IInd Batch

Section A $(2 \times 10 = 20)$

1. What is system calls? Explain the system call flow with the help of a block diagram.

A System call is a mechanism used by an application for requesting a service from the operating system. Examples of the services provided by the operating system are allocation and deallocation of memory, reporting of current date and time etc. These services can be used by an application with the help of system calls. Many of the modern OS have hundreds of system calls. For example Linux has 319 different system calls. Example: open, close, read, write, kill, wait etc.

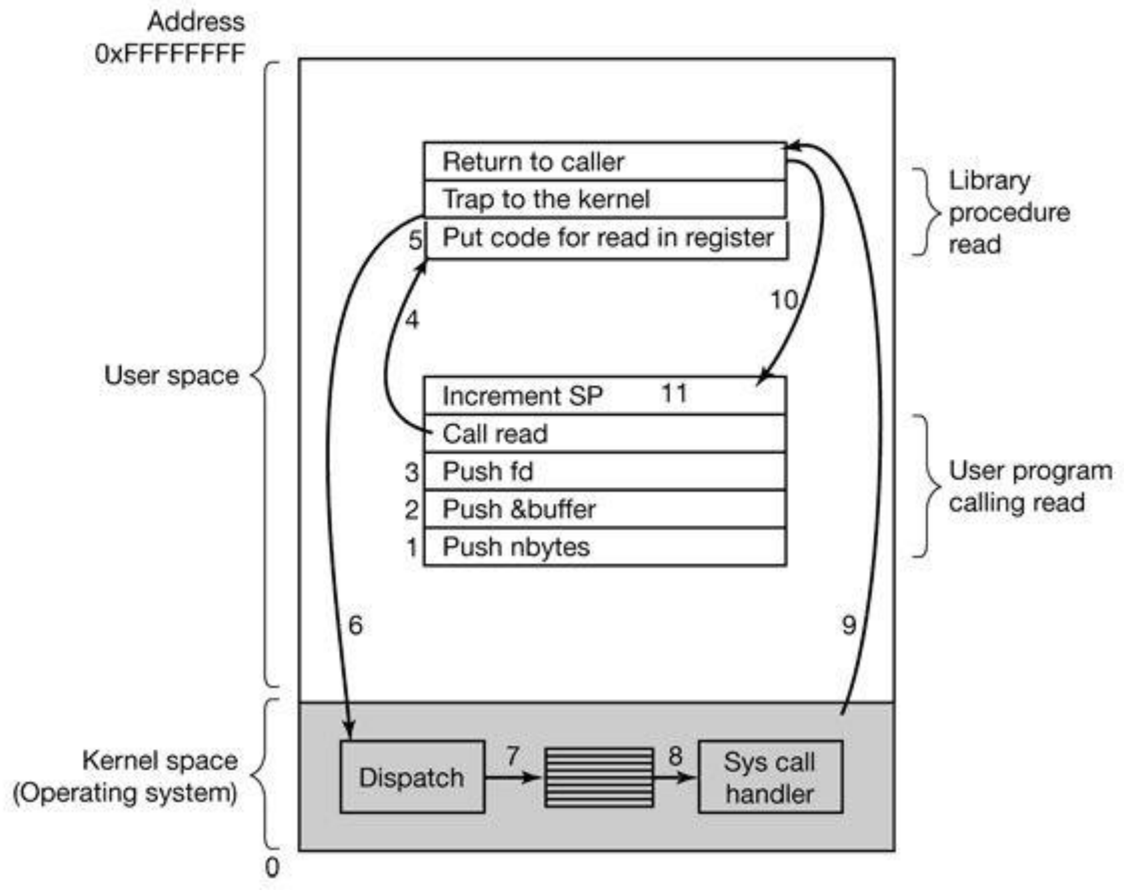


Fig: The 11 steps in making the system call read(id, buffer, nbytes)

System calls are performed in a series of steps. To make this concept clearer, let us examine the read call. In preparation for calling the read library procedure, which actually makes the read system call, the calling program first pushes the parameters onto the stack, as shown in steps 1-3 in figure. The first and third parameters are called by value, but the second parameter is passed by reference, meaning that the address of the buffer (indicated by &) is passed, not the contents of the buffer. Then comes the actual call to the library procedure (step 4). This instruction is the normal procedure call instruction used to call all procedures. The library procedure, possibly written in assembly language, typically puts the system call number in a place where the operating system expects it, such as a register (step 5). Then it executes a TRAP instruction to switch from user mode to kernel mode and start execution at a fixed address within the kernel (step 6). The kernel code that starts examines the system call number and then dispatches to the correct system call handler, usually via a table of pointers to system call handlers indexed on system call number (step 7). At that point the system call handler runs (step 8). Once the system call handler has completed its work, control may be returned to the user-space library procedure at the instruction following the TRAP instruction (step 9). This procedure then returns to the user program in the usual way procedure calls return (step 10). To finish the job, the user program has to clean up the stack, as it does after any procedure call (step 11). Assuming the stack grows downward, as it often does, the compiled code increments the stack pointer exactly enough to remove the parameters pushed before the call to read. The program is now free to do whatever it wants to do next.

OR 1. What do you mean by file systems? What are the major difference between file system interfaces and file system implementation? Explain.

We have three essential requirements for long-term information storage:

- It must be possible to store a very large amount of information.
- The information must survive the termination of the process using it.
- Multiple processes must be able to access the information concurrently.

The usual solution to all these problems is to store information on disks and other external media in units called files. Files are managed by the operating system. How they are structured, named, accessed, used, protected, and implemented are major topics in operating system design. As a whole, that part of the operating system dealing with files is known as the file system.

File system implementation deals with:

- How files and directories are stored?
- How disk space is managed?
- How to make everything work efficiently and reliably?

The main objective of file system implementation is:

- To describe the details of implementing local file systems and directory structures
- To describe the implementation of remote file systems
- To discuss block allocation and free-block algorithms and trade-offs

Different allocation methods are used in file system implementation. Some of the allocation methods are:

- Contiguous allocation
- Linked allocation

- Indexed allocation

File system interface provides applications with various system calls and commands such as open, write, read, seek, etc. Since main memory is usually too small, the computer system must provide secondary storage to back up main memory. The file system provides the mechanism for storage of and (multiple) access to both data and programs residing on the disks. Under this we describe following topics:

- Access method
- Directory structure
- File system mounting
- File sharing
- Protection

2. Write short notes on:

a. Disk Scheduling Algorithms

b. Error Handling and Formatting

c. File Operations

Disk Scheduling Algorithms:

There are two objectives for any disk scheduling algorithm:

- Maximize the throughput - the average number of requests satisfied per time unit.
- Minimize the response time - the average time that a request must wait before it is satisfied.

Some of the disk scheduling algorithms are explained below:

- FCFS (First Come First Serve): It performs operations in order requested. There is no reordering of work queue. There is no starvation: every request is serviced. It gives poor performance.
- SSTF (Shortest Seek Time First): After a request, goes to the closest request in the work queue, regardless of direction. It reduces total seek time compared to FCFS.
- SCAN: It goes from the outside to the inside servicing requests and then back from the outside to the inside servicing requests.
- LOOK: It is like SCAN but stops moving inwards (or outwards) when no more requests in that direction exist.

- C-SCAN (circular scan): It moves inwards servicing requests until it reaches the innermost cylinder and then jumps to the outside cylinder of the disk without servicing any requests.
- C-LOOK: It moves inwards servicing requests until there are no more requests in that direction, then it jumps to the outermost outstanding requests.

Error Handling and Formatting:

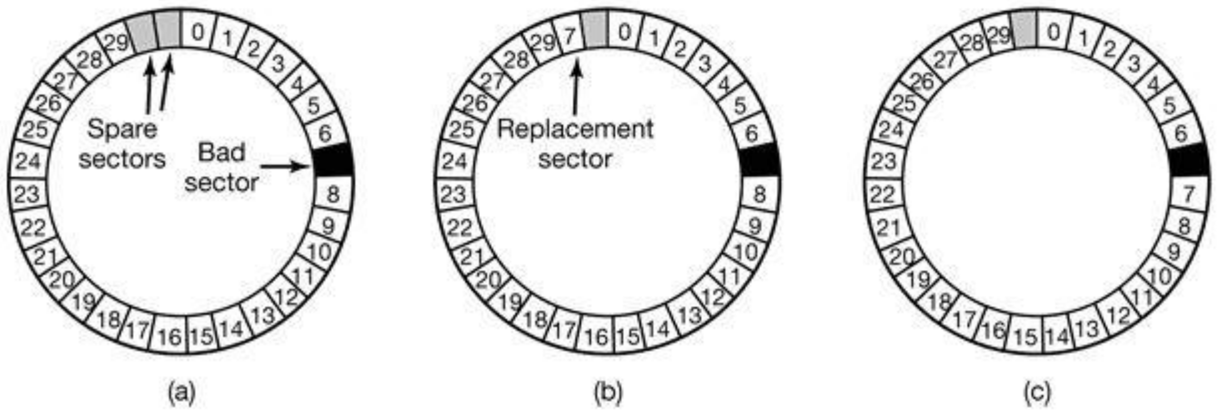


Fig (a): A disk track with a bad sector

Fig (b): Substituting a spare for the bad sector

Fig (c): Shifting all the sectors to bypass the bad one

Errors can also develop during normal operation after the drive has been installed. The first line of defense upon getting an error that the ECC cannot handle is to just try the read again. Some read errors are transient, that is, are caused by specks of dust under the head and will go away on a second attempt, if the controller notices that it is getting repeated errors on a certain sector, it can switch to a spare before the sector has died completely. In this way, no data are lost and the operating system and user do not even notice the problem. Usually, the method of Fig (b) has to be used since the other sectors might now contain data. Using the method of Fig (c) would require not only rewriting the preambles, but copying all the data as well.

A hard disk consists of a stack of aluminum, alloy, or glass platters 5.25 inch or 3.5 inch in diameter (or even smaller on notebook computers). On each platter is deposited a thin magnetizable metal oxide. After manufacturing, there is no information whatsoever on the disk. Before the disk can be used, each platter must receive a low-level format done by software. The format consists of a series of concentric tracks, each containing some number of sectors, with short gaps between the sectors. The format of a sector is shown in figure below:

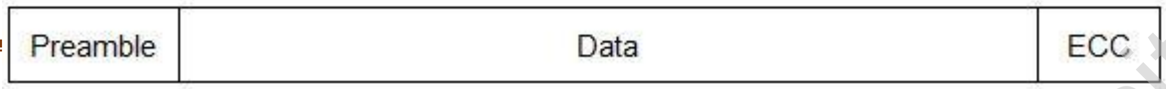


Fig. A disk sector

File Operations:

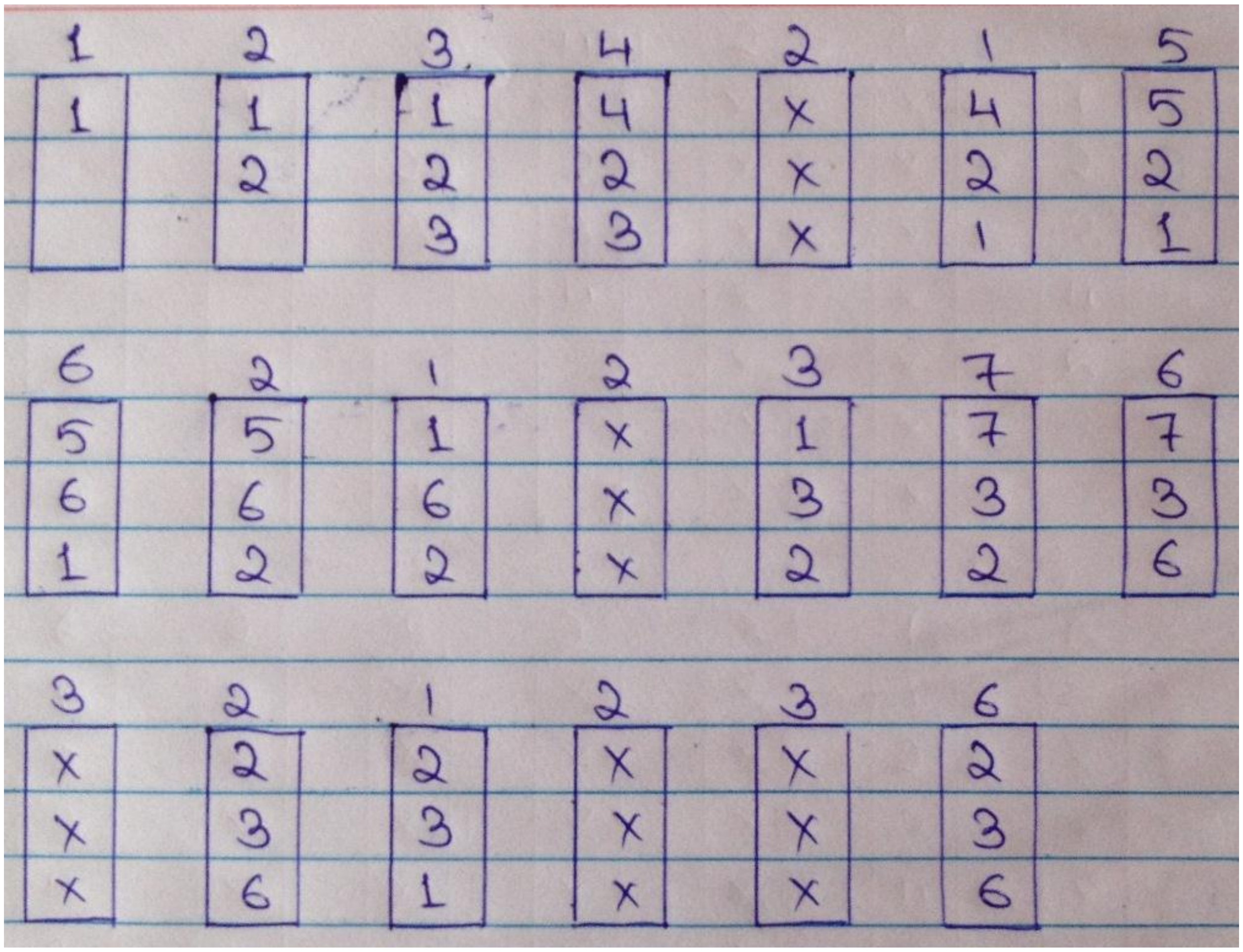
Files exist to store information and allow it to be retrieved later. Different systems provide different operations to allow storage and retrieval. Below is a discussion of the most common system calls related to files:

- Create: The file is created with no data. The purpose of the call is to announce that the file is coming and to set some of the attributes.
- Delete: When the file is no longer needed, it has to be deleted to free up disk space. There is always a system call for this purpose.
- Open: Before using a file, a process must open it. The purpose of the open call is to allow the system to fetch the attributes and list of disk addresses into main memory for rapid access on later calls.
- Close: When all the accesses are finished, the attributes and disk addresses are no longer needed, so the file should be closed to free up internal table space. Many systems encourage this by imposing a maximum number of open files on processes. A disk is written in blocks, and closing a file forces writing of the file's last block, even though that block may not be entirely full yet.
- Read: Data are read from file. Usually, the bytes come from the current position. The caller must specify how much data are needed and must also provide a buffer to put them in.
- Write: Data are written to the file, again, usually at the current position. If the current position is the end of the file, the file's size increases. If the current position is in the middle of the file, existing data are overwritten and lost forever.
- Append: This call is a restricted form of write. It can only add data to the end of the file. Systems that provide a minimal set of system calls do not generally have append, but many systems provide multiple ways of doing the same thing, and these systems sometimes have append.
- Seek: For random access files, a method is needed to specify from where to take the data. One common approach is a system call, seek, that repositions the file pointer to a specific place in the file. After this call has completed, data can be read from, or written to, that position.
- Get attributes: Processes often need to read file attributes to do their work. For example, the UNIX make program is commonly used to manage software development projects consisting of many source files. When make is called, it examines the modification times of all the source and object files and arranges for the minimum number of compilations required to bring everything up to date. To do its job, it must look at the attributes, namely, the modification times.

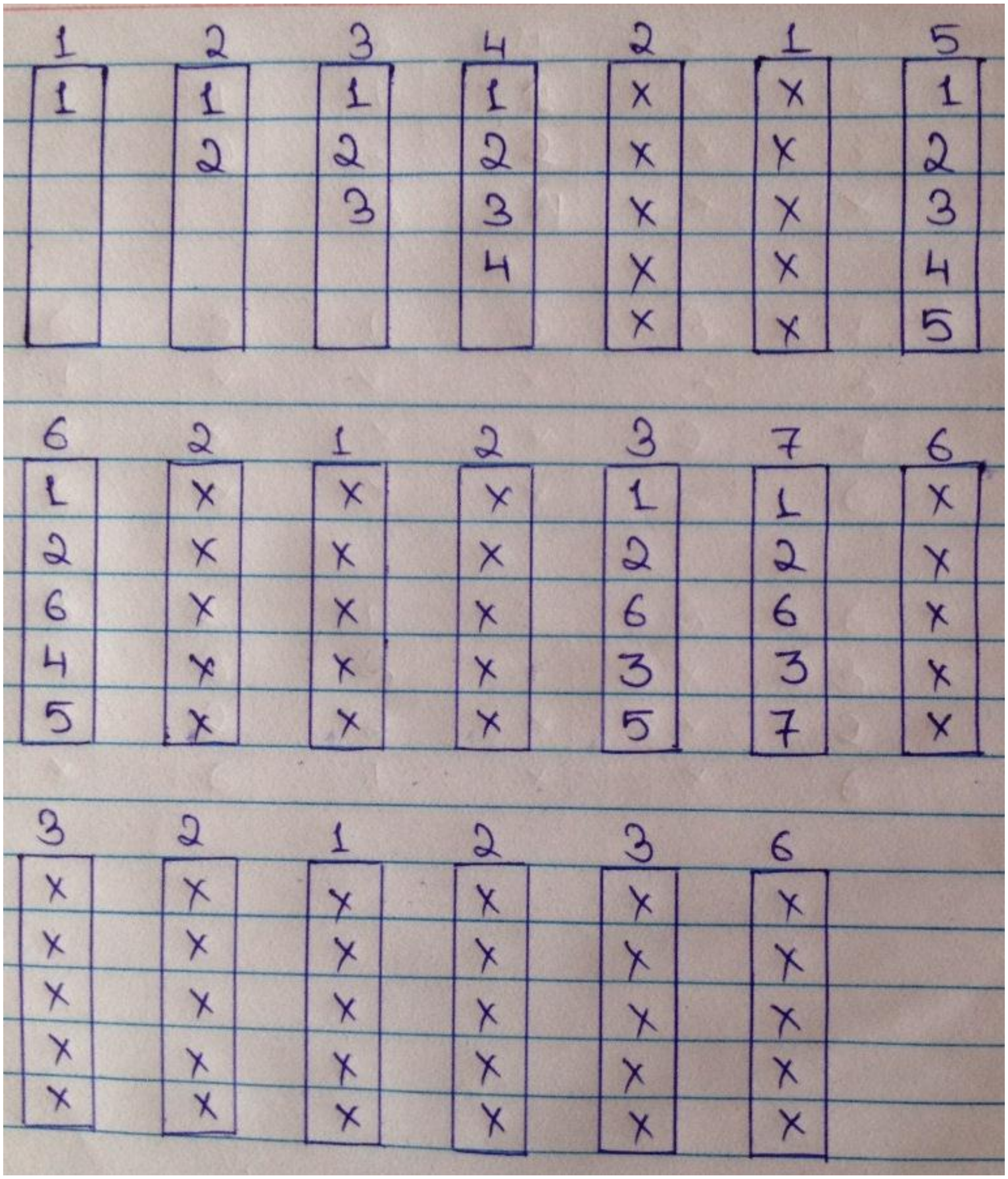
- Set attributes: Some of the attributes are user settable and can be changed after the file has been created. This system call makes that possible. The protection mode information is an obvious example. Most of the flags also fall in this category.
- Rename: It frequently happens that a user needs to change the name of an existing file. This system call makes that possible. It is not always strictly necessary, because the file can usually be copied to a new file with the new name, and the old file then deleted.

3. Consider the following page reference string: 1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 6. How many page faults would occur for the LRU replacement, FIFO replacement, & optimal replacement algorithms? Assuming three, five, or seven frames? Remember all frames are initially empty, so your first unique pages will all cost one fault each.

<u>LRU replacement for three frames:</u>



LRU replacement for five frames:

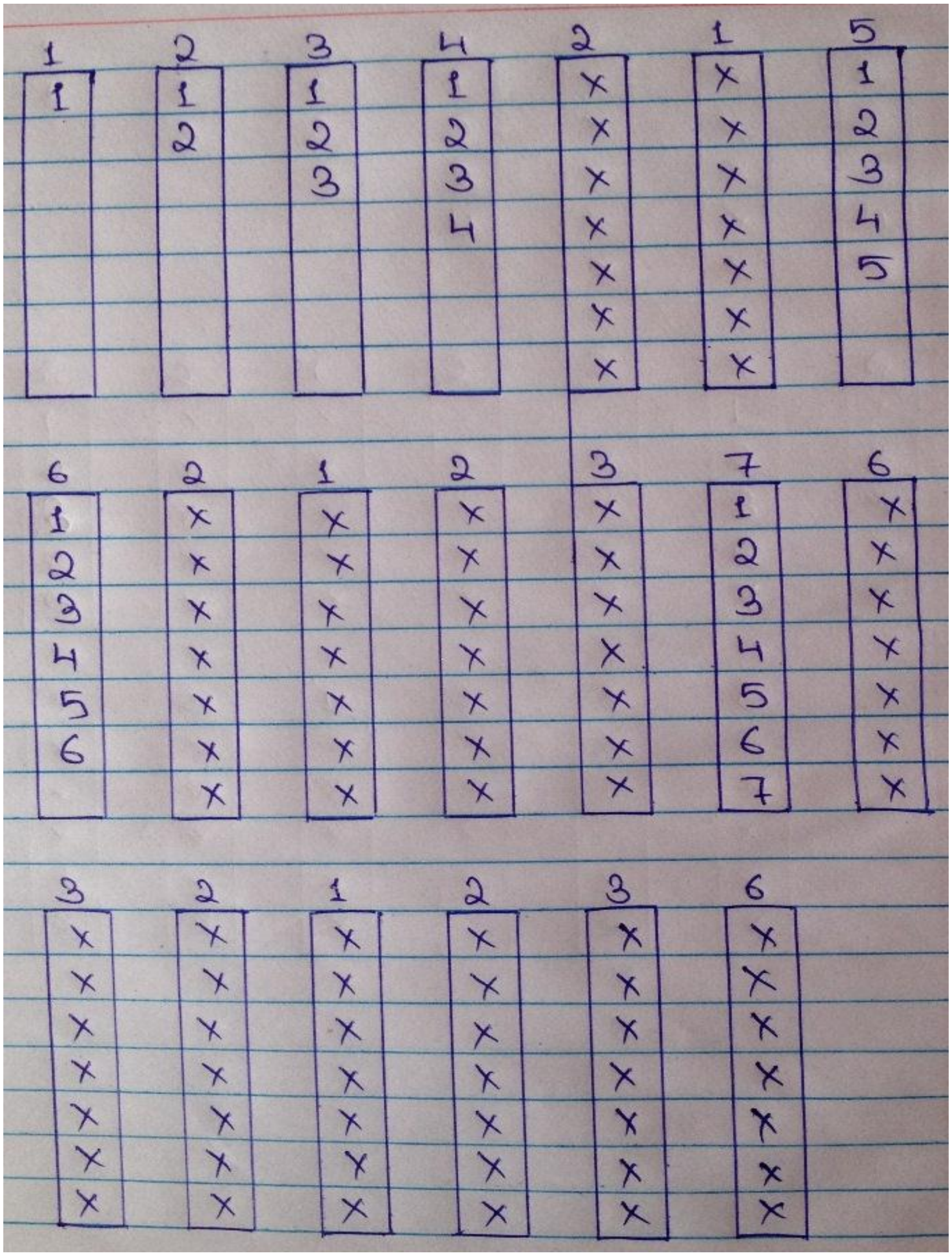


1	2	3	4	2	1	5
1	1	1	1	X	X	1
	2	2	2	X	X	2
		3	3	X	X	3
			4	X	X	4
				X	X	5

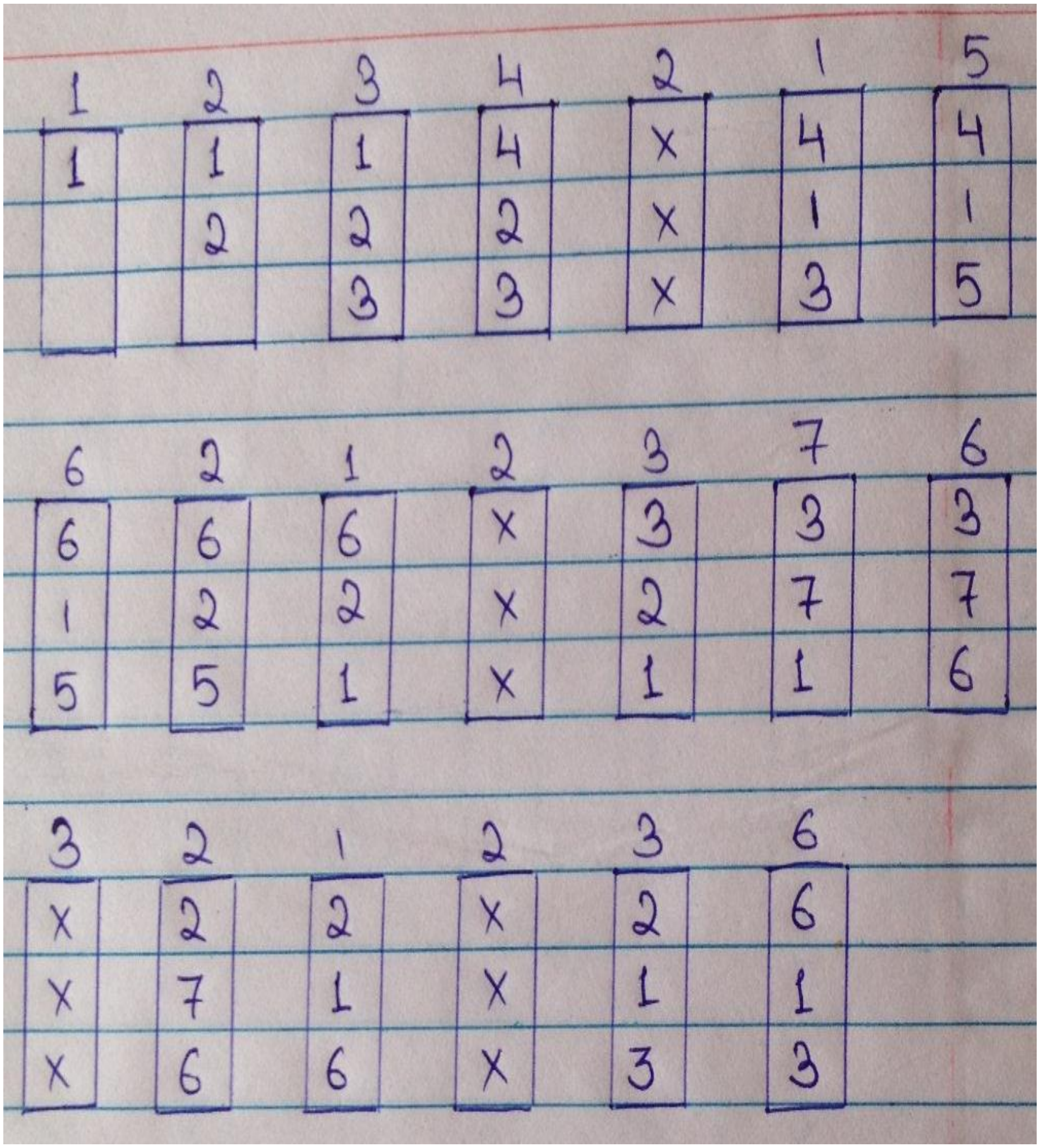
6	2	1	2	3	7	6
1	X	X	X	1	1	X
2	X	X	X	2	2	X
6	X	X	X	6	6	X
4	X	X	X	3	3	X
5	X	X	X	5	7	X

3	2	1	2	3	6
X	X	X	X	X	X
X	X	X	X	X	X
X	X	X	X	X	X
X	X	X	X	X	X
X	X	X	X	X	X

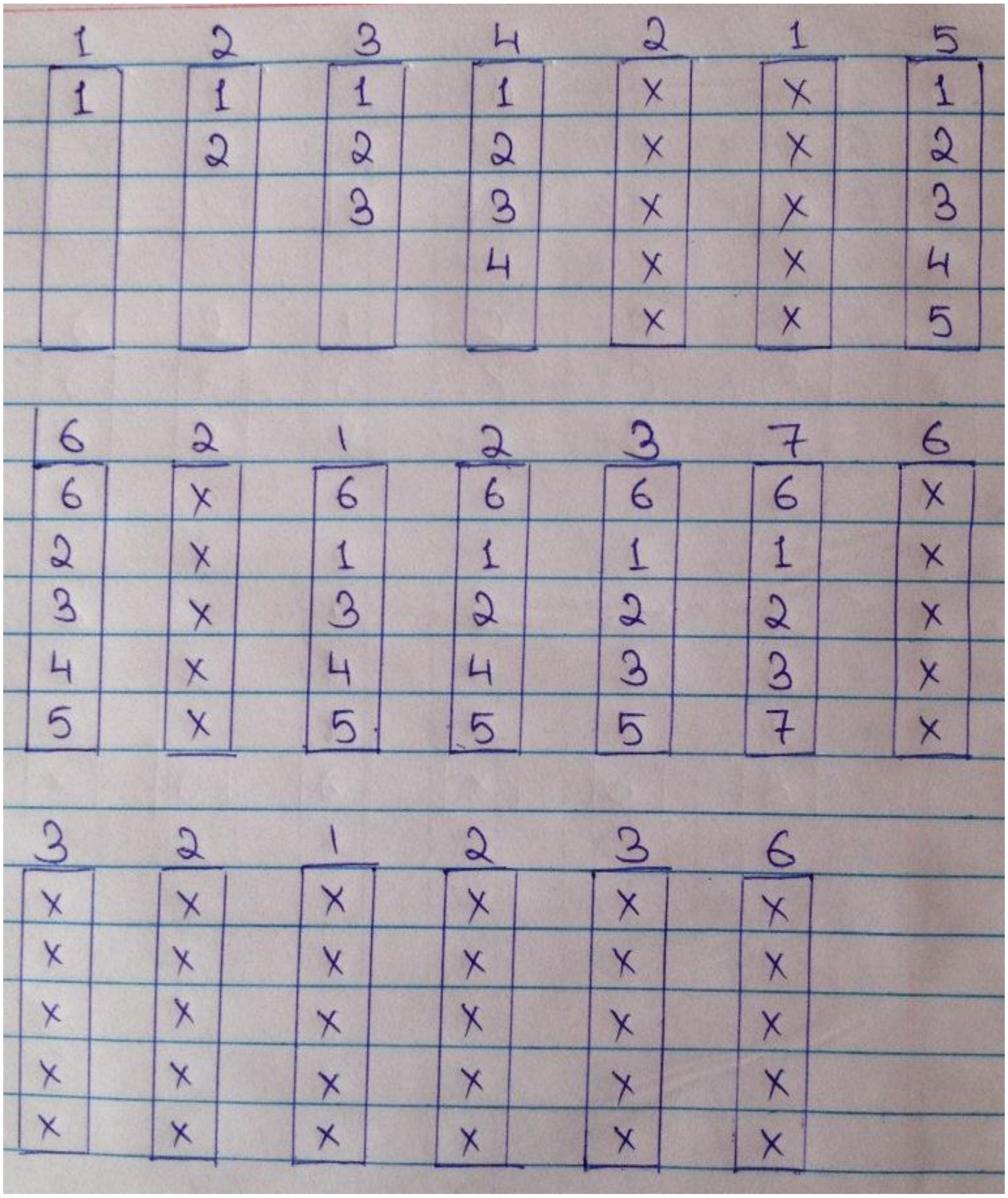
<u>LRU replacement for seven frames:</u>



FIFO replacement for three frames:


1 2 3 4 2 1 5
1 | 1 2 | 1 2 3 | 4 2 3 | X X X | 4 1 3 | 4 1 5
6 2 1 2 3 7 6
6 1 5 | 6 2 5 | 6 2 1 | X X X | 3 2 1 | 3 7 1 | 3 7 6
3 2 1 2 3 6
X X X | 2 7 6 | 2 1 6 | X X X | 2 1 3 | 6 1 3

FIFO replacement for five frames:

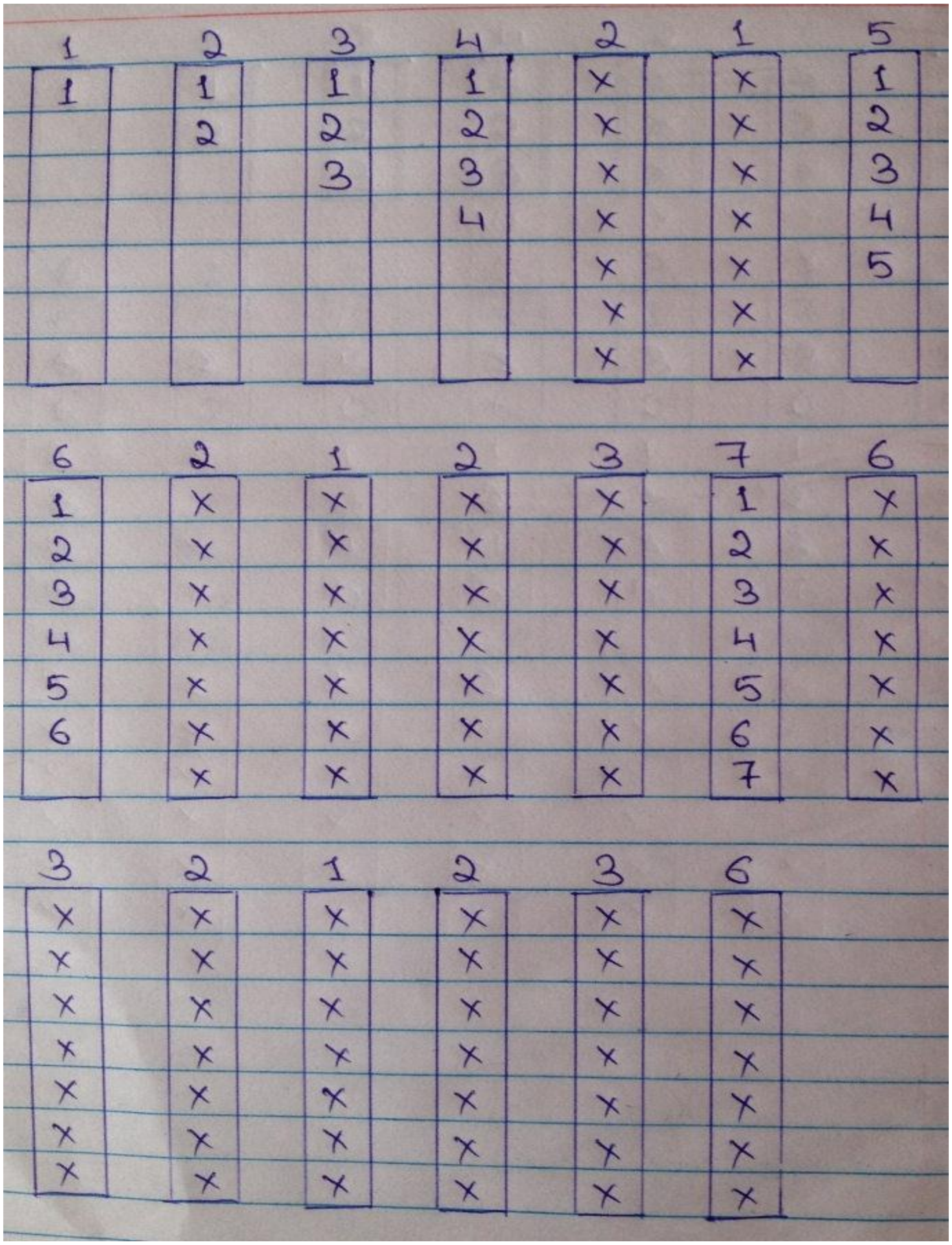


1	2	3	4	2	1	5
1	1	1	1	X	X	1
	2	2	2	X	X	2
		3	3	X	X	3
			4	X	X	4
				X	X	5

6	2	1	2	3	7	6
6	X	6	6	6	6	X
2	X	1	1	1	1	X
3	X	3	2	2	2	X
4	X	4	4	3	3	X
5	X	5	5	5	7	X

3	2	1	2	3	6
X	X	X	X	X	X
X	X	X	X	X	X
X	X	X	X	X	X
X	X	X	X	X	X
X	X	X	X	X	X

FIFO replacement for seven frames:

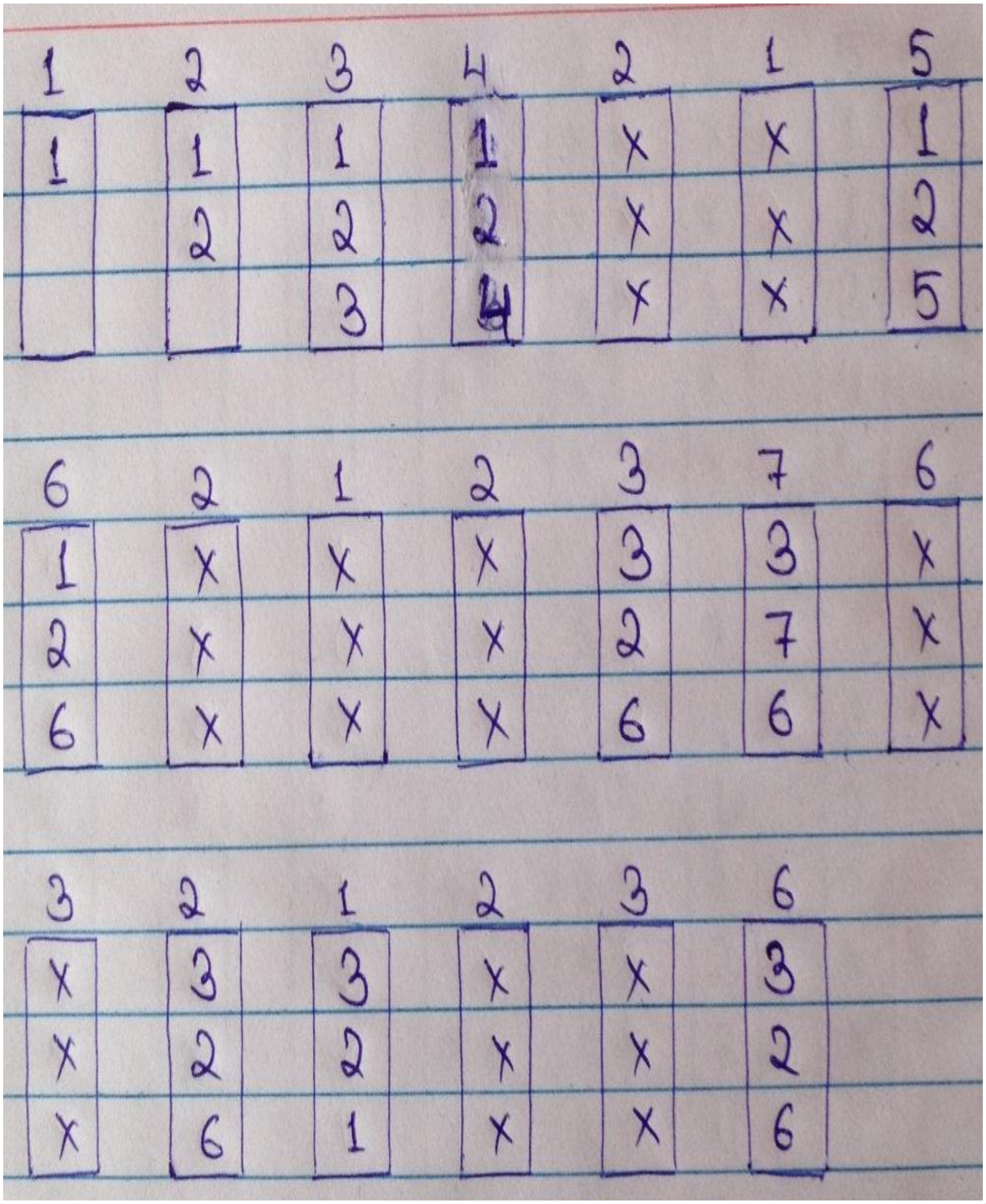


1	2	3	4	2	1	5
1	1	1	1	X	X	1
	2	2	2	X	X	2
		3	3	X	X	3
			4	X	X	4
				X	X	5
				X	X	
				X	X	

6	2	1	2	3	7	6
1	X	X	X	X	1	X
2	X	X	X	X	2	X
3	X	X	X	X	3	X
4	X	X	X	X	4	X
5	X	X	X	X	5	X
6	X	X	X	X	6	X
	X	X	X	X	7	X

3	2	1	2	3	6
X	X	X	X	X	X
X	X	X	X	X	X
X	X	X	X	X	X
X	X	X	X	X	X
X	X	X	X	X	X
X	X	X	X	X	X
X	X	X	X	X	X

Optimal replacement for three frames:

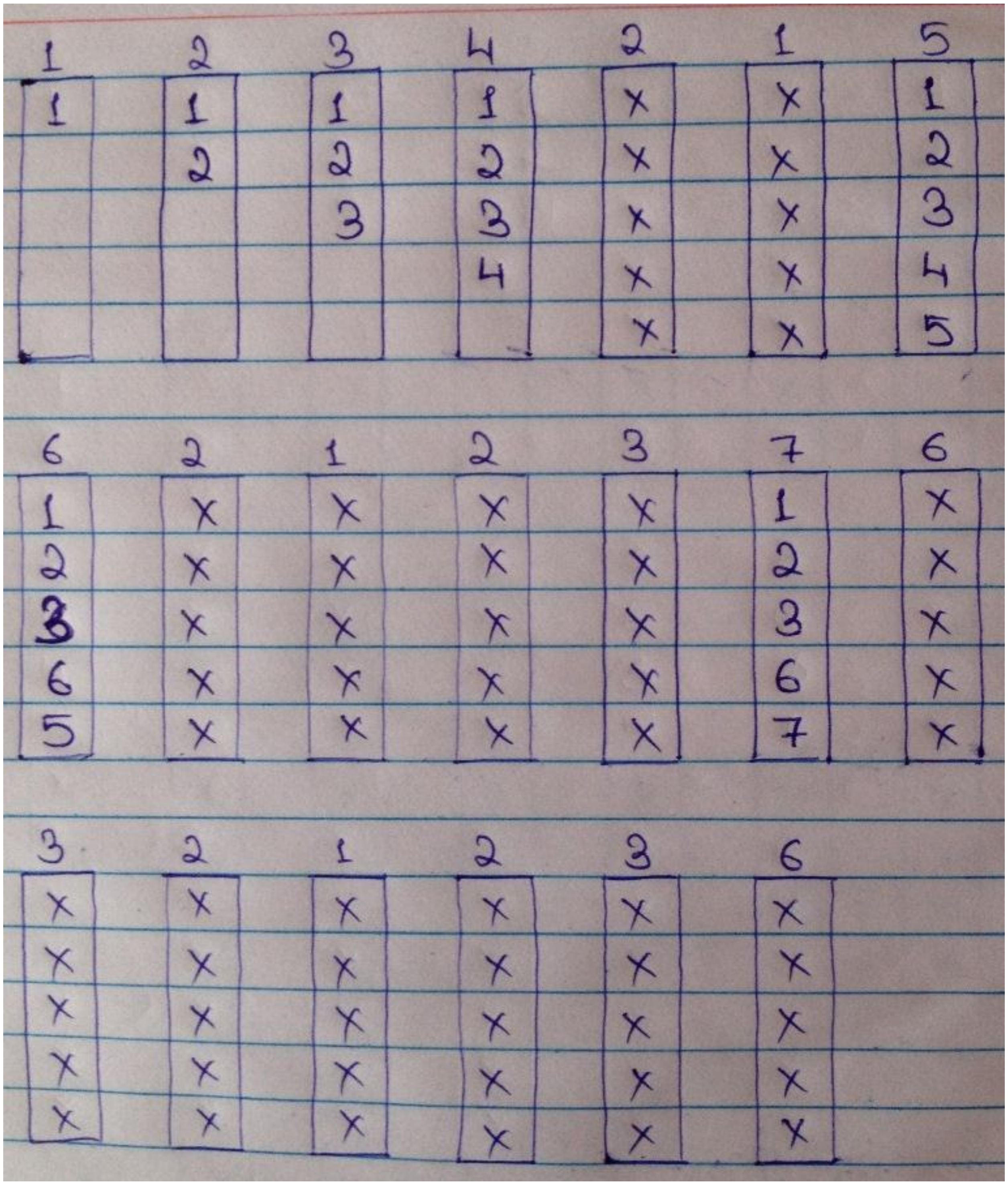


1	2	3	4	2	1	5
1	1	1	1	X	X	1
	2	2	2	X	X	2
		3	4	X	X	5

6	2	1	2	3	7	6
1	X	X	X	3	3	X
2	X	X	X	2	7	X
6	X	X	X	6	6	X

3	2	1	2	3	6
X	3	3	X	X	3
X	2	2	X	X	2
X	6	1	X	X	6

Optimal replacement for five frames:

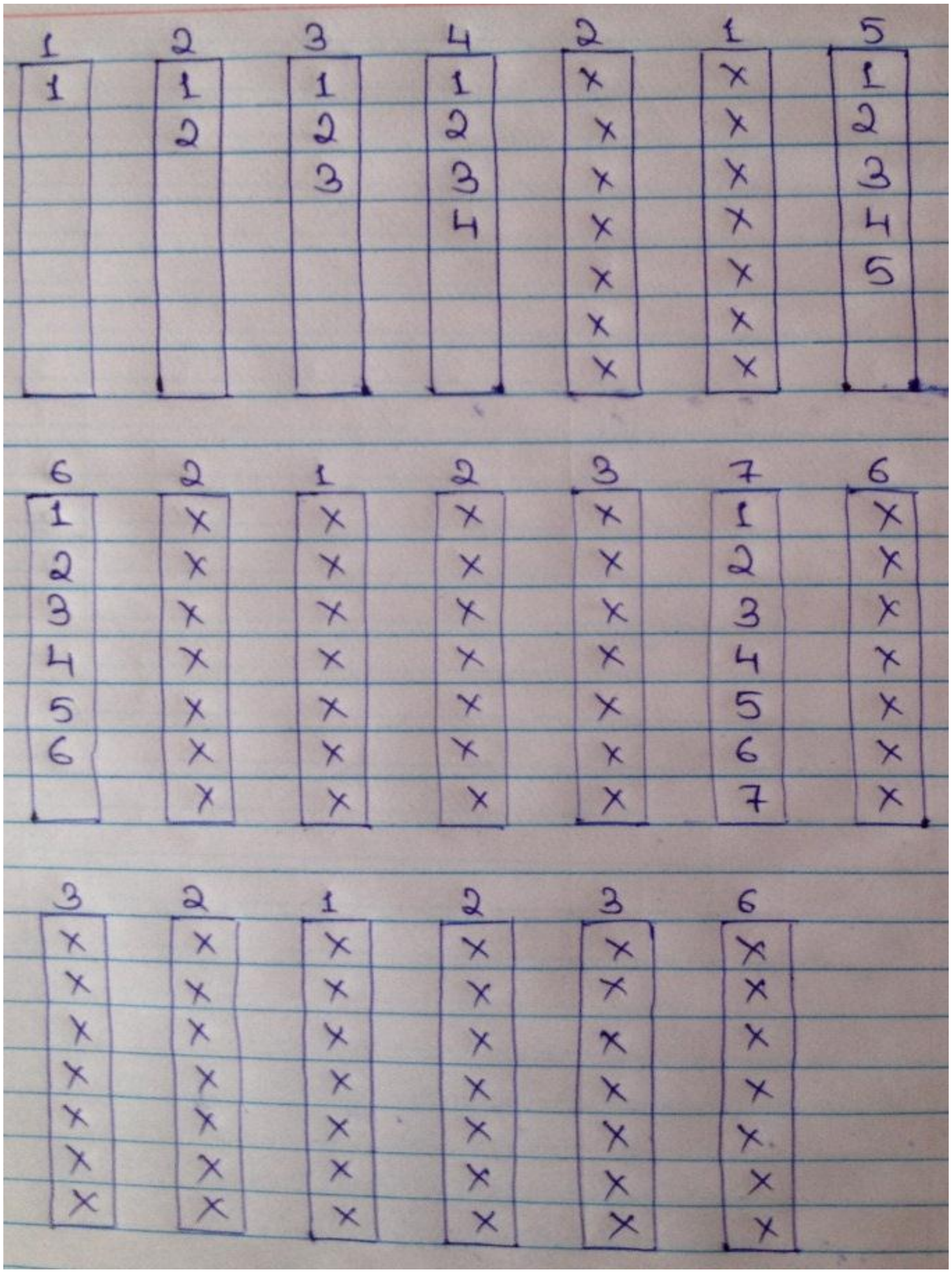


1	2	3	4	2	1	5
1	1	1	1	X	X	1
	2	2	2	X	X	2
		3	3	X	X	3
			4	X	X	4
				X	X	5

6	2	1	2	3	7	6
1	X	X	X	X	1	X
2	X	X	X	X	2	X
3	X	X	X	X	3	X
6	X	X	X	X	6	X
5	X	X	X	X	7	X

3	2	1	2	3	6
X	X	X	X	X	X
X	X	X	X	X	X
X	X	X	X	X	X
X	X	X	X	X	X
X	X	X	X	X	X

Optimal replacement for seven frames:



Therefore, the number of page faults that would occur for the LRU replacement, FIFO replacement, & optimal replacement algorithms assuming three, five, or seven frames are given in the table below:

Number of frames	Page faults for LRU	Page faults for FIFO	Page faults for Optimal
3	15	16	11
5	8	10	7
7	7	7	7

Section B $(8 \times 5 = 40)$

4. Differentiate between personal computer operating systems and mainframe operating systems.

The major difference between personal computer operating systems and mainframe operating systems is that a mainframe operating system will probably need to service many users at the same time, whereas a PC operating system normally only has to worry about one user at a time. There are some more differences between personal computer OS and mainframe OS, one of them is a mainframe OS can be used by many users at the same time so it must need to service for many users. However a personal computer operating system designed for one user normally. Second of them a mainframe OS is more powerful and expensive than PC OS. Mainframe OS designed to huge process from many users and it means, it manages lots of I/O for many users, but in PC OS there is just one user to log in it means, it does not manages lots of I/O for one user. Mainframe OS offers three kinds of service batch, transaction processing and timesharing. Personal computer OS used for some basic processing like word processing, spreadsheets and Internet access that PC OS is supports multiprogramming.

5. When do page fault occur? Describe the actions taken by an OS when a page fault occurs?

A page fault occurs when an access to a page that has not been brought into main memory takes place. The operating system verifies the memory access, aborting the program if it is invalid. If it is valid, a free frame is located and I/O is requested to read the needed page into the free frame. Upon completion of I/O, the process table and page table are updated and the instruction is restarted.

6. List four necessary conditions for deadlock. Explain each of them briefly what would be necessary (in the operating system) to prevent the deadlock.

Deadlock can be defined formally as: A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause. Because all the processes are waiting, none of them will ever cause any of the events that could wake up any of the other members of the set, and all the processes continue to wait forever. Here below are the four conditions which must be present for a deadlock to occur. If one of them is absent, no deadlock is possible.

- Mutual exclusion condition: Each resource is either currently assigned to exactly one process or is available.
- Hold and wait condition: Processes currently holding resources granted earlier can request new resources.

- No preemption condition: Resources previously granted cannot be forcibly taken away from a process. They must be explicitly released by the process holding them.
- Circular wait condition: There must be a circular chain of two or more processes, each of which is waiting for a resource held by the next member of the chain.

In general, four strategies are used for dealing with deadlocks. They are:

- Just ignore the problem altogether. Maybe if you ignore it, it will ignore you.
- Detection and recovery: Let deadlocks occur, detect them, and take action.
- Dynamic avoidance by careful resource allocation.
- Prevention: by structurally negating one of the four conditions necessary to cause a deadlock.

7. Draw and describe the 3-state process model.

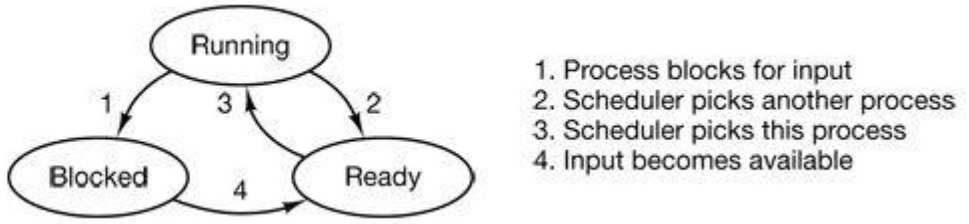

Running
1
3
2
Blocked
4
Ready
1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

Fig: 3-state process model

Figure above shows the 3-state process model. Here, we see a state diagram showing the three states a process may be in:

- Running (actually using the CPU at that instant)
- Ready (runnable; temporarily stopped to let another process run)
- Blocked (unable to run until some external event happens)

Logically, the first two states are similar. In both cases the process is willing to run, only in the second one, there is temporarily no CPU available for it. The third state is different from the first two in that the process cannot run, even if the CPU has nothing else to do.

Four transitions are possible among these three states, as shown. Transition 1 occurs when a process discovers that it cannot continue. In some systems the process must execute a system call, such as block or pause, to get into blocked state. In other systems, including UNIX, when a process reads from a pipe or special file (e.g., a terminal) and there is no input available, the process is automatically blocked. Transitions 2 and 3 are caused by the process scheduler, a part of the operating system, without the process even knowing about them. Transition 2 occurs when the scheduler decides that the running process has run long enough, and it is time to let another process have some CPU time. Transition 3 occurs when all the other processes have had their fair share and it is time for the first process to get the CPU to run again. Transition 4 occurs when the external event for which a process was waiting (such as the arrival of some input) happens. If no other process is running at that instant, transition 3 will be triggered and the process will start running. Otherwise it may have to wait in ready state for a little while until the CPU is available and its turn comes.

8. Does window have any concept of process hierarchy? How does parent control the child?

In some systems, when a process creates another process, the parent process and child process continue to be associated in certain ways. The child process can itself create more processes, forming a process hierarchy. Note that unlike plants and animals that use sexual reproduction, a process has only one parent (but zero, one, two, or more children). Windows does not have any concept of a process hierarchy. All processes are equal. The only place where there is something like a process hierarchy is that when a process is created, the parent is given a special token (called a handle) that it can use to control the child. However, it is free to pass this token to some other process, thus invalidating the hierarchy.

In the operating system UNIX, every process except process 0 (the swapper) is created when another process executes the fork () system call. The process that invoked fork is the parent process and the newly-created process is the child process. Every process (except process 0) has one parent process, but can have many child processes. The operating system kernel identifies each process by its process identifier. Process 0 is a special process that is created when the system boots; after forking a child process (process 1), process 0 becomes the swapper process (sometimes also known as the "idle task"). Process 1, known as init, is the ancestor of every other process in the system.

9. What is the problem with thread implementation in user space when any of the threads get blocked while performing I/O operation?

Somewhat analogous to the problem of blocking system calls is the problem of page faults. Computers can be set up in such a way that not all of the program is in main memory at once. If the program calls or jumps to an instruction that is not in memory, a page fault occurs and the operating system will go and get the missing instruction (and its neighbors) from disk. This is called a page fault. The process is blocked while the necessary instruction is being located and read in. If a thread causes a page fault, the kernel, not even knowing about the existence of threads, naturally blocks the entire process until the disk I/O is complete, even though other threads might be runnable.

10. Explain why two level scheduling is commonly used.

The objective of multiprogramming is to have some process running all time, to maximize CPU utilization. For this, the process scheduler selects an available process for execution on CPU from set of possible available process. Two level scheduling includes high level scheduling (job scheduling) and low level scheduling (dispatching).

High level scheduling:

- Selects jobs allowed to compete for CPU and other system resources.

Low level scheduling:

- Selects the ready process that will be assigned the CPU.
- Ready Queue contains PCBs of processes.

Thus, two level scheduling is commonly used.

11. What are the main motivations and issues in primary memory management?

The operating system must co-reside in main memory with one or more executing processes. Memory management is responsible for allocating memory to processes and for protecting the memory allocated to each process from undesired access b tither processes. It is also responsible for protecting the memory allocated to the operating system from unauthorized access. Memory management is not just a software task. The operating system requires hardware support to implement any of the nontrivial memory management schemes. Thus, some operating system design issues are also hardware design issues. Memory management hardware is typically protected from user access; the operating system is solely responsible for its control.

The main issues in primary memory management are:

- Single Partition
  - Absolute Single Partition
  - Relocatable Single Partition
- Multiple Partition
  - Multiple Fixed Partition
  - Multiple Variable Partition
- Partition Selection Algorithm

12. Explain the disk management with example.

The operating system is responsible for several other aspects of disk management too. Here we discuss about disk formatting, error handling with bad blocks.

Disk Formatting

Before a disk can store a data, it must be divided into sectors that the disk controller can read and write, called low level formatting. The sector typically consists of preamble, data and ECC (error correcting code). The preamble contains the cylinder and the sector numbers and the ECC contains redundant information that can be used to recover from read error. The size depends upon the manufacturer, depending on reliability.

Preamble	Data	ECC

If disk I/O operations are limited to transferring the single sector at a time, it reads the first sector from the disk and doing the ECC calculation, and transfers to main memory, during this time the next sector will fly by the head. When transferring completes the controller will have to wait almost an entire rotation for the second sector to come around again. This problem can be eliminated by numbering the sectors in an interleaved fashion when formatting the disks. According to copying rate, interleaving may be of single or double.

Error handling with bad blocks

Most frequently, one or more sectors become defective or most disks even come from factory with bad blocks. Depending on the disks and controller in use, these blocks handled on variety of ways.

- Bad blocks are handled manually. For example run MSDOS chkdsk command to find bad blocks, and format command to create new block- data resided on bad blocks usually are lost.
- Using bad block recovery: The controller maintains the list of bad blocks on the disk and for each bad block, one of the spares is substituted.