# ANSWER SHEET OF PRACTICE QUESTIONS

## FILE I/O

Q1. Assume that barfoo.txt contains “ABCDEF” 6 characters. What will the output from the code?

```
int main() {

int fd1, fd2;

char c;

fd1 = open("barfoo.txt", O_RDONLY, 0);

fd2 = open("barfoo.txt", O_RDONLY, 0);

read(fd1, &c, 1);

dup2(fd2, fd1);

read(fd1, &c, 1);

printf("c = %c", c);

exit(0);

}
```

ANS :

The output of the code will be "B".

Here is the explanation of how the code works:

Two file descriptors, fd1 and fd2, are opened to the same file "barfoo.txt" in read-only mode.

The first character of the file is read into the variable c using fd1, which means c will contain "A".

Then the dup2() system call is used to make fd1 a copy of fd2. This means that both fd1 and fd2 now refer to the same file descriptor, which is pointing to the beginning of the file.

The next character of the file is read into the variable c using fd1. Since fd1 and fd2 are now the same file descriptor, this read will start from the beginning of the file again, so c will contain "B".

Finally, "B" is printed to the screen using printf().

# ANSWER SHEET OF PRACTICE QUESTIONS

Q2. If a user runs a following command on a file that has 444 permissions, what will happen?

```
rm file1.txt
```

If a user tries to run the "rm file1.txt" command on a file that has 444 permissions, the command will fail, and the user will receive a "Permission denied" error message.

This is because the 444 permission means that the file is readable but not writable or executable by anyone, including the owner and group. Therefore, no one, including the owner or group, can delete or modify the file.

If the user wants to remove the file, they would need to have write permission on the directory containing the file. They can use the command "chmod u+w <directory>" to add write permission to the directory and then delete the file using "rm file1.txt".

Q3. Considering File Control Block (FCB) supports 10 direct pointers, 3 single indirect pointers, 3 double indirect pointers and 1 triple indirect pointer. Calculate the maximum size of the file supported by this FCB when each of the data block address is 32 bits. Assume each data block to be of 2KB (2048 bytes).

The maximum size of the file supported by this FCB can be calculated as follows:

10 direct pointers can address 10 blocks directly, so they can address 10 x 2048 = 20480 bytes of data.

3 single indirect pointers can address 2KB/4B = 512 data blocks. Therefore, they can address 512 x 2048 = 1048576 bytes of data.

3 double indirect pointers can address 2KB/4B x 2KB/4B = 512 x 512 = 262144 data blocks. Therefore, they can address 262144 x 2048 = 536870912 bytes of data.

1 triple indirect pointer can address 2KB/4B x 2KB/4B x 2KB/4B = 512 x 512 x 512 = 134217728 data blocks. Therefore, it can address 134217728 x 2048 = 274877906944 bytes of data.

Therefore, the maximum size of the file supported by this FCB would be the sum of all the above sizes, which is:

20480 + 1048576 + 536870912 + 274877906944 = 274914624912 bytes

So, the maximum size of the file supported by this FCB when each data block address is 32 bits and each data block is of size 2KB would be approximately 274.9 GB.

# ANSWER SHEET OF PRACTICE QUESTIONS

Q4. When ls -i is executed, following output is generated, what can you say about the output? 1000 file1.txt 1000 file2.txt 1001 file3.txt → file1.txt a. file1.txt and file2.txt have same inode numbers so they are soft links b. file1.txt and file2.txt have same inode numbers so they are hard links c. If we delete file1.txt, we can still get the content of the same file by executing cat file3.txt d. If we delete file1.txt, we can still get the content of the same file by executing cat file2.txt e. None of the above

From the given output, we can see that file1.txt and file2.txt have the same inode number (1000). This means that they are hard links to the same file.

However, there is no information to suggest that file3.txt is a link to either file1.txt or file2.txt. It is simply a regular file with a different inode number (1001).

Therefore, the correct answer is (b) file1.txt and file2.txt have same inode numbers so they are hard links. Option (c) and (d) are incorrect because they involve file3.txt being a link to file1.txt, which is not indicated by the given output. Option (a) is incorrect because there is no indication that either file1.txt or file2.txt are soft links.

Q5. What will be content of the Process File Descriptor (PFD) table and Systems Wide File Open (SWFO) table upon executing following code? int main() { int fd1, fd2; char c; fd1 = open("barfoo.txt", O_RDONLY, 0); fd2 = open("barfoo.txt", O_RDONLY, 0); read(fd1, &c, 1); dup2(fd2, fd1); read(fd1, &c, 1); printf("c = %c", c); exit(0); } Assume that the initial entries of PFD and SWFO are shown below.

| Index | SysFD Ptr |
|---|---|
| 1 | fd1=10 |
| 2 | |
| 3 | |
| 4 | |

| SysFD | Offset | RefCnt | inode Ptr |
|---|---|---|---|
| 10 | 0 | 1 | 100 |

# ANSWER SHEET OF PRACTICE QUESTIONS

Ans :

The Process File Descriptor (PFD) table and Systems Wide File Open (SWFO) table will be updated as follows upon executing the given code:

Initially, the PFD table has an entry for fd1 and is associated with SysFD 10, and fd2 is not yet used. The SWFO table has an entry for SysFD 10, which is associated with inode 100.

PFD table:

| Index | SysFD Ptr |
|---|---|
| 1 | fd1=10 |
| 2 | |
| 3 | |
| 4 | |
| 5 | fd2=11 |

SWFO table:

| SysFD | Offset | RefCnt | inode Ptr |
|---|---|---|---|
| 10 | 0 | 2 | 100 |
| 11 | 0 | 1 | 100 |

After the first call to read(), the file offset of fd1 will be at position 1. The file offset of fd2 is still at position 0 because it has not been used yet.

PFD table:

| Index | SysFD Ptr |
|---|---|
| 1 | fd1=10 |
| 2 | |

# ANSWER SHEET OF PRACTICE QUESTIONS

3

4

5 fd2=11

SWFO table:

SysFD Offset RefCnt inode Ptr

10 1 2 100

11 0 1 100

After the call to dup2(), fd1 and fd2 will refer to the same file descriptor, which is associated with SysFD 10. The file offset of fd1 and fd2 will be the same and at position 1.

PFD table:

Index SysFD Ptr

1 fd1=10

2 fd2=11

3

4

5

SWFO table:

SysFD Offset RefCnt inode Ptr

10 1 2 100

11 1 2 100

Finally, after the second call to read(), the file offset of the shared file descriptor will be at position 2. The variable c will contain the character at position 1 of the file, which is 'B'.

# ANSWER SHEET OF PRACTICE QUESTIONS

PFD table:

| Index | SysFD Ptr |
|---|---|
| 1 | fd1=10 |
| 2 | fd2=11 |
| 3 | |
| 4 | |
| 5 | |

SWFO table:

| SysFD | Offset | RefCnt | inode Ptr |
|---|---|---|---|
| 10 | 2 | 2 | 100 |
| 11 | 2 | 2 | 100 |

Therefore, the content of the PFD and SWFO tables after executing the code will be as shown above.

# ANSWER SHEET OF PRACTICE QUESTIONS

# PROCESS MANAGEMENT

Q1. How many times 'ls' command will get executed in the below code?

```
int main()

{

fork(); // line 1

fork(); // line 2

execlp("ls", "ls", NULL); // line 3

fork(); // line 4

execlp("ls", "ls", NULL); // line 5

}
```

The ls command will be executed a total of 4 times in the given code.

Here's how the code executes:

At line 1, the first fork() call creates a new child process. There are now two processes, the parent process and the child process.

At line 2, both the parent process and the child process created in the previous step call fork(), which creates two more child processes. Now, there are four processes: the original parent process and three child processes.

At line 3, the execlp("ls", "ls", NULL) call replaces the current process image with the ls command, which is executed once by each of the four processes.

At line 4, both child processes created in step 2 again call fork(), creating two more child processes. Now, there are six processes: the original parent process and five child processes.

At line 5, the execlp("ls", "ls", NULL) call replaces the current process image with the ls command, which is executed once by each of the five child processes.

Therefore, the ls command will be executed a total of four times by the child processes created in steps 1, 2, and 5.

# ANSWER SHEET OF PRACTICE QUESTIONS

Q2. Draw a binary to show the number of processes created by following code. What is the count of

number of processes created?

```
int main(void)

{

fork() && fork();

}
```

Here's the binary tree diagram showing the process creation:

Copy code

```
      P

    /  \

   P     P

  / \   / \

 C   C C   C
```

The code calls fork() once, and then uses the logical AND (&&) operator to execute another fork() call only in the child process that was created by the first fork() call.

In the above diagram, P represents a process, and C represents its child process. The initial process (parent) calls fork(), creating a child process. Then, the child process executes the second fork() call, creating two more child processes. Therefore, there are a total of 4 processes created.

So the count of the number of processes created is 4.

# ANSWER SHEET OF PRACTICE QUESTIONS

Q3. What value of i variable be printed by parent and the child process from the below code?

```
int i=0;

int main(void)

{ int pid, childpid, status;

i++;

pid = fork();

i++;

if (pid != 0) // parent process

{

i++;

childpid = wait(&status);

i = i + WEXITSTATUS(status);

printf("parent: i = %d\n", i);

}

else // child process

{

i += 2;

printf("child: i = %d\n", i);

exit(10);

}

}
```

The value of i printed by the parent and child processes would be as follows:

Child process: The child process starts with i equal to 0, and then increments it twice to become 2. It then prints the value of i, which is 2, and exits with a status of 10.

Parent process: The parent process also starts with i equal to 0, and then increments it once to become 1. It then forks a child process, which starts executing from where the parent left off. The parent then increments i again to become 2, and prints the value of i. At this point, the child process has already printed its value of i and exited with a status of 10. The parent then calls wait() to wait for the child process to exit, and adds the exit status (10) to i, making it 12. Finally, the parent prints its updated value of i, which is 12.

Therefore, the value of i printed by the child process is 2, and the value of i printed by the parent process is 12.

Q4. List out the process states in the same order the child process goes through when the following code gets executed.

```
int main()
{
if (fork() == 0)
{
int fd; char buf[100000]
fd = open("large.txt",O_RDONLY);
read(fd,buf,len(buf));
exit(10)
}
else
{
while(1)
sleep(5*60);
}
}
```

Ans :

The child process goes through the following process states in the order listed:

Created: The child process is created by calling fork() from the parent process.

# ANSWER SHEET OF PRACTICE QUESTIONS

Ready: Once the child process is created, it is added to the system's process scheduler and made ready to run.

Running: When the system's process scheduler selects the child process to run, it enters the running state and begins executing the code.

Blocked: The child process becomes blocked when it calls open() to open the file "large.txt" for reading. The child process waits until the file is opened before proceeding with the read() system call.

Running: After the file is opened, the child process continues running and calls read() to read the contents of the file into the buffer.

Terminated: After the read operation is completed, the child process calls exit() with a status of 10 to terminate itself.

The parent process goes through the following process states in the order listed:

Created: The parent process is created at the start of the program.

Ready: The parent process is made ready to run after the child process is created and added to the process scheduler.

Running: When the system's process scheduler selects the parent process to run, it enters the running state and begins executing the code.

Blocked: The parent process becomes blocked in an infinite loop of sleep() system calls, waiting for the child process to terminate.

Running: After the child process terminates, the parent process continues running and exits normally when the program ends.

Q5. Write a master-slave implementation for the following situation.

1. Main program accepts 5 filenames of text files as command line argument.

2. You need to create 5 child processes and pass each of the 5 filenames as a command line argument to each of the 5 child process. For example first child will get the filename 1, second child will get filename 2 and so on.

3. Child process will execute wc command for the filename provided and display the output from the wc command.

# ANSWER SHEET OF PRACTICE QUESTIONS

Ans :

```
#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>


#define NUM_FILES 5


int main(int argc, char *argv[]) {

    pid_t pid[NUM_FILES];

    int i;


    // Create 5 child processes

    for (i = 0; i < NUM_FILES; i++) {

        pid[i] = fork();

        if (pid[i] < 0) {

            fprintf(stderr, "Error: Fork failed.\n");

            exit(1);

        } else if (pid[i] == 0) {  // Child process

            char *filename = argv[i + 1];

            execlp("wc", "wc", filename, NULL);

        }

    }


    // Wait for all child processes to complete

    for (i = 0; i < NUM_FILES; i++) {

        waitpid(pid[i], NULL, 0);

    }
    return 0;

}
```

In this implementation, the main program creates 5 child processes using a for loop. In each iteration of the loop, the filename is passed as a command line argument to the child process using `argv[i + 1]`. The `execlp` function is used to execute the `wc` command with the given filename as argument.

After all child processes have been created, the main program waits for all child processes to complete using the `waitpid` function. Once all child processes have completed, the program exits.