Name: Kaustik Jolapalla
SAP ID: 60004200107
Div: B

DATE
PAGE NO.

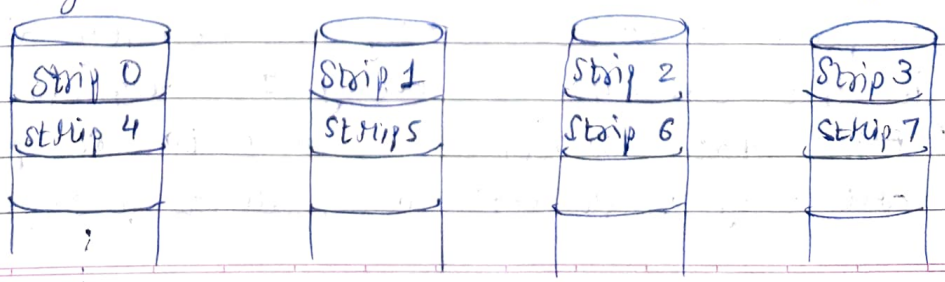
OS
Assignment 1

Q. 1) Describe RAID disk management. Provide detailed desc of the all the levels of RAID.

→
- RAID is a set of physical disk drivers viewed by the operating system as a signal logical drive
- RAID is the acronym for Redundant Array of Independent Disks or Redudant Array of Inexpensive Disks.
- Here the data are distributed across the physical drivers of an array in a scheme known as stripping
- Redudent disk capacity is used to store parity information which guarantees data recoverability in the case of an disk failure
- RAID consists of seven levels.

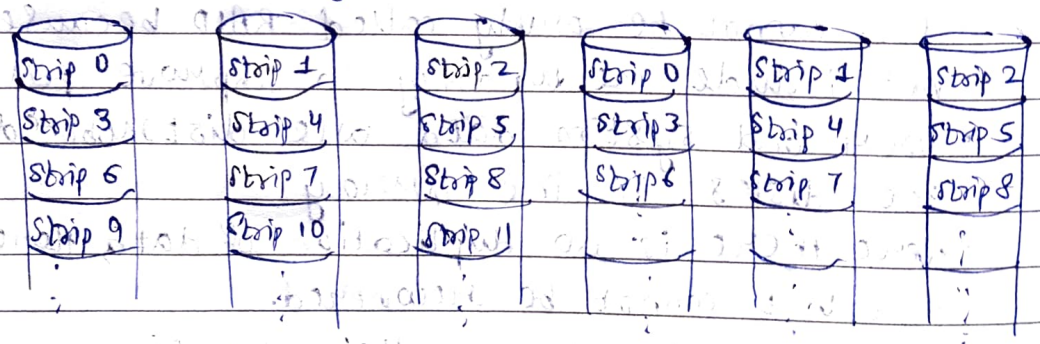
RAID level 0

- This cannot be truly called RAID because it does not include redundancy to improve performance
- User and system data are distributed across all the disks in the array
- Here there is no duplication of data here a block once lost cannot be recovered.
- This is used for parallel processing of data which increases the speed of operation.
- The logical disk is divided into strips.



⊛RAID level 1

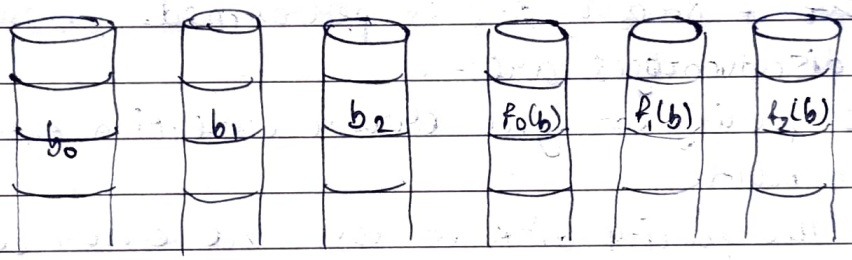
- Also known as mirrored disks as redundancy is achieved by the simple expedient of duplicating all the data
- Here if one drive fails the data may still be accessed from the second drive.
- Here there is no 'write penalty' ie. a write request requires that both corresponding strips be updated but this can be done in parallel
- The principal disadvantage is cost as it requires twice the disk space of the logical device it supports
- RAID 1 configuration is limited to device that stores system software and data and other highly critical files. In these cases, RAID 1 provides real time backup of all data. So that is the event of a disk failure, all of the critical data is still immediately available



⊛RAID Level 2

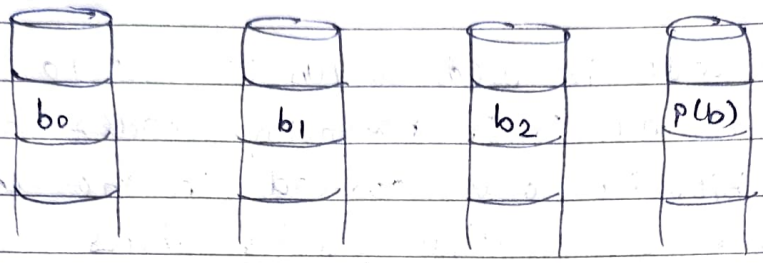
- It makes use of parallel ~~to~~ access technique. In a parallel access array, all number disks participate in the execution of every I/O request.
- Here, data stripping is used to strip. The strips are very small often of small as a single byte/word.

- A hamming code is used, which is able to correct single bit error. These hamming codes are calculated bitwise and stored in the corresponding bit positions on multiple parity disks
- RAID level 2 requires less disks than RAID level 1.
- ~~RAID level~~ If there is a single bit error, the controller can recognize and correct the error instantly so that read access time is not slow. But multiple errors cannot be checked and corrected.
- RAID level 2 will only be effective choice in an environment in which many disk error occour.



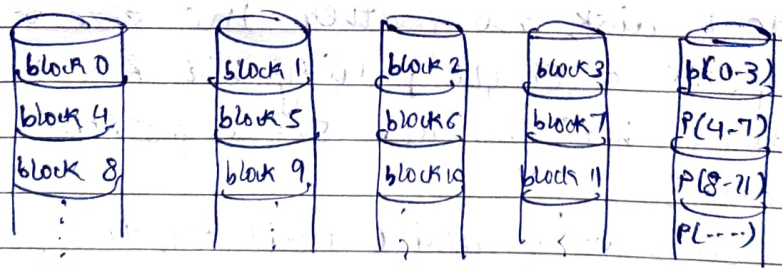
⊛ RAID level 3

- It is similar to RAID level 2
- Here, the difference is that it requires only a single redundant disk, no matter the size of disk array
- It computes a single parity bit for the set of individual bits in the same position on all the data disks.
- In the event of a drive failure, the parity drive is accessed and data is reconstructed from the remaining devices. Once the failed drive is replaced the missing data can be restored and operations are resumed
- It can achieve very high data transfer rates.



⊛ RAID level 4

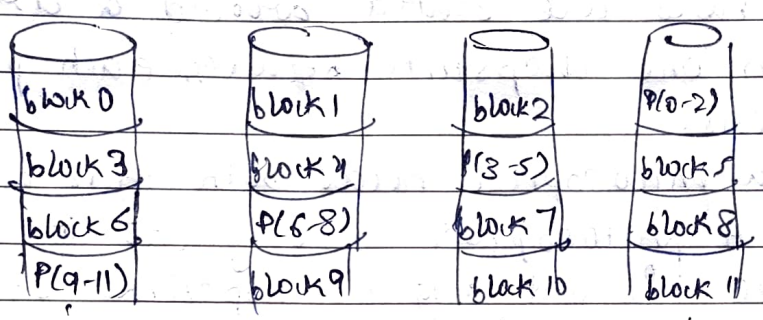
- Uses independent access technique. In independent access array each number disk operates seperately.
- Here the data is striped in blocks
- A parity strip is calculated and this parity is stored in the corresponding strip on the parity disk.
- It involves a write penalty when an I/O write request of small size is performed.
- The disadvantages are:-
  1. The parity can give error correction for only one block
  2. If the parity disk is lost the error correction cannot be done
  3. Since parity is given to one disk only, this disk is repeatedly used. This results in an overload and it may slow down.



⊛ RAID Level 5

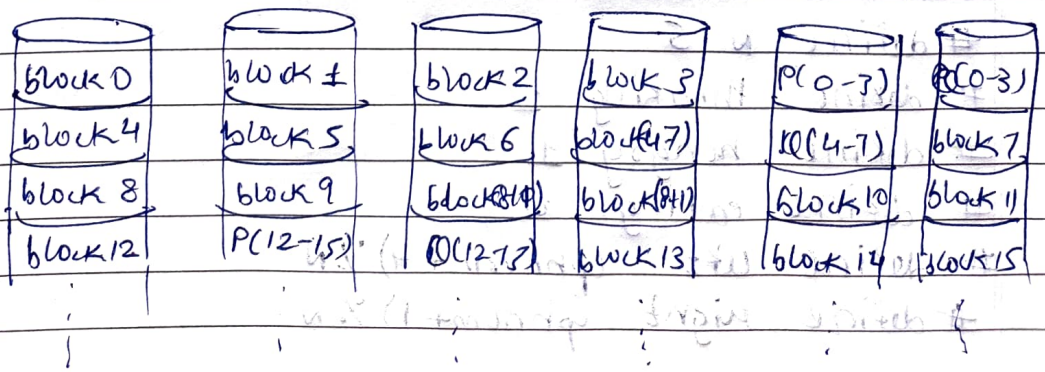
- Similar to RAID level 4
- Here the parity strips are distributed all across the disks.

- The allocation is done by round-robin scheme
- This helps in avoiding the potential I/O bottleneck of single parity disk as seen in RAID level 4.
- It has the characteristics that the loss of any one disk does not result in data loss.



RAID level 6

- Here two different parity calculations are carried out and stored in seperate blocks on different disks.
- Thus, if user data requires N disks RAID level 6 will contain N+2 disks
- The main advantage is that it provides extremely high data availability
- It includes a write penalty because each write affects two parity blocks.
- Presence of two parity strips helps in calculating for two errors.



2) Explain dining philosopher's problem and its solution using semaphores.

→ DINING PHILOSOPHER'S PROBLEM

- The dining philosophers problem states that K philosophers are seated around a circular table with one chopstick between each pair of philosophers.
- To eat a philosopher needs both their and left chopstick. A philosopher can only eat if both immediate left and right chopsticks of the philosopher is available
- In case if both immediate left & right chopstick of the philosopher are not available then the philosopher puts down their (either left or right) chopstick and starts thinking again.
- The dining philosophers problem is a classical problem of synchronization and demonstrates a large class of concurrency control problems.

Solution using semaphores

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#define N 5
#define thinking 2
#define hungry 1
#define eating 0
#define left (phnum + 4) % N
#define right (phnum + 1) % N
```

```
int state[N];
int phil[N] = {0, 1, 2, 3, 4};

sem_t mutex;
sem_t S[N];

void test(int phnum)
{
    if(state[phnum] == hungry && state[left] != eating
        && state[right] != eating)
    {
        state[phnum] = eating;
        sleep(2);
        printf("Philosopher %d takes fork %d and
                %d\n", phnum+1, left+1, phnum+1);
        printf("Philosopher %d is Eating\n", phnum+1);
        sem_post(S[phnum]);
    }
}
void take_forks(int phnum)
{
    sem_wait(&mutex);
    state[phnum] = hungry;
    printf("Philosopher %d is hungry\n", phnum+1);
    test(phnum);
    sem_post(&mutex);
    sem_wait(&S[phnum]);
    sleep(1);
}
```

```
void put_fork(int phnum)
{
    sem_wait(&mutex);
    state[phnum] = thinking;
    printf("philosopher %d putting fork %d and %d
            down\n", phnum+1, left+1, phnum+1);
    printf("philosopher %d is thinking\n", phnum+1);
     test(left);
     test(right);
      sem_post(&mutex);
}

void *philosopher(void *num)
{
    while(1)
    {
        int *i = num;
        sleep(1);
        take_fork(*i);
        sleep(0);
        put_fork(*i);
    }
}

int main()
{
   int i;
   pthread_t thread_id[N];
   sem_init(&mutex, 0, 1);
```

```
for(i=0; i<N; i++)
{
    sem_init(&S[i], 0, 0);
}
for (i=0; i<N; i++)
{
    pthread_creat (&thread_id[i], NULL,
                   philosophers, &phil[i]);
    printf("Philosopher %d is thinking\n", i+1);
}
for (i=0; i<N; i++)
    pthread_join(thread_id[i], NULL);
}
```