UNIX INTERNALS

UNIT - 4

UNIX KERNEL AND ITS FILES

Reference Book: Maurice J. Bach, "The Design of the UNIX Operating System", Pearson Education, Pearson Prentice Hall, 2004.

Chapter 2 : Introduction to the kernel

Objectives : * To focus more on the kernel.
* To provide an Overview of kernel architecture
* To provide basic concepts and structures essential for understanding.

- Architecture of the UNIX operating system

→ The UNIX system supports the illusions that the file system has "places" and the respective processes have "life".

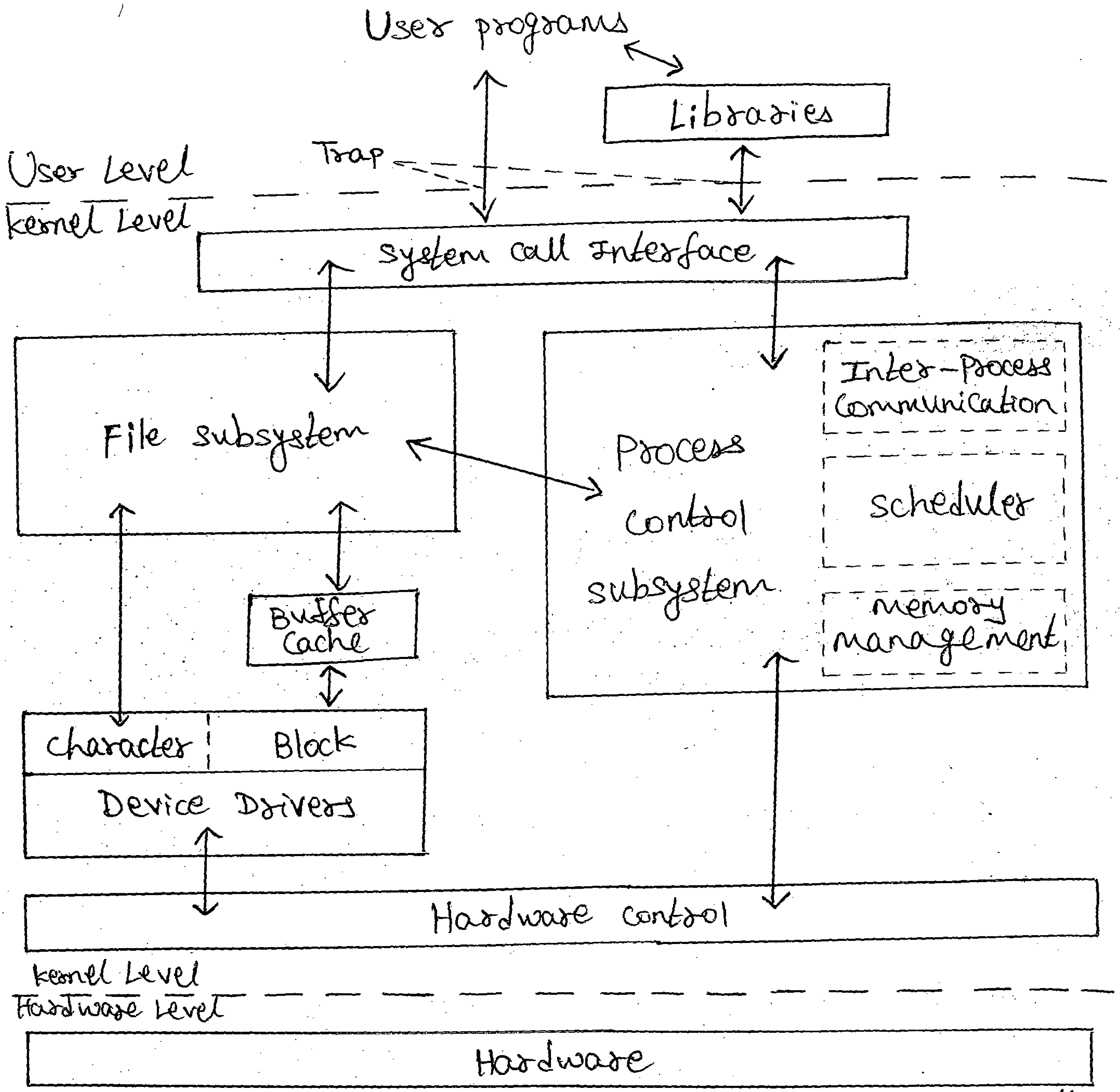
→ The two entities, files and processes, are the two central concepts in the UNIX system model.

→ Block Diagram of the system kernel, shows the various modules and their relationships to each other

* shows the file subsystem on the left and the process control subsystem on the right.
* The diagram serves as a useful logical view of the kernel.
* Three levels: User, kernel and Hardware.

→ The system call library interface represent the border between user programs and the kernel.

Block Diagram of the System kernel



- → System calls are the ordinary function calls in C programs and libraries map these function calls to the primitives needed to enter the operating system.
- → Assembly language programs may invoke system calls directly without a system call library.
  - — However, programs frequently use other libraries such as standard I/O library to provide a more sophisticated use of the system calls.
- → The libraries are linked with the programs at compile time and are thus part of the user program.

→ The diagram partitions the set of system calls into those that interact with the file subsystem and those that interact with the process control subsystem.

→ The file subsystem manages files, allocating file space, administering free space, controlling access to files and retrieving data for users.

→ Processes interact with the file subsystem via a specific set of system calls, such as open(), close(), read(), write(), stat() [query the attributes of a file], chown() [change the record of who owns the file] and chmod() [change the access permissions of a file].

→ The filesubsystem accesses file data using a buffering mechanism that regulates data flow between the kernel and secondary storage devices.

* The buffering mechanism interacts with block I/O device drivers to initiate data transfer to and from the kernel.

* Device drivers are the kernel modules that control the operation of peripheral devices.
  - Block I/O devices are random access storage devices, their device drivers make them appear to be random access storage devices to the rest of the system.
  - For example, a tape driver may allow the kernel to treat a tape unit as a random access storage device.

* File subsystem also interacts directly with "raw" I/O device drivers without the intervention of a buffering mechanism.

* Raw devices, sometimes called Character devices, include all devices that are not block devices.

→ The process control subsystem is responsible for process synchronization, interprocess communication, memory management and process scheduling.

* The file subsystem and the process control subsystem interact when loading a file into memory for execution.

* The process subsystem reads executable files into memory before executing them.

* Some of the system calls for controlling processes are:
  - (i) fork - Create a new process.
  - (ii) exec - Overlay the image of a program onto the running process.
  - (iii) exit - finish executing a process.
  - (iv) wait - Synchronize process execution with the exit of a previously forked process.
  - (v) brk - control the size of memory allocated to a process.
  - (vi) signal - Control process response to extraordinary events.

→ The memory management module controls the allocation of memory.

* If at any time the system does not have enough physical memory for all processes, the kernel moves them between main memory and secondary memory so that all processes get a fair chance to execute.

* Two policies for managing memory: Swapping and Demand Paging.
  - Swapper process is sometimes called the scheduler, because it "schedules" the allocation of memory for processes and influences the operation of the CPU scheduler.

→ The scheduler module allocates the CPU to processes. It schedules them to run in turn until they voluntarily relinquish the CPU while awaiting a resource (or) until the kernel preempts them when their recent run time exceeds a time quantum.

→ There are several forms of interprocess communication, ranging from asynchronous signalling of events to synchronous transmission of messages between processes.

> Finally, the hardware control is responsible for handling interrupts and for communicating with the machine.
  * Devices such as disks/terminals may interrupt the CPU while a process is executing.
  * The kernel may resume execution of the interrupted process after servicing the interrupt.
  * Interrupts are not serviced by special processes but by special functions in the kernel, called in the context of the currently running process.

Introduction to System Concepts.

This section gives an overview of some major kernel data structures and describes the function of modules shown in block diagram of the system kernel.

An Overview of the File Subsystem.

→ The internal representation of a file is given by an <u>inode</u>, which contains description of disk layout of the file data and other information such as the file owner, access permissions and access times

→ The term inode is the contraction of the term <u>index node</u> and is commonly used in literature on the UNIX system.

→ Every file has one inode, but it may have several names, all of which map into the inode. Each name is called a <u>link</u>.

→ when a process refers to a file by name, the kernel parses the file name one component at a time, checks that the process has permission to search the directories in the path and eventually retrieves the inode for the file. For example, if a process calls

open ("/fs2/mjb/rje/sourcefile", 1);

the kernel retrieves the inode for "/fs2/mjb/rje/sourcefile".

→ when a process creates a new file, the kernel assigns it an unused inode. Inodes are stored in the file system, but the kernel reads them into an in-core inode table when manipulating files.

→ The kernel contains two other data structures, the <u>file table</u> and the <u>user file descriptor table</u>.

* The file table is a global kernel structure, but the user file descriptor table is allocated per process.

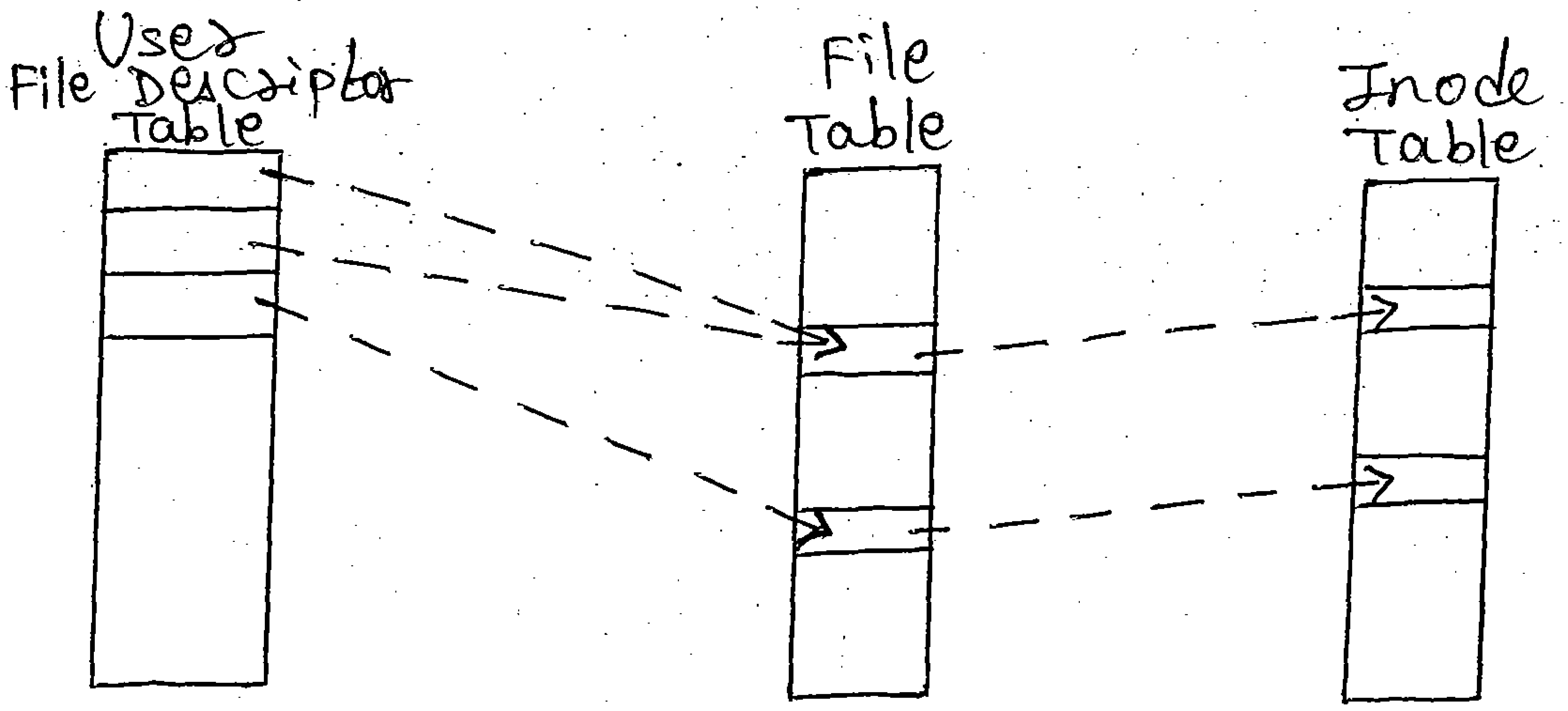
→ when a process opens/creates a file, the kernel allocates an entry from each table, corresponding to the file's inode.

> Entries in the three structures - User file descriptor table, file table and inode table (maintain the state of the file and the user's access to it).

> The file table keeps track of the byte offset in the file where the user's <u>next read/write</u> will start and the access rights allowed to the opening process.

> The user file descriptor table identifies all open files for a process.

> Diagram shows the tables and their relationship to each other.



<u>File Descriptors, File Table and Inode Table</u>

> The kernel returns a <u>file descriptor</u> for the <u>open</u> and <u>create</u> system calls, which is an index into the user file descriptor table.

> when executing <u>read</u> and <u>write</u> system calls, the kernel uses the file descriptor to access the user file descriptor table, follows pointers to the file table and inode table entries, and from the inode, finds the data in the file.

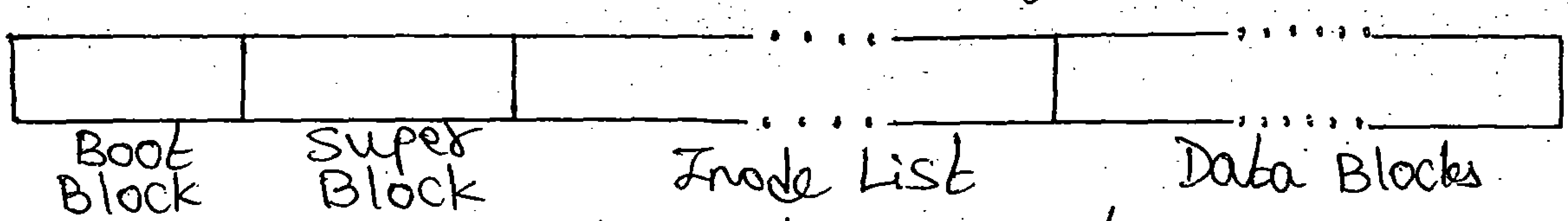
<u>File System</u> - A file system consists of a sequence of logical blocks, each containing 512, 1024, 2048 (or) any convenient multiple of 512 bytes, depending on the system implementation.

* The size of a logical block is homogeneous within a file system but may vary between different file systems in a system configuration.

* Using large logical blocks increases the effective data transfer rate between disk and memory, because the kernel can transfer more data per disk operation and therefore make fewer time-consuming operations.

* For example, reading 1K bytes from a disk in one read operation is faster than reading 512 bytes twice.

→ A file system has the following structure:



File system Layout

* The boot block occupies the beginning of a file system, typically the first sector and may contain the bootstrap code that is read into the machine to boot (or) initialize the operating system.
  - Although only one boot block is needed to boot the system, every file system has a possibly empty boot block.

* The super block describes the state of a file system — how large it is, how many files it can store, where to find free space on the file system, and other information.

* The inode list is a list of inodes that follows the super block in the file system.
  - Administrators specify the size of the inode list when configuring a file system.

The kernel references inodes by index into the inode list. One inode is the root inode of the file system: it is the inode by which the directory structure of the file system is accessible after the execution of the mount system call.

The data blocks start at the end of the inode list and contain file data and administrative data. An allocated data block can belong to one and only one file in the file system.

Processes

A process is the execution of a program and consists of a pattern of bytes that the CPU interprets as machine instructions called text, data and stack.

A process executes by following a strict sequence of instructions that is self-contained and does not jump to that of another process.

Processes communicate with other processes and with the rest of the world via system calls.

Practical, a process on a UNIX system is the entity that is created by the fork() system call.

Every process except process 0 is created when another process executes the fork().

The process that invoked the fork() is the Parent process and the newly created process is the child process.

The kernel identifies each process by its process number, called the process ID (PID).

Every process has : 1 parent process; Many child processes.

Process 0 is a special process that is created "by hand" when the system boots.

→ A user compiles the source code of a program to create an executable file, which consists of several parts:

(i) A set of "headers" that describe the attributes of the file.

(ii) The program text.

(iii) A machine language representation of data that has initial values when the program starts execution, and indication of how much space the kernel should allocate for this data, called bss. (Block Started by Symbol).

(iv) Other sections, such as symbol Table information.

→ The kernel loads an executable file into memory during an exec() and the loaded process consists of atleast three parts: $\underbrace{\text{Text, data and stack}}_{\text{Regions}}$.

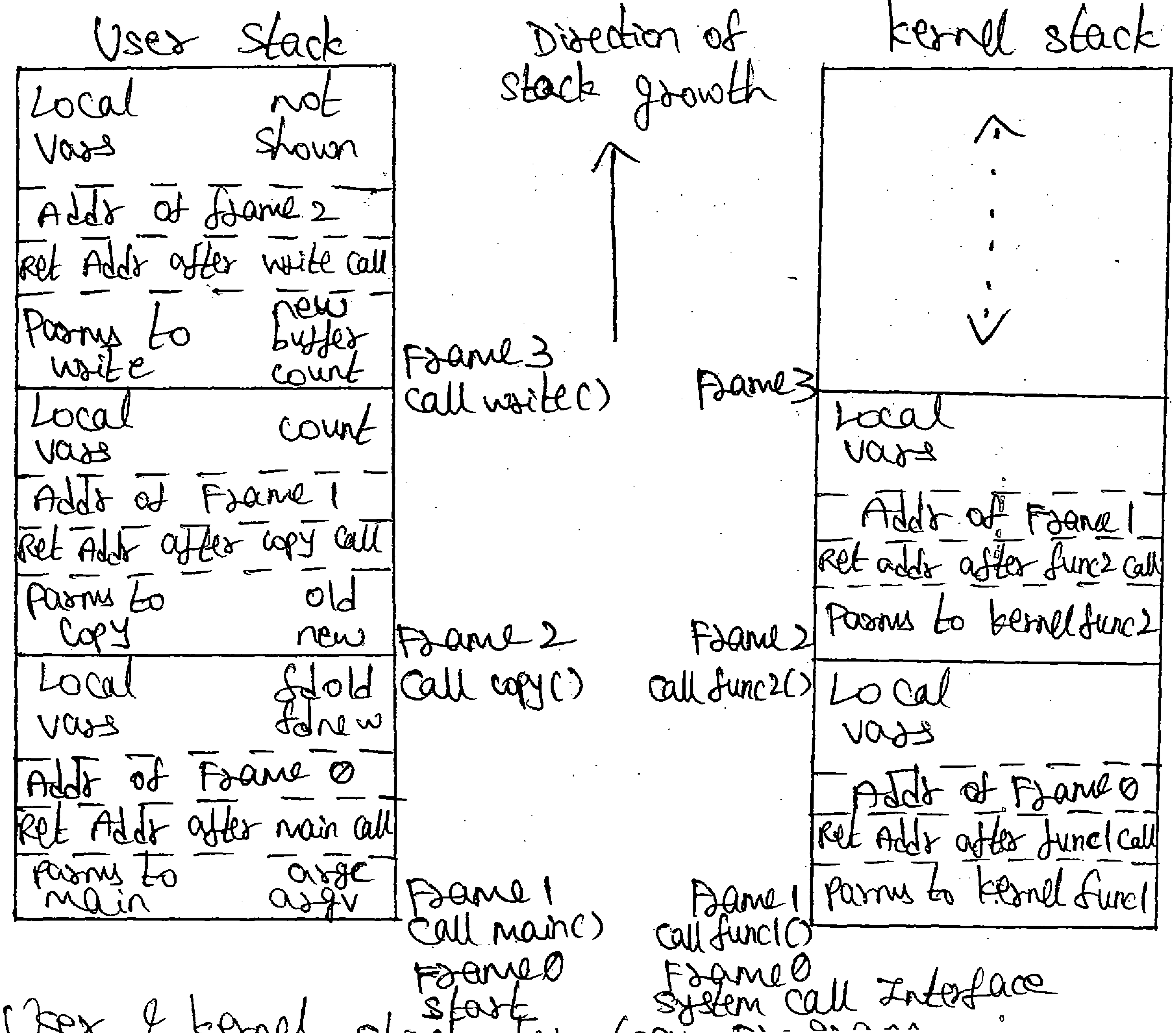
→ Because a process in the UNIX system can execute in two modes: kernel (or) User, it uses a separate stack for each mode

* The user stack contains the arguments, local variables & other data for functions executing in user mode.

* The left side of the figure, shows the user stack for a process when it makes the write() in the copy program.

* Process startup procedure (i.e, in Library) had called the function main with two parameters, pushing frame 1 onto the user stack; frame 1 contains space for the two local variables of main

* Main then called copy with two parameters, old and new and pushed frame 2 onto the user stack; frame 2 contains space for the local variable count.
* Finally, the process invoked the system call write by invoking the library function write.
* When the process executes the special instruction, it switches mode to the kernel, executes kernel code and uses the kernel stack.
* The kernel stack contains the stack frames for functions executing in kernel mode.
* The function and data entries on the kernel stack refer to functions and data in the kernel.
* The kernel stack of a process is null when the process executes in user mode.



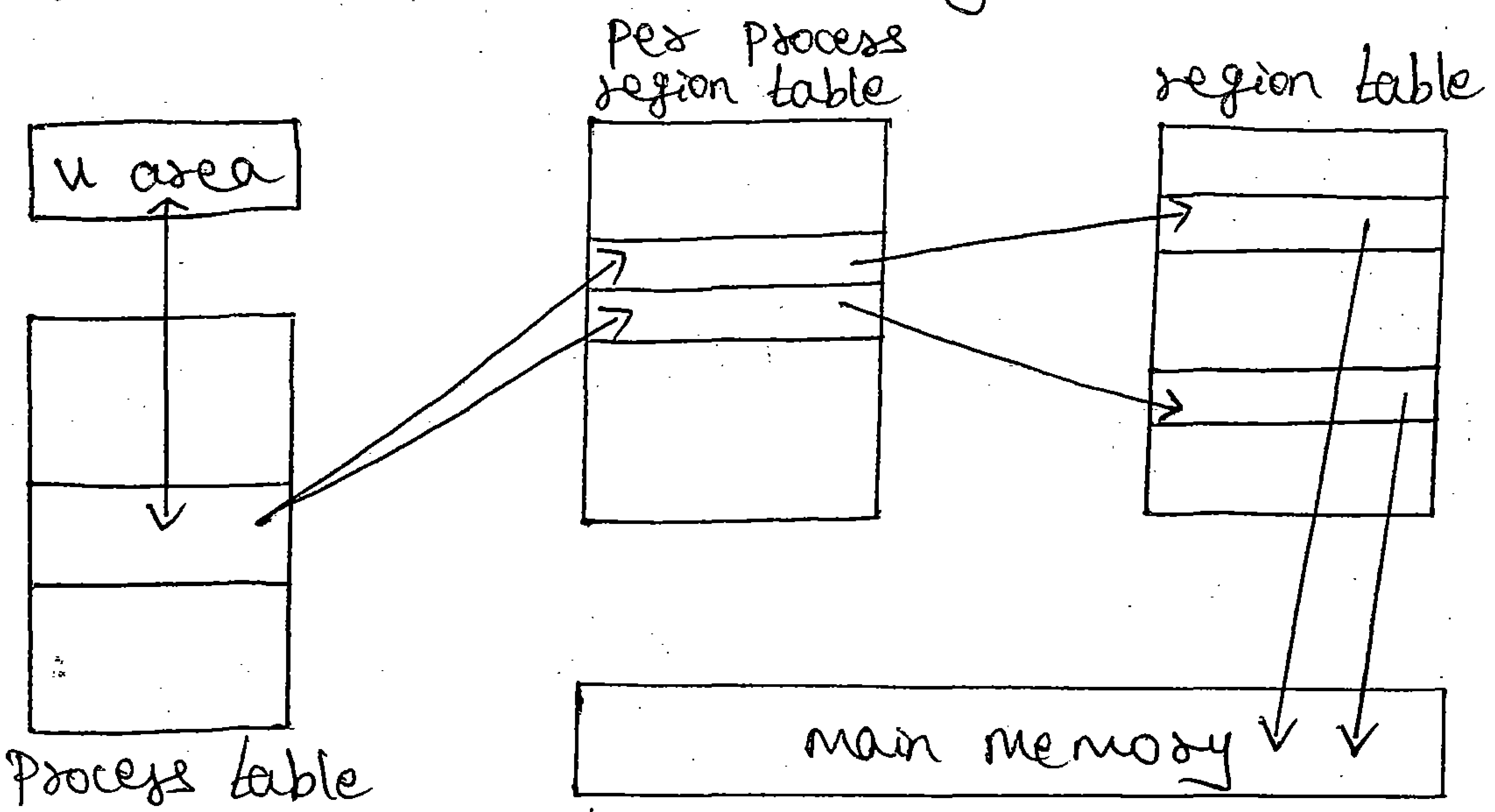
User & kernel stack ...

→ Every process has an entry in the <u>kernel process table</u> and each process is allocated a <u>u area</u> (user area) that contains private data manipulated only by the kernel.

→ The process table contains a <u>per process region table</u> whose entries point to entries in a <u>region table</u>.

→ A region is a contiguous area of a process's address space, such as text, data & stack.

→ Region table entries describe the attributes of the region, such as whether it contains text/data, whether it is shared or private, and where the "data" of the region is located in memory.



<u>Data structures for processes</u>

→ The process table points to a per process region table with pointers to the region table entries for the text, data & stack regions of the process.

→ The process table entry and the u area contain control and status information about the process.

→ Fields in the process table:

(i) A state field.
(ii) Identifiers : who owns the process (user IDs/UIDs)
(iii) An event descriptor, set when a process is suspended (in the sleep state).

(iv) A pointer to the process table slot of the currently executing process.
(vi) parameters of the current system call, return values.
(iii) File descriptors for all open files.
(iv) Internal I/o parameters.
(v) Current directory & current root.
(vi) Process & file size limits.

→ The u area contains information describing the process that needs to be accessible only when the process is executing. The important fields are mentioned above

→ The kernel can directly access fields of the u area of the executing process but not of the u area of other processes.

Context of a process.

The context of a process is its state, as defined by its text, the values of its global user variables and data structures.

The kernel does a context switch when it changes context from process A to process B; it changes execution mode from user to kernel (or) from kernel to user.

The kernel saves enough information so that it can later resume execution of the interrupted process and services the interrupt in kernel mode.

process states

→ The lifetime of a process can be divided into a set of states.

→ Following states of a process are:

(i) The process is currently executing in user mode.

(ii) The process is currently executing in kernel mode.

(iii) The process is not executing, but it is ready to run.

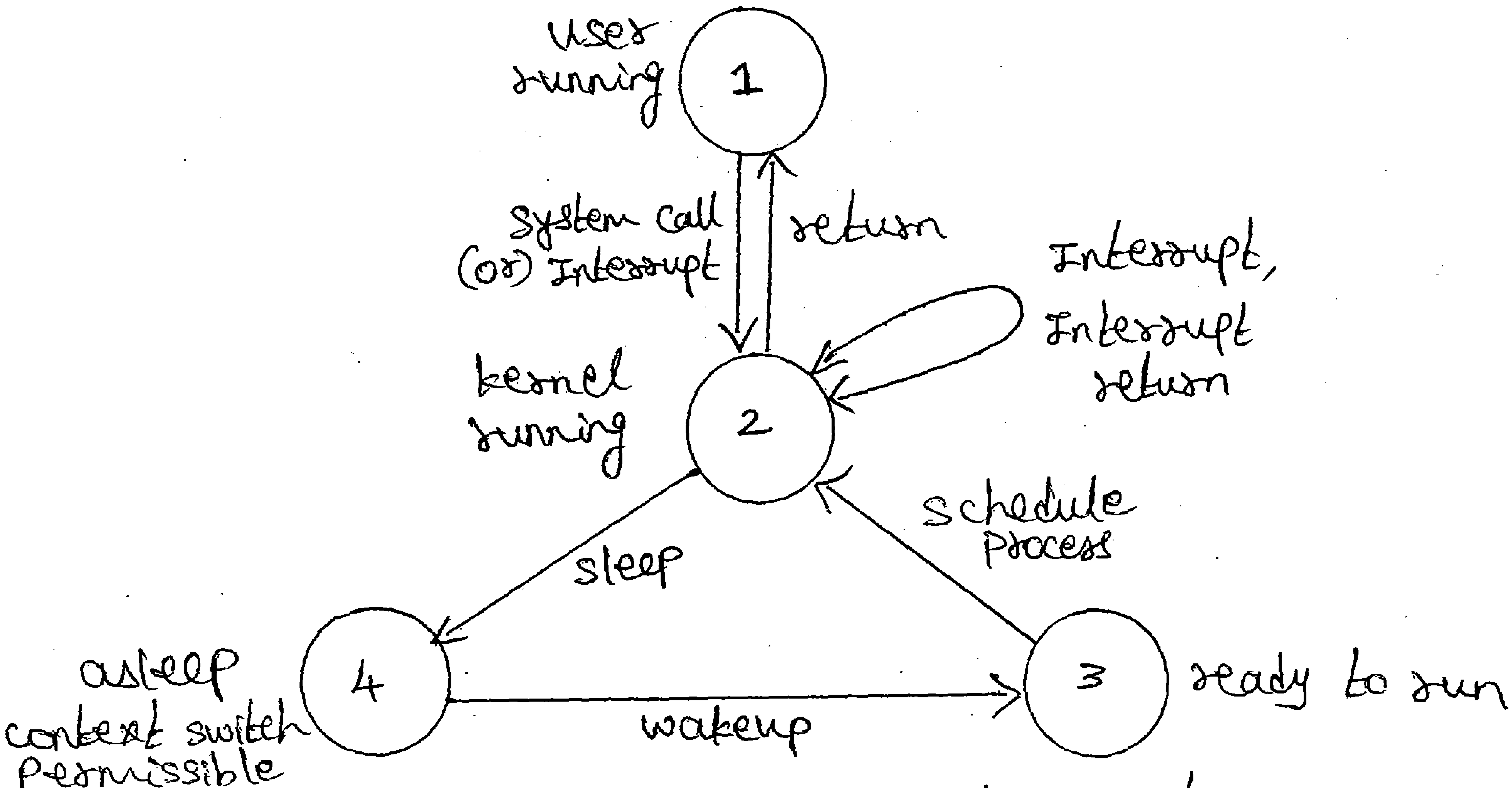
(iv) The process is sleeping.

- The process puts itself to sleep when it can no longer continue executing, such as when it is waiting for I/O to complete.

---

state transitions.

→ A state transition diagram is a directed graph whose nodes represent the states a process can enter and whose edges represent the events that cause a process to move from one state to another.



Process states and Transitions

→ The kernel allows a context switch only when a process moves from the state "kernel running" to the state "asleep in memory".

> Processes running in kernel mode cannot be preempted by other processes; therefore the kernel is sometimes said to be non-preemptive.
  - In user mode, process can be preempted.

> The kernel maintains consistency of its data structures because it is non-preemptive, thereby solving the <u>mutual exclusion problem</u> - making sure that critical sections of code are executed by at most one process at a time.

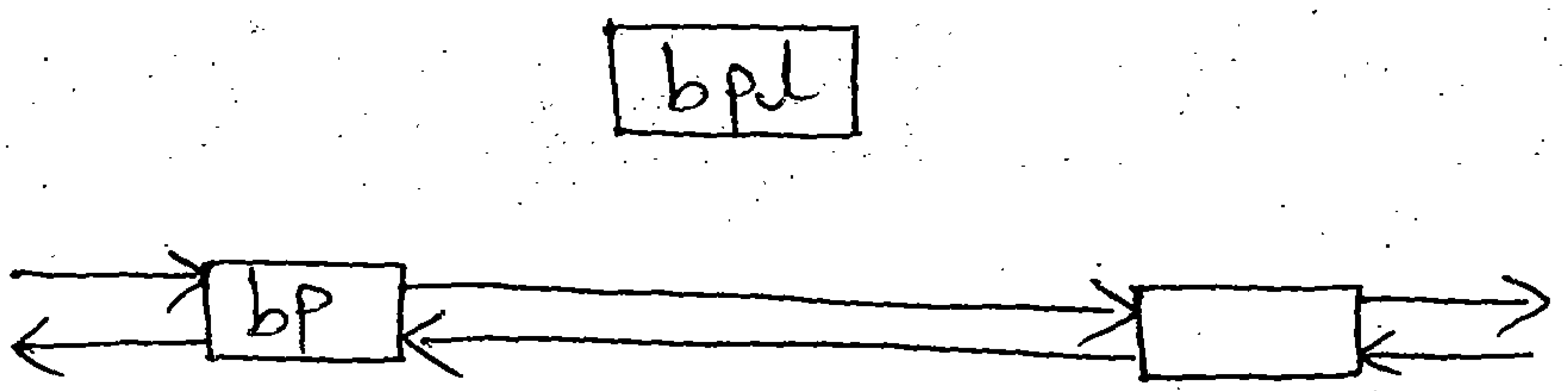
> For instance, consider the sample code to put a data structure, whose address is in the pointer <u>bp1</u>, onto a doubly linked list after the structure whose address is in <u>bp</u>.

* If the system allowed a context switch while the kernel executed the code fragment, the following situation could occur.
* suppose the kernel executes the code until the comment and then does a context switch.
* The doubly linked list is in an inconsistent state: the structure <u>bp1</u> is half on and half off the linked list.
* If a process were to follow the forward pointers, it would find <u>bp1</u> on the linked list, but if it were to follow the back pointers, it would not find <u>bp1</u>.
* The UNIX system prevents such situations by disallowing context switches when a process executes in kernel mode.

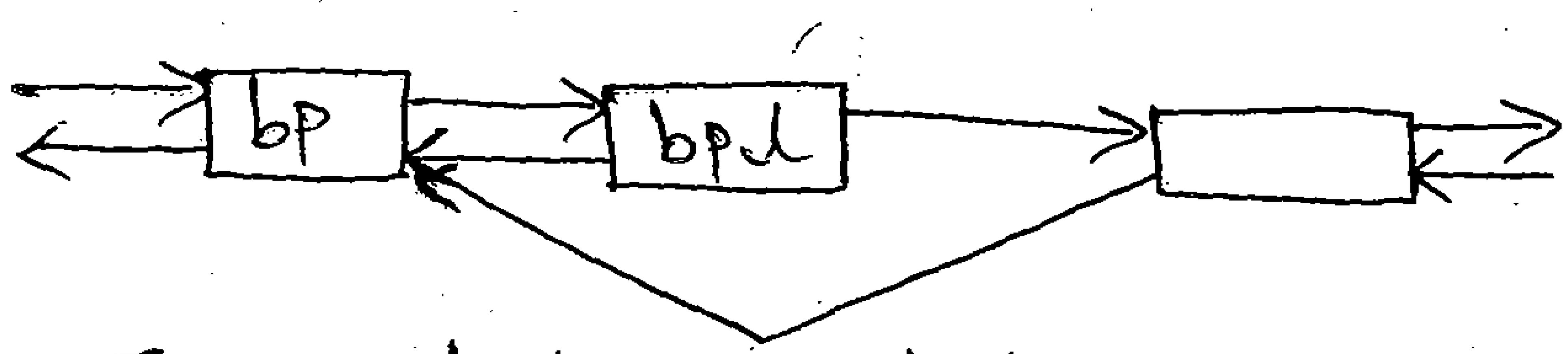
```
struct queue {

} *bp, *bp1;
bp1 -> forp = bp -> forp;
bp1 -> backp = bp;
bp -> forp = bp1;
/* Consider possible context switch here */
bp1 -> forp -> backp = bp1;
```

Sample Code Creating Doubly Linked List



Placing bp1 on doubly linked list



Incorrect Linked List because of Context Switch

---

sleep and wakeup

→ Processes go to sleep because they are awaiting the occurrence of some event, such as waiting for I/O completion from a peripheral device, waiting for a process to exit, waiting for system resources to become available and so on.

→ Processes are said to "sleep on an event", meaning that they are in the sleep state until the event occurs, at which time they wake up and enter the state "ready to run".

- Many processes can simultaneously sleep on an event.
  - when an event occurs, all processes sleeping on the event, wake up.
- when a process wakes up, it follows the state transition from the "sleep" state to the "ready-to-run" state, where it is eligible for later scheduling.
- sleeping processes do not consume CPU resources. The kernel does not constantly check to see that a process is still sleeping, but waits for the event to occur and awakens the process.

Example: A process executing in kernel mode must sometimes lock a data structure in case it goes to sleep at a later stage; processes attempting to manipulate the locked structure must check the lock and sleep if another process owns the lock.

The kernel implements such locks in the following manner:

```
while (condition is true)
        sleep (event: the condition becomes false);
    set condition true;
```

It unlocks the lock and awakens all processes asleep on the lock in the following manner:

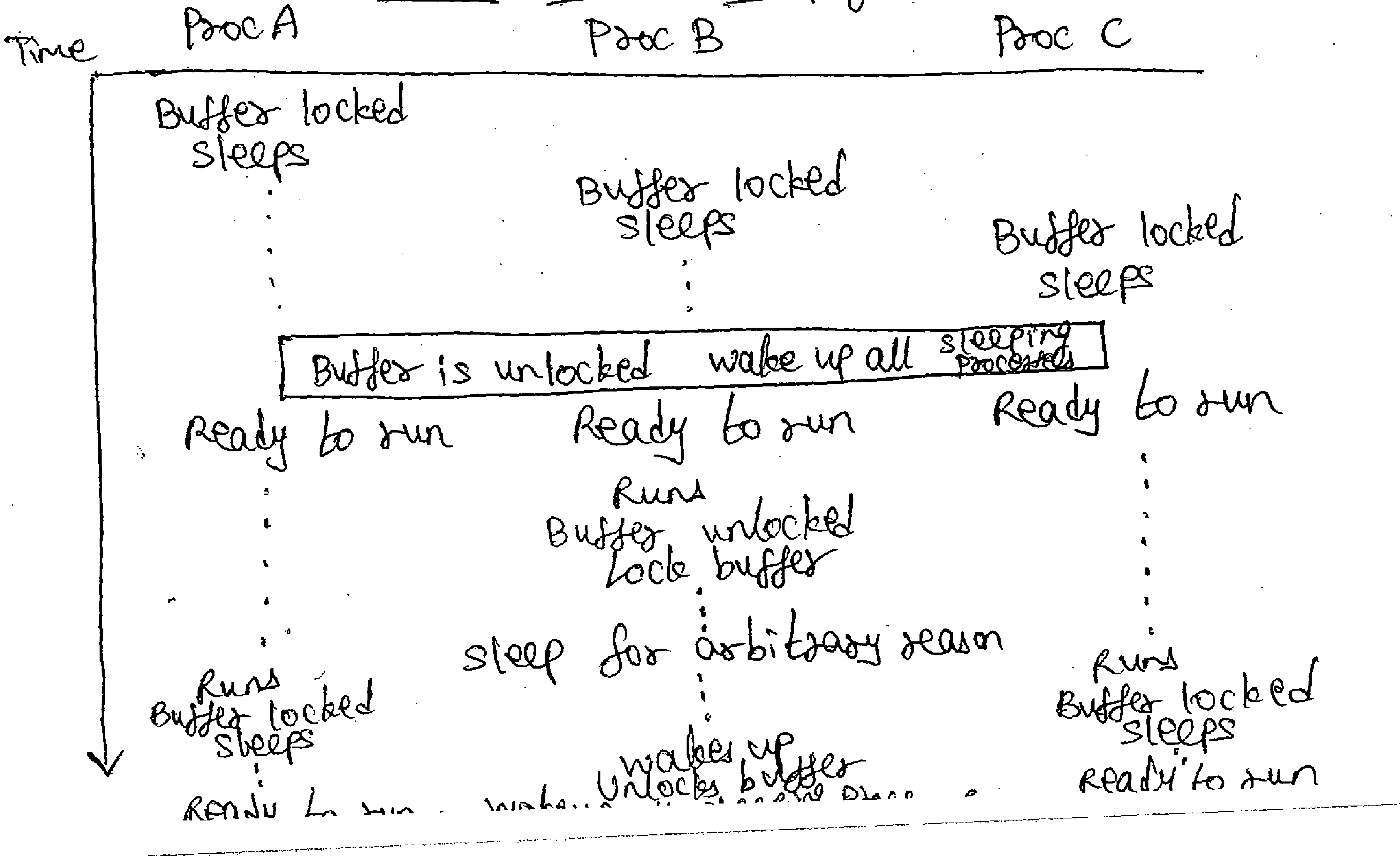
```
set condition false;
wakeup (event: the condition is false);
```

Example: A scenario where three processes A, B and C contend for a locked buffer.

* The sleep condition is that the buffer is locked.
* The processes execute one at a time, find the buffer locked, and sleep on the event that the buffer becomes unlocked.

* Eventually, the buffer is unlocked, and all processes wakeup and enter the state "ready to run".
* The kernel eventually chooses one process, say 'B' to execute first (may be based on priority).
* Process B executes the "while" loop, finds that the buffer is unlocked, sets the buffer lock and proceeds.
* If process B later goes to sleep again before unlocking the buffer (waiting for I/O completion), the kernel can schedule other processes to run.
* If it chooses process A, process A executes the "while" loop, finds that buffer is locked & goes to sleep again; similarly for process C.
* Eventually, process B awakens & unlocks the buffer allowing either process A (or) C to gain access to the buffer.
* Thus, "while-sleep" loop ensures that at most one process can gain access to a resource (buffer)

Multiple Processes sleeping on a Lock



Kernel Data structures

- Most kernel data structures occupy fixed-size tables rather than dynamically allocated space.
- The advantage of this approach is that the kernel code is simple, but it limits the number of entries for a data structure to the number that was originally configured when generating the system
- If kernel run out of entries for a data structure, it report an error to the requesting user.

On the otherhand, if it is unlikely to run out of table space, the extra table space may be wasted because it cannot be used for other purposes.

Algorithms typically use simple loops to find free tables entries, a method that is easier to understand and sometimes more efficient than more complicated allocation schemes.

System Administration.

Administrative processes are loosely classified as those processes that do various functions for the general welfare of the user community.

functions includes: disk formatting, creation of new file systems, repair of damaged file systems, kernel debugging and others.

There is no much difference between administrative processes and user processes.

They use same set of system calls available to the general community.

They are distinguished from general user processes only in the permission rights and privileges they are allowed.

→ Example: file permission modes may allow administrative processes to manipulate files otherwise off-limits to general users.

→ Internally, the kernel distinguishes a special user called the <u>super user</u>, endowing it with special privileges.

→ A user may become a 'superuser' by going through a login-password sequence (or) by executing special programs.

→ The kernel does not recognize a separate class of administrative processes.

Chapter 4: Internal Representation of Files

Objectives:
* To describe the internal structure of files in the UNIX system.
* To examine the inode and how the kernel manipulates it.
* To examine the internal structure of regular files and how the kernel reads and writes their data.
* To investigate the structure of directories.
* To describe the structure of super block.
* To present the algorithms for assignment of disk inodes and disk blocks to files; other file types in the system (Pipes & device files)

File System Algorithms

- The algorithms described here, occupy the layer above the buffer cache algorithms.
- The algorithm 'iget' returns a previously identified inode, possibly reading it from disk via the buffer cache.
- The algorithm 'iput' releases the inode.
- The algorithm 'bmap' sets kernel parameters for accessing a file.

The algorithm 'namei' converts a user-level pathname to an inode, using the algorithms iget, iput and bmap.

Algorithms 'alloc' and 'free' allocate and free disk blocks for files respectively.

Algorithms 'ialloc' and 'ifree' assign and free inodes for files respectively.

Buffer Allocation algorithms.

* getblk – To allocate a buffer for a disk block.
* brelse – Releasing a buffer.
* bread – Reading a Disk Block.
* breada – Block Read Ahead.
* bwrite – Writing a Disk Block.

Lower Level File System Algorithms

<table>
<tr><td>namei</td><td rowspan="2">alloc free</td><td rowspan="2">ialloc ifree</td></tr>
<tr><td>iget iput bmap</td></tr>
<tr><td colspan="3">buffer allocation algorithms</td></tr>
<tr><td colspan="3">getblk brelse bread breada bwrite</td></tr>
</table>

File System Algorithms

(1) Inodes

→ Inode exist in a static form on disk and the kernel reads them into an incore inode to manipulate them.

→ Disk inodes consists of the following fields:

(i) File owner Identifier.

* Individual Owner
* Group owner.

(ii) File type

* Types: Regular, ~~[illegible]~~ directory, character, block (or) FIFO (pipes).

(iii) File access permissions.

* Read, write and execute
(4) (2) (1)

7 7 7
↓ Owner ↓ Group ↳ other

Ex:- 1) -rwx-rwx-rwx → 777
2) -rwx-r~~wx~~-r--
(O) (G) (other)
↳ 764

(iv) File access times

* File was created.
* File was last modified
* File was last accessed.

(v) Number of links to the file

* Representing the numbers of names the file has in the directory hierarchy.

(vi) Table of contents for the disk addresses of data in a file.

(vii) File size

* In terms of bytes: 1B, 1KB, 1MB, 1GB etc...

→ Example:

owner mno
group os
type regular file
perms rwxr-xr-x (755)
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 inode does not specify the path name(s) that access the file.

> The in-core copy of the inode contains the following fields in addition to the fields of the disk node:

(i) The status of the in-core inode, indicating whether
- the inode is locked,
- a process is waiting for the inode to become unlocked,
- the file is a mount point

(ii) The logical device number of the file system that contains the file.

(iii) The inode number.

(iv) pointers to other in-core inodes.

(v) A reference count, indicating the number of instances of the file that are active. (such as when 'opened').

Accessing Inodes

→ **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 */
        }
        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 (ex: reference count to 1);
      return (inode);
    }
}
```

→ Formula to compute the byte offset of the inode in the block:

((inode number − 1) modulo (number of inodes per block)) * size of disk inode
(multiply)

- Releasing Inodes
- <u>Releasing Inodes Algorithm</u>

```
algorithm iput
input: pointer to in-core inode
output: none
{
    lock inode if not already locked;
    decrement inode reference count;
    if (reference count == 0)
    {
      if (inode link count == 0)
      {
        free disk blocks for file; (algorithm free)
        set file type to 0;
        free inode (algorithm ifree);
      }
      if (file accessed or inode changed or file changed)
            update disk inode;
      put inode on free list;
    }
    release inode lock;
}
```


<u>Releasing an Inode</u>

Structure of a Regular File

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

→ 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 (i.e; the file occupied a linear sequence of disk blocks)

.....	File A	File B	File C	.....

40 50 60 70

Block Addresses

.....	File A	Free	File C	File B

40 50 60 70 81

Block Addresses

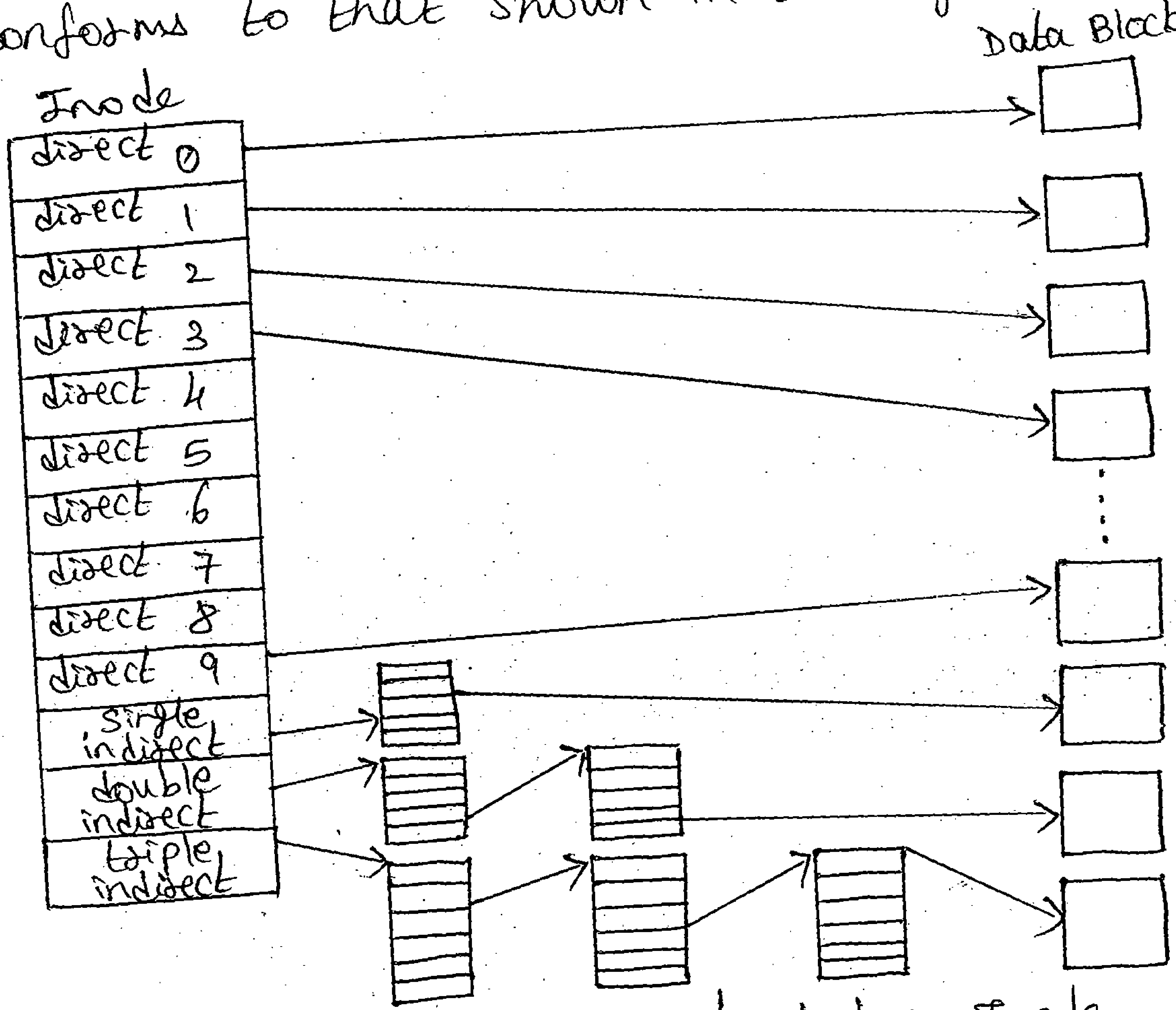
Allocation of Contiguous Files and Fragmentation of Free space

→ Example: Suppose a user creates three files, A, B and C, each consisting of 10 disk blocks of storage (length) and suppose the system allocated storage for the three files contiguously

* If the user then wishes to add 5 blocks of data to the middle file, 'B', the kernel would have to copy (move) file B to a place in the file system that had room for 15 blocks of storage.

* The disk blocks previously occupied by file B's data would be unusable except for files smaller than 10 blocks.

To keep the inode structure small yet still allow large files, the table of contents of disk blocks conforms to that shown in the figure below:



Direct and Indirect Blocks in Inode.

* The blocks marked "direct" in the figure, contain the numbers of disk blocks that contain real data.
* The block marked "single indirect" refers to a block that contains a list of direct block numbers.
* The block marked "double indirect" contains a list of indirect block numbers.
* The block marked "triple indirect" contains a list of double indirect block numbers.

Byte Capacity of a File - 1k Bytes Per Block

10 direct blocks with 1 k 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

→ Algorithm 'bmap' for converting a file byte offset into a physical disk block

```
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 byte offset;
    calculate start byte in block for I/O;   // o/p 2
    calculate number of bytes to copy to user;   // o/p 3
    check if read-ahead applicable, mark inode;   // o/p 4
    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;
    }
}
```

7) Directories

- Directories are the files that give the file system its hierarchical structure.
- They play an important role in conversion of a file name into an inode number.
- A directory is a file whose data is a sequence of entries, each consisting of an inode number and the name of a file contained in the directory.
- A path name is a null terminated character string divided into separate components by the slash ("/") character.
- Each component except the last must be the name of a directory; last component is a non-directory file.

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	checklist
176	92	fsdblb
192	84	config
208	1432	getty
224	0	crash
240	95	mkfs
256	188	inittab

Directory Layout for /etc

→ The kernel stores data for a directory just as it stores data for an ordinary file, using the inode structure & levels of direct and indirect blocks.

→ The access permissions of a directory have the following meaning:

* read permission on a directory allows a process to read a directory.
* write permission allows a process to create new directory entries (or) remove old ones (via the creat, mknod, link and unlink system calls).
* execute permission allows a process to search the directory for a file name.

Conversion of a path Name to an Inode

→ Algorithm for conversion of a path Name to an Inode

```
algorithm namei
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 & 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    /* component not in directory */
        return (no inode);
}
return (working inode);
}
```

Super Block

The super block consists of the following fields:

* The size of the file system.
* The numbers 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

→ Algorithm 'ialloc' for Assigning New Inodes

```
algorithm ialloc
input: file system
output: locked inode
{
  while (not done)
  {
     if (super block locked)
     {
        sleep (event super block becomes 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 (algorithm 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;

}

/* these are inodes in super block inode list */
get inode number from super block inode list;
get inode (algorithm iget);
if (inode not free after all)
{
    write inode to disk;
    release inode (algorithm iput);
    continue;
}
/* inode is free */
initialize inode;
write inode to disk;
decrement file system free inode count;
return (inode);
}

Algorithm for Freeing Inode

gorithm ifree
put: file system inode number
tput: none

    increment file system free inode count;
    if (super block locked)
        return;
    if (inode list full)
    { if (inode number less than remembered inode for search)
            set remembered inode for search = input inode number;
    }else

store inode number in inode list;

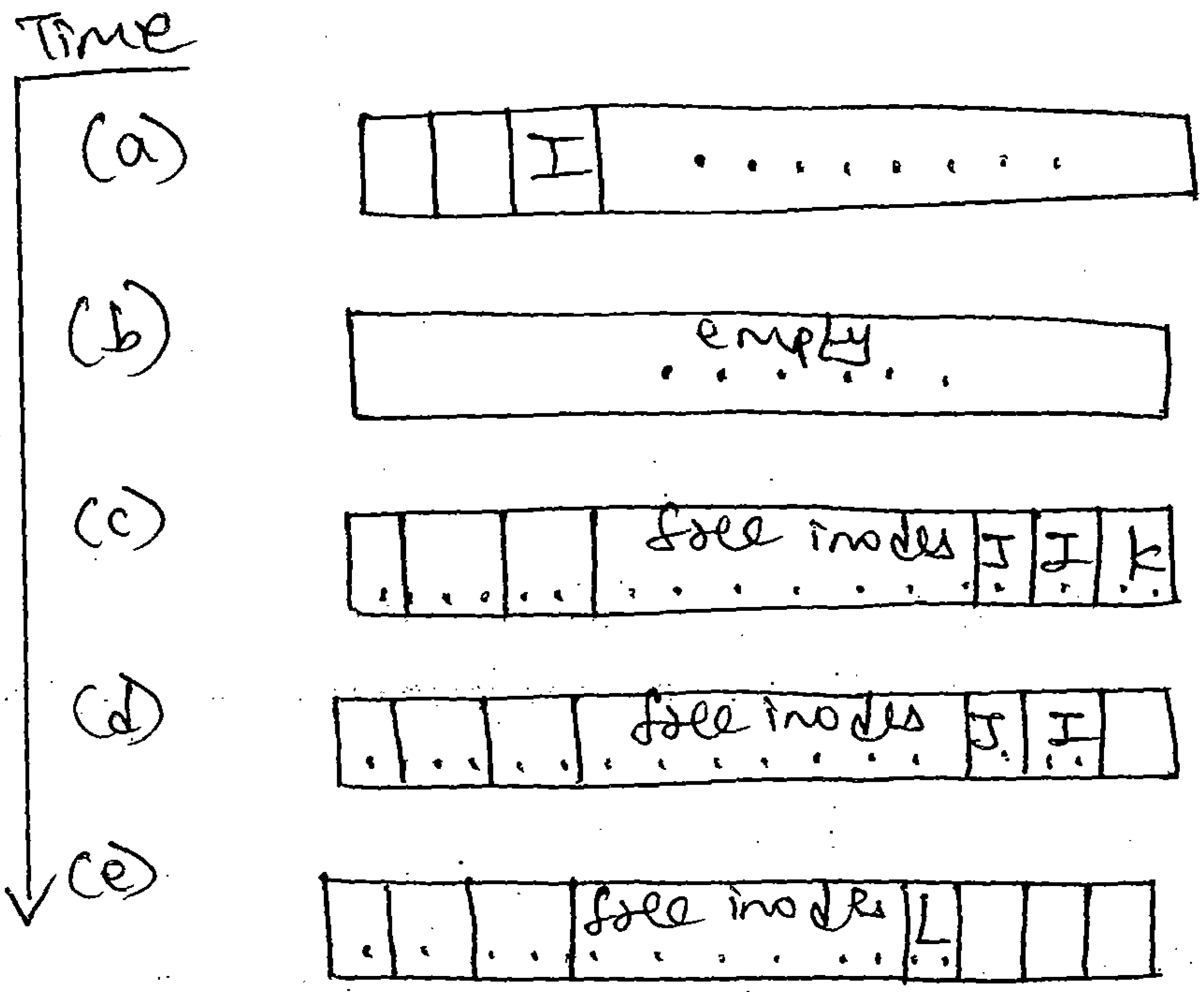
return;

}

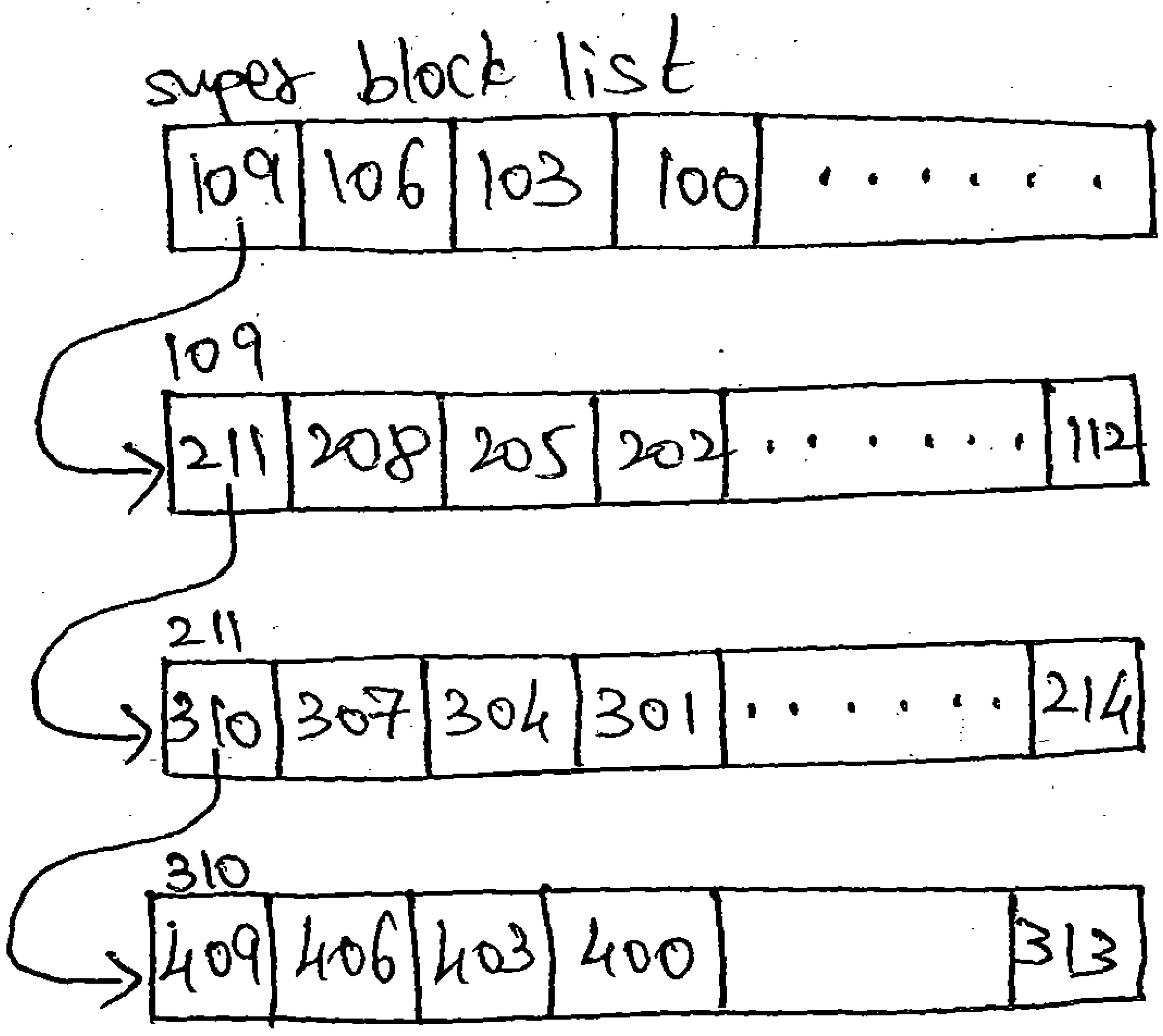
→ Race condition in Assigning Inodes

Process A	Process B	Process C
Assigns inode I from super block		
sleeps while reading inode (a)		
	Tries to assign inode from super block	
	super block empty (b)	
	search for free inodes on disk, puts inode I in super block (c)	
Inode I in core, Does usual activity		
	completes search, assign another inode (d)	
		Assigns inode I from super block. I is in use!
		Assign another inode (e)

Time



Allocation of Disk Blocks



Linked List of Free Disk Block Numbers

Algorithm 'alloc' for Allocating Disk Block

```
rithm alloc
put: file system number
put: buffer for new block
   while (super block locked)
        sleep (event super block not locked);
```

remove block from super block free list;
if (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 processes (event super block 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;
return buffer;
}

→ The figure, shows a sequence of alloc and free
operations, starting with one entry on the super
block free list.

* The kernel frees block 949 and places the block number on the free list.
* It then allocates a block and removes block number 949 from the free list.
* Finally, it allocates a block and removes block number 109 from the free list.
* Because the super block free list is now empty, the kernel replenishes the list by

copying in the contents of block 109, the next link on the linked list.

* Figure (d) shows the full super block list and the next link block, block 211.

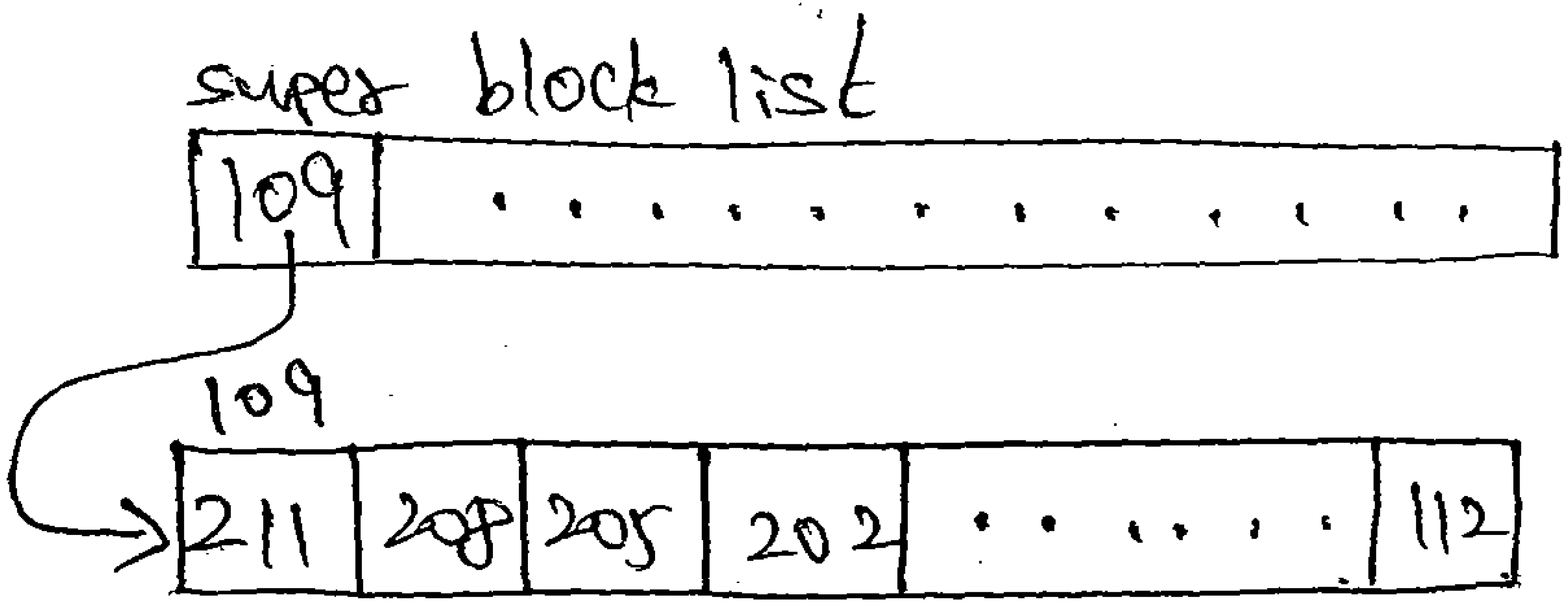
⇒ There are three reasons for the different treatment:

(1) The kernel can determine whether an inode is free by inspection. If the type field is clear, the inode is free. The kernel requires an external method to identify free blocks and traditional implementations have used a linked list.

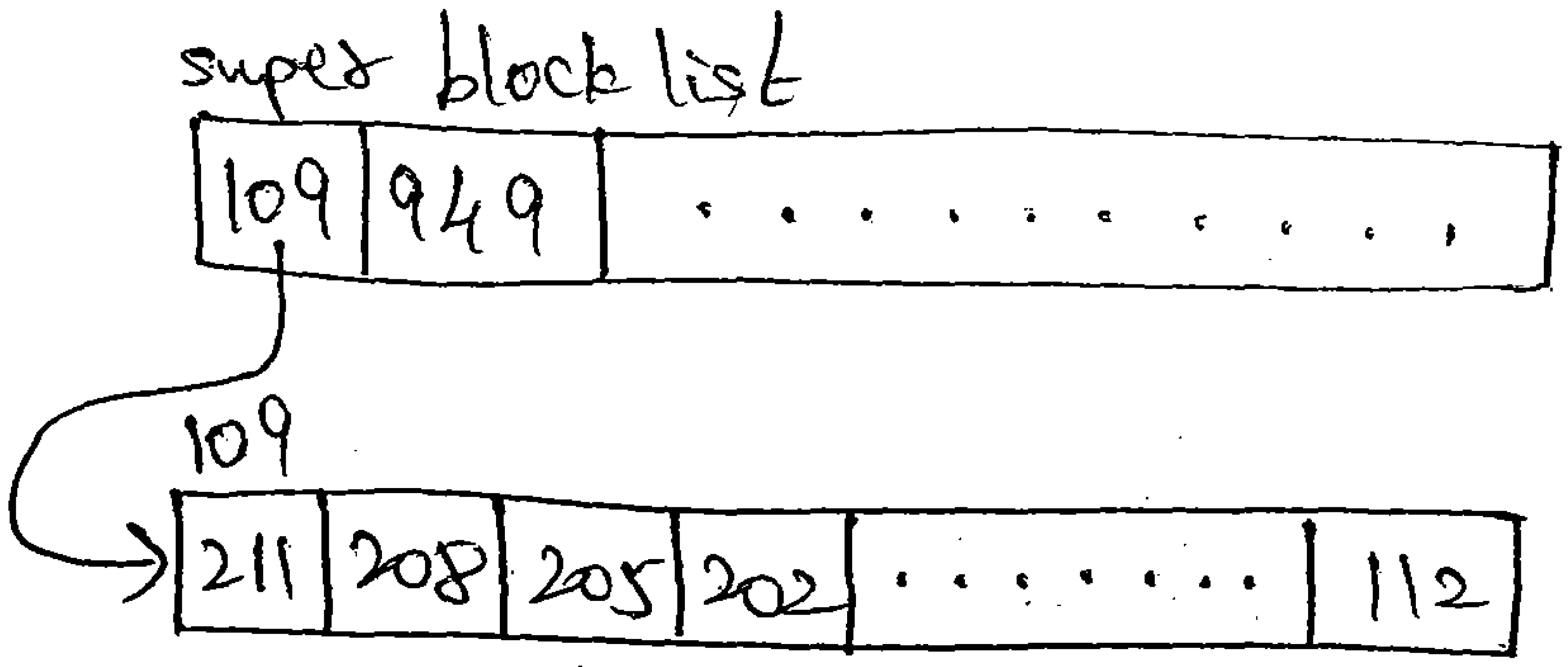
(2) Disk blocks lend themselves to the use of linked lists. A disk block easily holds large list of free block numbers. But inodes have no convenient place for bulk storage of large lists of free inode numbers.

(3) Users tend to consume disk block resources more quickly than they consume inodes, so the apparent lag in performance when searching the disk for free inodes is not as critical as it would be for searching for free disk blocks.

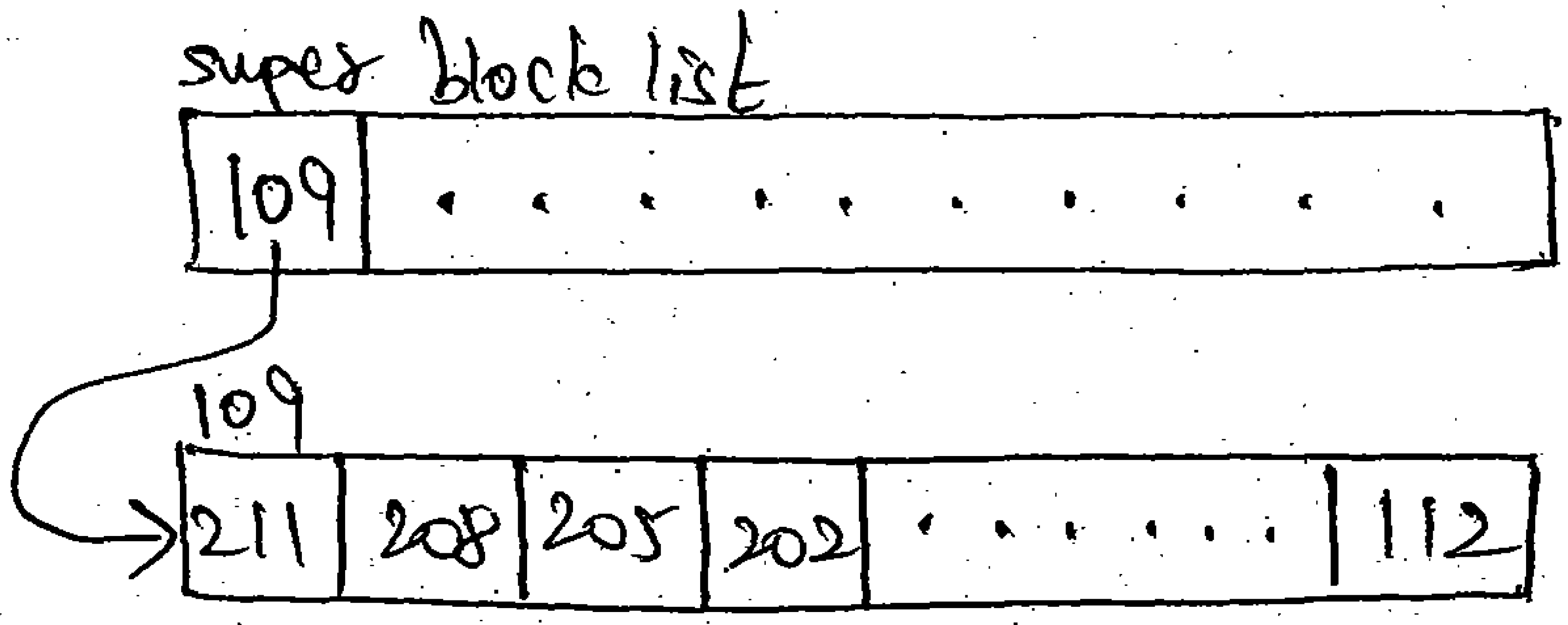
Requesting and Freeing Disk blocks



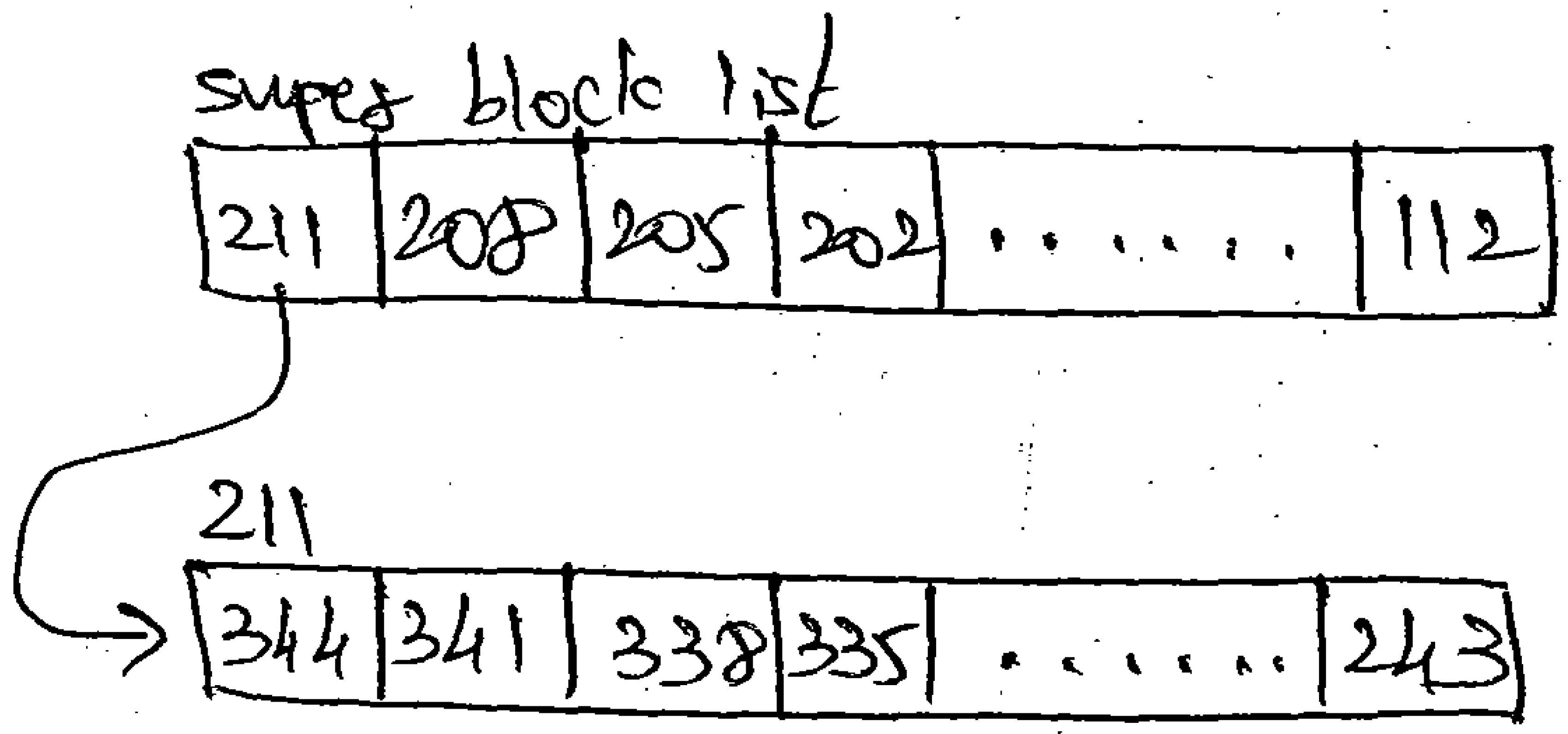
(a) Original configuration.



(b) After freeing block number (949)



(c) After assigning block number (949)



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

Requesting and Freeing Disk Blocks

---

(11) Other File Types

→ The UNIX system supports two other file types: pipes and special files.

→ A pipe, sometimes called a fifo (first-in-first-out), differs from a regular file, that its data is transient.

→ Once data is read from a pipe, it cannot be read again.

→ Also, the data is read in the 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.

→ The last file types in the UNIX system are 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.

→ The inode contains two numbers known as the major and minor device numbers.

→ The major number indicates a device type such as terminal / disk.

→ The minor number indicates the unit number of the device
