1. Please write the answer for each question ONLY in the space provided below the question.

5 1. (a) What are the datatypes of a process and process id in MINIX? Also, if you want to print a process id, which type specifier you will use in C and why?

Solution:

→ process is represented as a process control Block(PCB). This has type struct . ~~Specif~~ specifically, it is struct mproc
↳ (we used rmp, rmc pointers)

→ Process ID is of pid_t datatype.

→ To print pid, we can use %u, since it is effectively an unsigned integer. (in C)

(b) In Lab2, You had implemented three programs `half`, `square` and `twice` and you were asked to calculate half(twice(half(square(twice(10))))) and print 200 as result. It should also print the process ids of each program as it executes. Will the printed PIDs will be same or different? Explain with reason.

Solution:

→ The printed PIDs will be same.

→ This is because of our use of execvp(), which replaces the code part of the process with the code given as input (in its executable form). We never use fork(). Hence, execvp() does not create any 'new' process. ⇒ PID will be same

(c) What is endpoint in schedproc(struct)? What does the spawn and syscall benchmarks in unixbench do?

Solution:

→ Endpoint is a unique identifier for a ~~[illegible]~~ process that is composed of PID, and a 'generated number', such that the ordered pair of the two is almost always unique. This facilitates PID recycling.

→ spawn benchmark forks a lot of processes, effectively 'spawning' them. This would be CPU-Intensive.

→ syscall benchmark invokes lots of system calls, ~~which may also involve some file descriptor reads/writes.~~
↳ So, it is only 'primarily' CPU-Intensive

6 2. (a) In Lab 4, when you designed the SJF/RR/SRTF schedulers, how did you calculate the penalty ratio for each process. For an interactive process which algorithm is better: SJF, RR, or SRTF? Explain

Solution: (per process)

→ $\boxed{\text{Penalty Ratio} = \frac{\text{Waiting Time}}{\text{Turnaround Time}}}$ } → Some sources consider the reciprocal of this!

(Note that the definition of Penalty Ratio has been explained differently in different sources; An IITB PPT had a definition based on Response Ratio.)
↳ our theory course also did not fully establish a formula for this!!

(In terms of Due Diligence of finding out the definition)

1

→ For an interactive process, response time is a very important metric. So, ROUND ROBIN would be the ~~better algorithm.~~ ~~(followed by SRTF, [illegible])~~ ~~since it ensures fairness~~. ↓ a sort of guarantee in the response time

(b) How many priority queues are present in Minix? What is the highest and lowest priority for the user processes in Minix? Also, Why were you not able to implement a complete FIFO scheduler in Minix?

Solution: → MINIX has 16 priority levels in its Multi-level Feedback Queue (MLFQ).

→ The highest priority for a user process is $\boxed{7}$.
The lowest priority for a user process is $\boxed{15}$.

1.5

→ The problem with implementing pure-FIFO would be dealing with I/O-bound processes. These processes would go to blocked state, allowing other processes to use the CPU. Then, when I/O is complete, it re-enters the ready queue, engaging in it's own ~~[illegible]~~ ~~[illegible]~~ pseudo round robin type scheduling

(c) How does the priority of the process change depending on the nature of the workload? How is the time quantum decided in Minix for different workloads? How does the Minix scheduler avoid starvation?

(Hence, it was pseudo FIFO)

Hence, even though this I/O process may have come before some other CPU process, the CPU-bound process would still finish first, in one long

MINIX prevents starvation by periodically ~~to be~~ boosting the priorities of all processes, "forgetting" ~~its~~ its knowledge of the workloads' nature. This is essential to prevent starvation, and is a rule of MLFQ Implementation that MINIX adheres to.

Solution: → If the workload was CPU bound, it uses a greater fraction of its allotted quantum. ~~Based on it~~

↳ In fact, almost always using the entire quantum.

Such processes' priorities are decreased. Hence, CPU-bound processes' ~~or~~ priorities get lowered overtime. While, I/O-bound processes don't use their whole quantum, most of the time. Such processes' priorities aren't changed.

→ So, I/O-bound workloads would often be alloted 500ms quanta, as opposed to the usual 200ms quanta, since they are more likely to relinquish the CPU in that time anyway.

1.5

(d) Which function you used to create a shared memory in C, explain along with its parameters ? Also, Can you comment and compare the results obtained for image processing transformations performed by two different processes and the communication between them implemented using shared memory and pipes ?

Solution: → To create a shared memory, we use $\boxed{\text{shmget()}}$.

In this function, we pass an shmid, ~~and~~ the memory location to be shared, and the size of aforementioned memory location.

→ Shared memory is the fastest for moderate-sized ~~files~~ images, compared to sequential and piped implementations. While, for larger files, pipes would eventually surpass even the performance of shared memory.

The lack of parallelism (actually, concurrency) causes sequential to lag behind in all cases.

1.5

shm>pipes>seq

pipes>shm>seq

Note: For very small files, shared memory management and pipes may both incur overhead costs, making sequential better for such a case

3 3. The following code is given to you. You need to explain what the given code does? What is achieved by using the variables HIGH and FREQ ?

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <assert.h>

#define HIGH 30
#define FREQ 70

void init(int *sequence, int refs, int pages){
  int high = (int) (pages*((float)HIGH/100));
  for(int i=0; i<refs; i++){
      if(rand()%100 < FREQ) {
        sequence[i] = rand()%high;
      }
      else{
        sequence[i] = high + rand()%(pages-high);
      }
  }
}
```

10

100

10

30

→ 70% of references

→ 30% of pages.

70

30

30

% 100-30 → 70

(in the init)
→ Note that, here we are assuming that that 30% of pages consists of addresses 0-~~89~~, while the 70% of pages are 30-69. In real implementations, the spread would not be this contiguous

```
int main(int argc, char *argv[]){
    int refs =10;
    int pages = 100;
    int *sequence = (int*) malloc(refs*sizeof(int));
    init(sequence, refs, pages);
    for(int i=1; i<refs; i++){
        printf(", %d", sequence[i]);
    }
    return 0;
}
```

→ 10 integers.

→ an array of 10 integers.

(int) (int)

} printing the generated page request sequence

Solution:

→ In the given code, we seem to be ~~constructing~~ generating a page ~~access~~ request sequence for a 30-70 workload. (may also be called 70-30) a/for { HIGH=30 FREQ=70

Here, 70% of the references occur to 30% of the pages, while 30% of the references (the rest) occur to the other 70% pages. This implicates a notion of temporal locality.

→ (FREQ) determines the percentage of references made to (HIGH)% of the pages in the system. They do not have to add up to 100, i.e., we can have 90% of accesses be made to 1% of pages. But it is common to do so.

→ Other code tidbits are written alongside the given code.

3

6 4. (a) What is the data type of inode number? When will the common_open() function be called?

Solution:

→ Inode number is of type long long unsigned → in terms of C (Question did not specify MINIX data type or C data type)

↳ (%llu) type specifier

−0.75

→ common_open() function is called when creating a file, or opening it. In the case of creation, the O_CREAT flag would be set, unlike the case where an existing file is ~~created~~ opened.

inode is not specific to 'c'

(b) What happens to the file content in the disk, when you delete a file?

Solution: → In ~~[illegible]~~ the MINIX3 implementation, we observed that, the structure containing the file inode is eventually freed, after some locks are set/unset.

However, the contents on the disk are there to stay, albeit with valid bit set to zero, so that it is treated as garbage, and can be recycled for files created in the future.

—1—

(c) What happens to the inode number a) when copying a file into the same file system b) when moving a file from one file system to another file system.

Solution: Note that inode number is unique within a given file system for a file. Using this fact, we can say that:

(a) when copying file into the same file system, the inode number DOES NOT CHANGE. (since no other file in this filesystem has THAT inode number)

(b) when moving file ~~into~~ from one file system to another file system, the inode CHANGES, since the original inode value may ~~not~~ be vacant in the new filesystem,

—0.5—

(d) What is the purpose of MARKDIRTY()?

Solution: → MARKDIRTY() is used to mark file table entries as being 'dirty' or 'modified'. ~~This would cause~~

(~~a writethrough to the~~
In the case of writethrough, this would invoke an immediate disk write.

Meanwhile, in the case of writeback, this would help in knowing which blocks are to be written to.

—0.75—

(e) What are zones in the inode? What are they used for? What is the size of the zone?

Solution: → Zones in the inode are locations where disk addresses are stored, to locate disk blocks where ~~tha~~ a particular file is stored.

(use of Zones)

↳ In the case of immediate files, where using entire disk blocks would be overkill, we stored the file content in this 'i_zone', rather than disk block addresses

— 1 —

→ ~~A zone~~ If zone refers to the ~~storage location~~ record of 1 disk address, then it is of 3 Bytes. Sometimes, we use ~~it~~ (zone) to refer to the entire space where disk addresses are stored. In this case ~~(13) disk addresses can fit, effectively having~~ using 39 Bytes of space

(f) Are there any disadvantages to having too many immediate files?

Solution:

Too many immediate files would make the number of direct addresses larger, but ~~for~~ this would just increase the number of files present on the system, making searching through the files harder ~~ø~~ (i.e., more expensive), ⇒ slower

— 0.5 —

~~[illegible]~~ ~~System would be~~ ↳ (A lot of TLB misses possible ~~if not~~ as well.)

increased fragmentation

Note: A thesis paper from NIT Calicut that ~~sp~~ detailed the implementation of immediate files in MINIX3 "used" 40 Bytes of space here. So, if this 40B were used to store the abovementioned 39B-worth of disk addresses, it is unclear what the remaining 1 Byte is for.