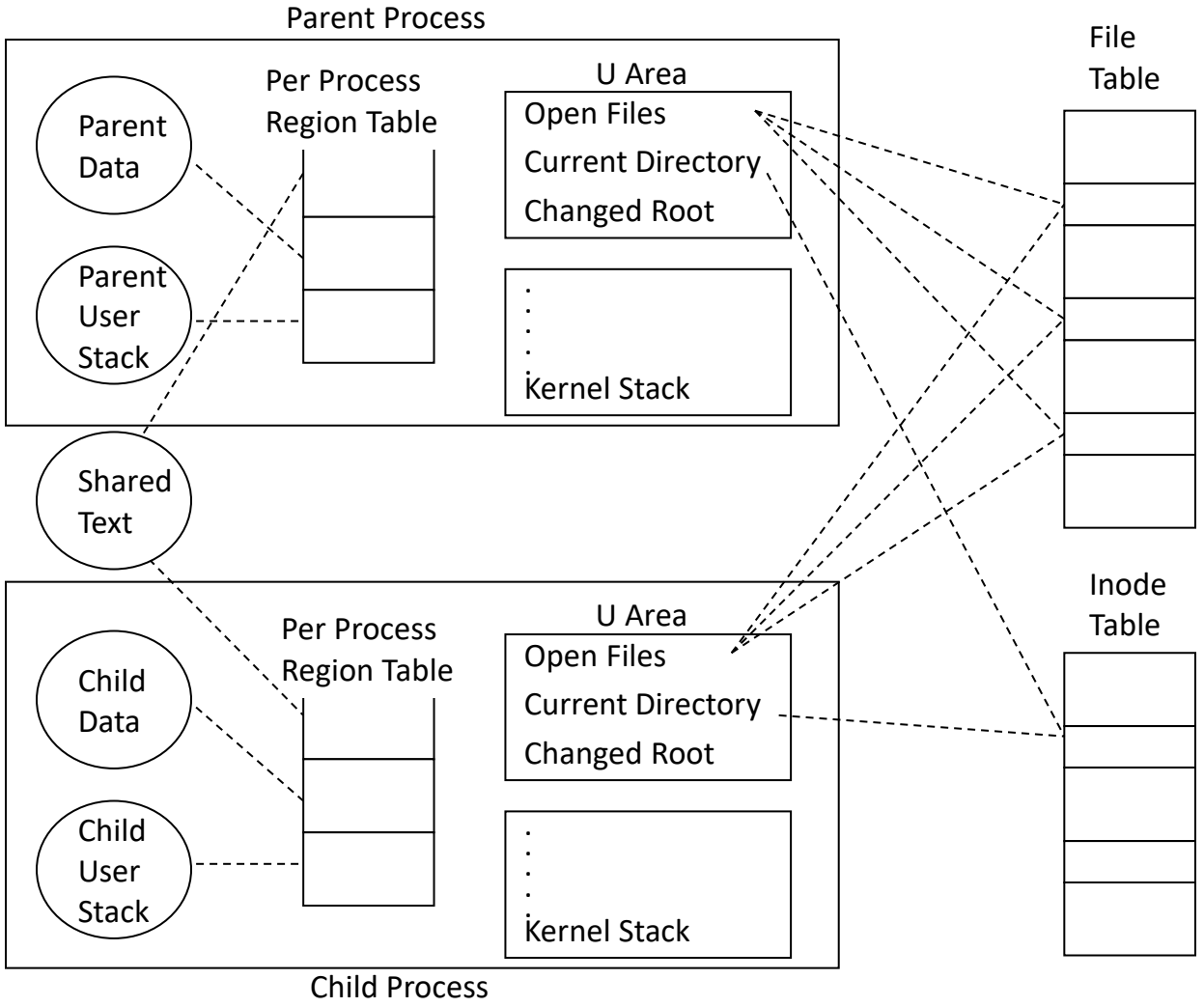
The kernel is now ready to create the user-level context of the child process. It allocates memory for the child process *u area*, regions, and auxiliary page tables, duplicates every region in the parent process using algorithm *dupreg*, and attaches every region to the child process using algorithm *attachreg*.

So far, the kernel has created the static portion of the child context; now it creates the dynamic portion. The kernel copies the parent context layer 1, containing the user saved register context and the kernel stack frame of the *fork* system call. If the implementation is one where the kernel stack is part of the *u area*, the kernel automatically creates the child kernel stack when it creates the child *u area*. Otherwise, the parent process must copy its kernel stack to a private area of memory associated with the child process. In either case, the kernel stacks for the parent and child processes are identical. The kernel then creates a dummy context layer (2) for the child process, containing the saved register context for context layer (1). It sets the program counter and other registers in the saved register context so that it can "restore" the child context, even though it had never executed before, and so that the child process can recognize itself as the child when it runs. For instance, if the kernel code tests the value of register 0 to decide if the process is the parent or the child, it writes the appropriate value in the child saved

register context in layer 1.

When the child context is ready, the parent completes its part of *fork* by changing the child state to "ready to run (in memory)" and by returning the child process ID to the user. The kernel later schedules the child process for execution via the normal scheduling algorithm, and the child process "completes" *its* part of the *fork*. The context of the child process was set up by the parent process; to the kernel, the child process appears to have awakened after awaiting a resource. The child process executes part of the code for the *fork* system call, according to the program counter that the kernel restored from the saved register context in context layer 2, and returns a 0 from the system call.

Fork Creating a New Process Context



Example of Sharing File Access

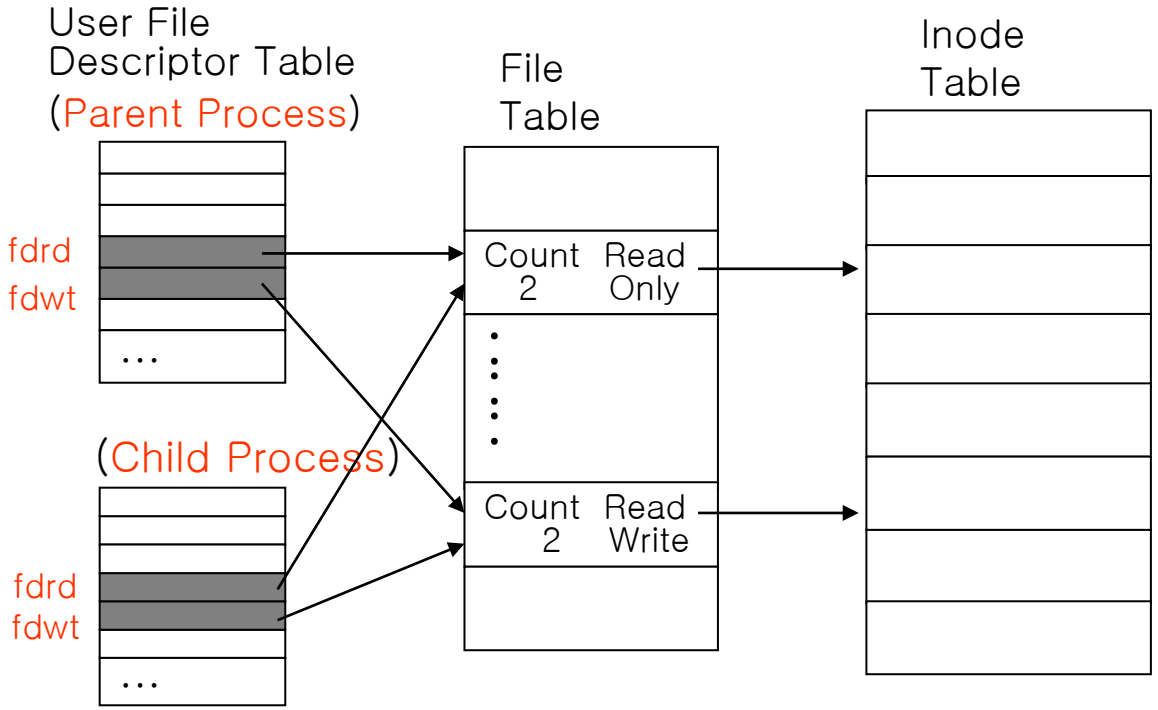
```
#include <fcntl.h>
int fdrd, fdwt;
char c;

main( argc,argv )
        int argc;
        char *argv[];
{
        if ( argc != 3 )
            exit(1);
        if ((fdrd=open(argv[1],O_RDONLY))==-1)
            exit(1);
        if ((fdwt=creat(argv[2],0666))==-1)
            exit(1);
        fork();
        /*both process execute same code*/
        rdwt();
        exit(0);
}
```

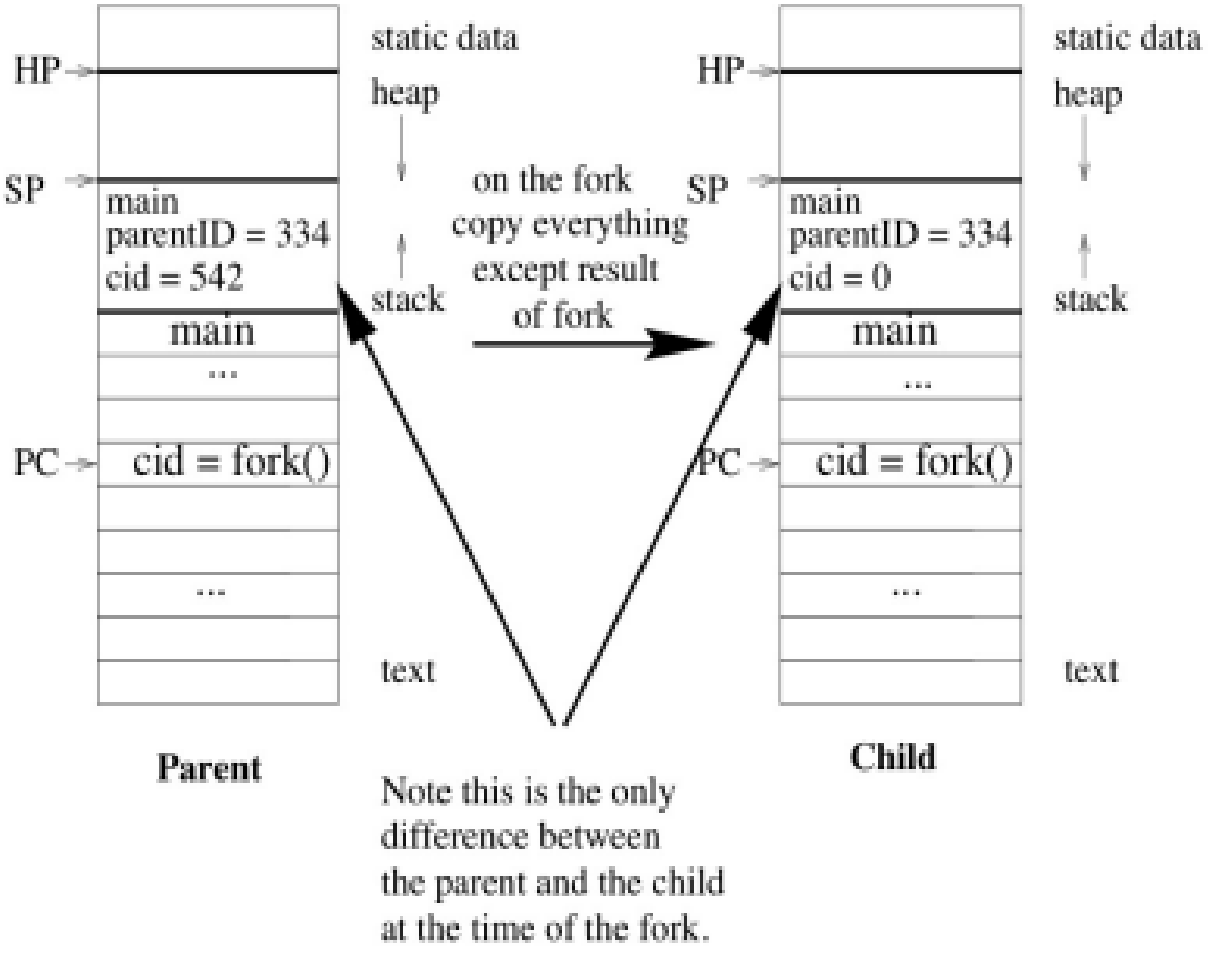
```
rdwt()
{
        for(;;)
        {
                if (read(fdrd,&c,1)!=-1)
                    return;
                write(fdwt,&c,1);
        }
}
```

Example of Sharing File Access(Cont.)

- *fdrd* for both process refer to the file table entry for the source file(argv[1])
- *fdwt* for both process refer to the file table entry for the target file(argv[2])
- two processes never read or write the same file offset values.



What is happening on the Fork



Child Process' Inheritance

- Parent process real and effective id
- Parent process group and nice value(for calculation of scheduling priority)
- controlling terminal
- current working directory
- root directory
- file mode creation
- close-on-exec flag for open file descriptors
- environment
- attached shared memory segments
- resource limits

Items That Distinguish Parent From Child

- Return value from fork
- unique pid
- different parent pid (ppid)
  - child's ppid is the parent's pid
  - parent's pid stays same
- certain time values reset in child
- parent's file locks aren't inherited
- pending alarms are not inherited
- pending signals in parent aren't inherited

Examples of interrupt

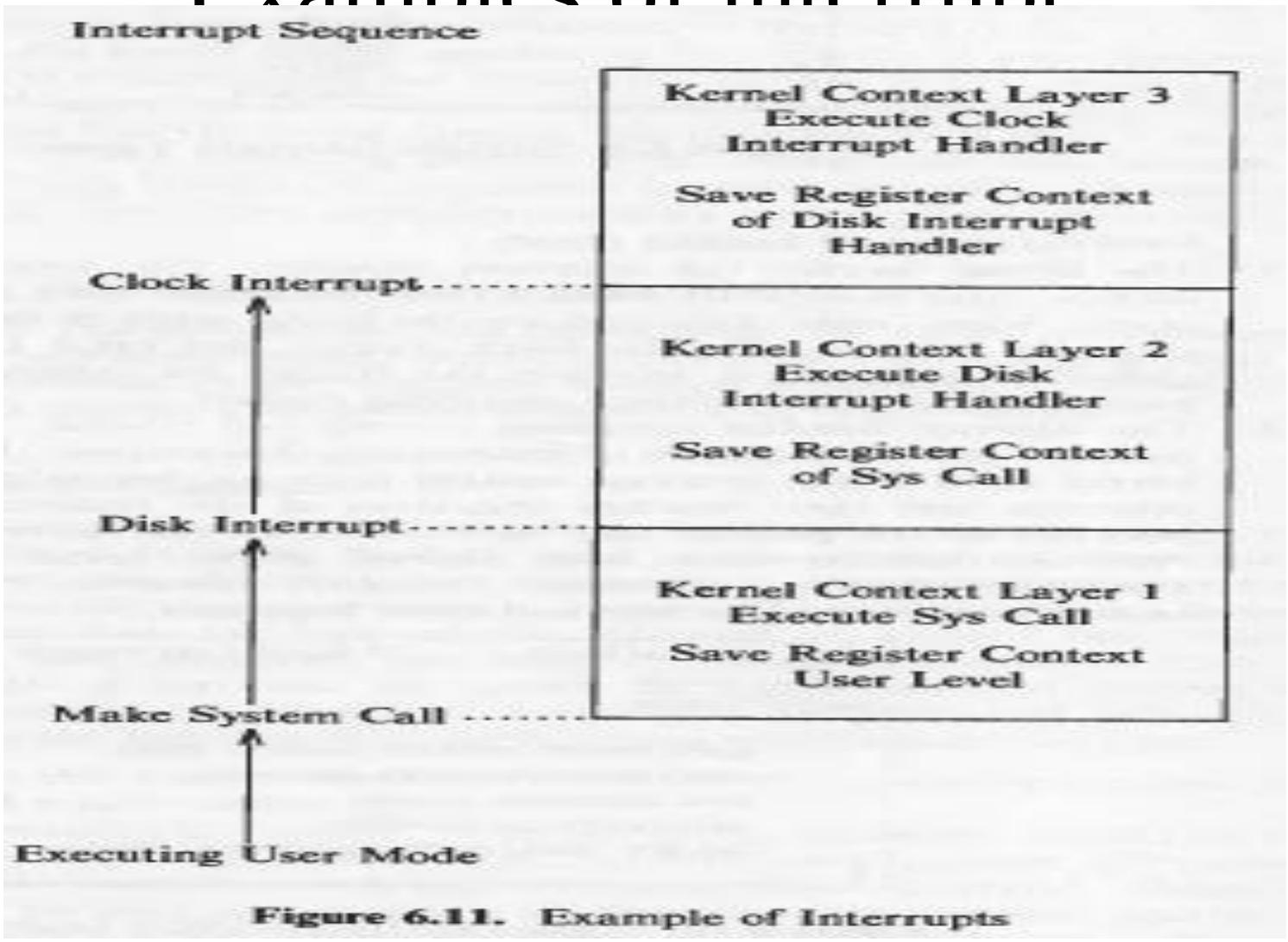


Figure 6.11. Example of Interrupts

Context switch

1. Decide whether to do a context switch, and whether a context switch is permissible now.
2. Save the context of the "old" process.
3. Find the "best" process to schedule for execution, using the process scheduling algorithm in Chapter 8.
4. Restore its context.

Figure 6.15. Steps for a Context Switch

Sleep algorithm ,which changes the process state from "kernel running" to "asleep in memory" and wakeup algorithm that changes the process state from "asleep" to "ready to run" in memory or swapped

Sleep Algorithm

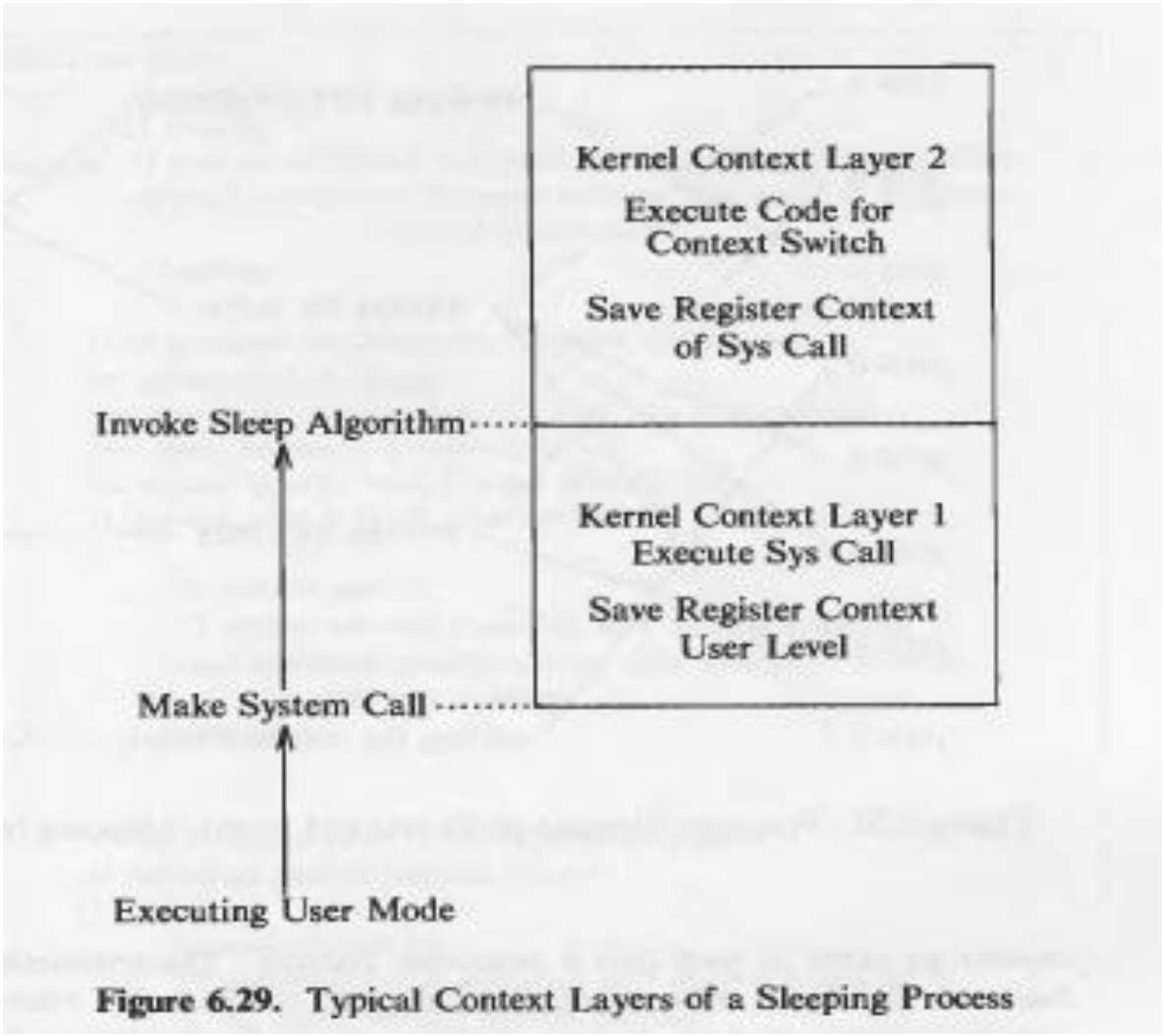


Figure 6.29. Typical Context Layers of a Sleeping Process

Sleep Algorithm

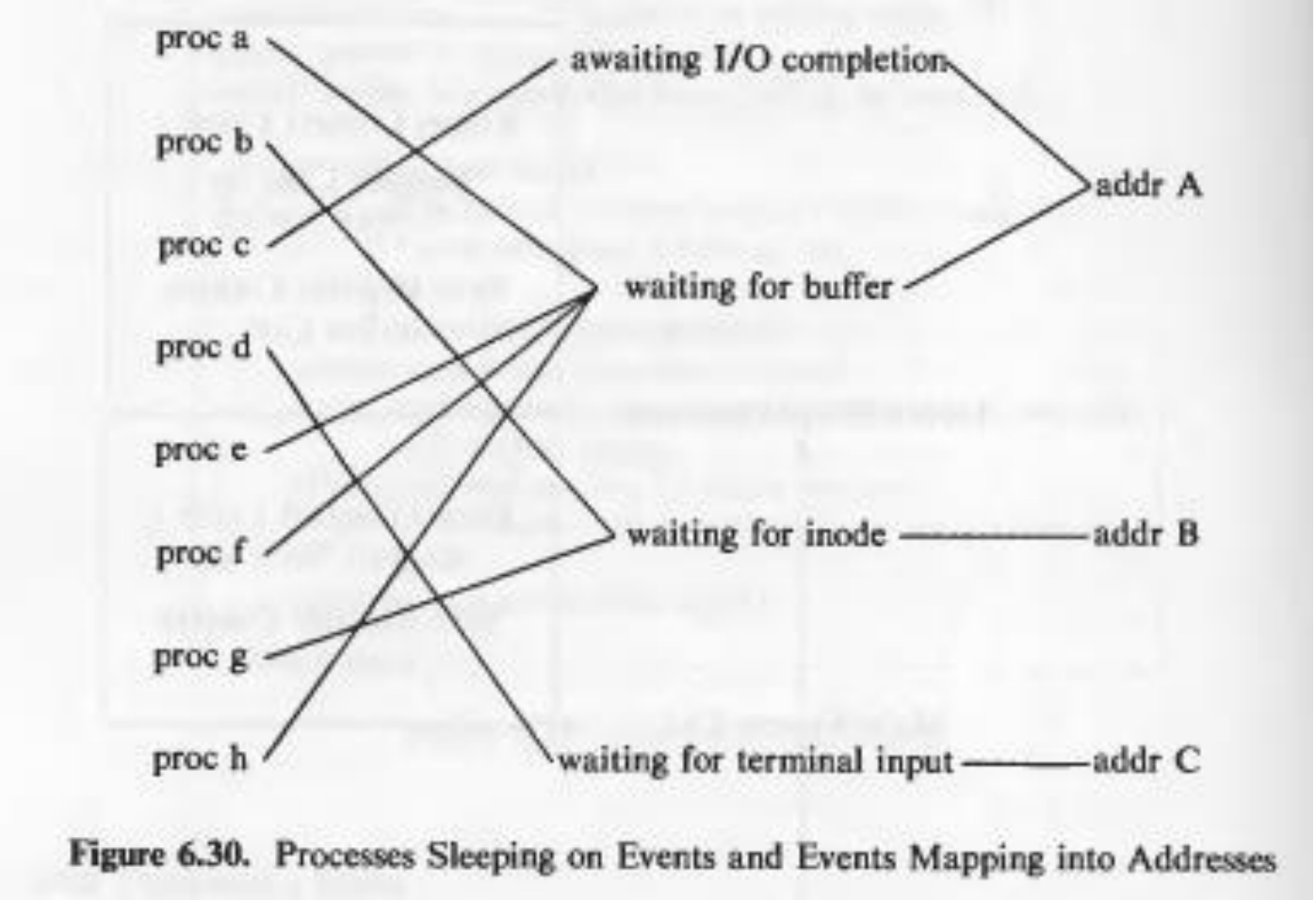


Figure 6.30. Processes Sleeping on Events and Events Mapping into Addresses

The implementation maps the Set of events into a set of kernel virtual addresses

Two anomalies

1. when an event occurs and a wakeup call is issued for processes that are sleeping on the event ,they all wakeup and move from a sleep state to a ready to run state.
2. Several events may map into one address.

Sleep Algorithm

```
algorithm sleep
input:   (1) sleep address
         (2) priority
output: 1 if process awakened as a result of a signal that process catches,
         longjump algorithm if process awakened as a result of a signal
                                  that it does not catch,
         0 otherwise;
{
      raise processor execution level to block all interrupts;
      set process state to sleep;
      put process on sleep hash queue, based on sleep address;
      save sleep address in process table slot;
      set process priority level to input priority;
      if (process sleep is NOT interruptible)
      {
            do context switch;
            /* process resumes execution here when it wakes up */
            reset processor priority level to allow interrupts as when
                  process went to sleep;
            return(0);
      }
```

Sleep Algorithm

```
/* here, process sleep is interruptible by signals */
if (no signal pending against process)
{
        do context switch;
        /* process resumes execution here when it wakes up */
        if (no signal pending against process)
        {
                reset processor priority level to what it was when
                        process went to sleep;
                return(0);
        }
}
remove process from sleep hash queue, if still there;

reset processor priority level to what it was when process went to sleep;
if (process sleep priority set to catch signals)
        return(1)
do longjmp algorithm;
}
```

The kernel does a **longjump** to restore a previously saved context if it has no way to complete the system call it is executing

Wakeup Algorithm

```
algorithm wakeup                    /* wake up a sleeping process */
input:  sleep address
output: none
{
        raise processor execution level to block all interrupts;
        find sleep hash queue for sleep address;
        for (every process asleep on sleep address)
        {
                remove process from hash queue;
                mark process state "ready to run";
                put process on scheduler list of processes ready to run;
                clear field in process table entry for sleep address;
                if (process not loaded in memory)
                        wake up swapper process (0);
                else if (awakened process is more eligible to run than
                                        currently running process)
                        set scheduler flag;
        }
        restore processor execution level to original level;
}
```

Figure 6.32. Algorithm for Wakeup

The iput algo releases a locked inode and awakens all processes waiting for the lock to become free .

Disk interrupt handler awakens processes waiting for I/O completion

Sleep and wakeup algorithm

- Wakeup algorithm does not cause a process to be scheduled immediately; it only makes the process eligible for scheduling.
- **Processes** may sleep on an event and wakeup soon as the locked resources (inode and buffers) or I/O completion of disk is temporary.
- **Processes** may sleep on an event that is not sure to happen and if so ,there should be some way to regain control and continue execution. Thus the kernel interrupts the sleeping process immediately by sending it a signal.

Sleep and wakeup algorithm

- To distinguish the type of sleep states,kernel sets the scheduling priority of the sleeping process when it enters the sleep state ,based on sleep priority parameter.
- It invokes the sleep algorithm with a priority value,based on its knowledge that the sleep event is sure to occur or not.
- If the priority > threshhold value : the process will not wakeup prematurely on receipt of a signal but will sleep until the event it is waiting for happens.
- If the priority < threshhold value : the process will awaken immediately on receipt of the signal.

Process Termination

- ***exit* system call**
  - process terminated by *exit* system call
    - enters the zombie status
    - relinquish resources (close all open files)
    - buffered output written to disk
    - dismantle its context except for its slot in the process table.

    Syntax:

    exit(status);

    - the value of status returned to parent process for its examination
  - Processes may call exit explicitly or implicitly(by startup routine) at the end of program.
  - The startup routine linked with all C programs call exit when the program returns from the main function.
  - kernel may invoke internally on receipt of uncaught signals. In this case, the value of status is the signal number.

Process Termination

- The system imposes notime limit on the execution of a process and processes frequntly exist for a long time.
- Process 0 (swapper) and process 1(init) exist throughout the lifetime of a system.
- getty processes : which monitor a terminal line, waiting for a user to login and special administrative processes.

Algorithm for Exit

```
algorithm exit
input : return code for parent process  output : none
{
    ignore all signals;
    if ( process group leader with associated control terminal )
    {
        send hangup signal to all members of process group;
        reset process group for all members to 0;
    }
    close all open files(internal version of algorithm close)
    release current directory(algorithm iput);
    release current(changed) root, if exists (algorithm iput);

free regions, memory associated with process(algorithm freereg,detachreg);
write accounting record;
  // To a global accounting file, containing  various run-time statistics such as user ID,CPU and memory  usage and amt of I/O for the process
    make process state zombie;
 // kernel saves exit status code and accumulated user and kernel execution time of the process and its decendents in the process table
    assign parent process ID of all child processes to be init process(1);
        if any children were zombie, send death of child signal to init;
    send death of child signal to parent process;
    context switch;
}
```

Control terminal is the terminal on which a user logs into the system and it controls processes that the user initiates from the terminal

Delete ,break,quit keys Ctrl+d presses

- The kernel resets the process group no to 0 for processes in the process group because it is possible that another process will get the process ID of the process that just exited and that it too will be a process group leader.

**User-level programs can later read the accounting file to gather various statistics, useful for performance monitoring and customer billing