

Devay Rathore

2023190

OS Sec-B End Semester Exam
Monsoon 2024

Max points: 100

Time: 2 hours

1. A filesystem has a block size of 4 KB, and the Inode size is 128 B. The Inode table start address is 0 B, and the size of each sector on the disk is 512 B. What is the block and disk sector associated with Inode number 8000? To simplify the calculations, you can assume 1 KB = 1000 B instead of 1024 B. **5 pts**

2. A disk has an RPM of 5400, and the seek time per track is 10 ms. There are 500 tracks on the disk with 100 sectors per track. The sectors are numbered 0 to 99 on each track. The SCAN (Elevator) algorithm for disk scheduling. The disk head is initially positioned at track 100, sector 50, moving toward track 0. The following I/O requests need to be scheduled. Calculate the seek time, rotational latency, and total time to reach the sector for each request. **10 pts**

Track 20, sector 25
Track 450, sector 70
Track 150, sector 50
Track 45, sector 65
Track 240, sector 99

3. Consider the following sequence of file operations performed on a file descriptor fd. Assume the initial offset is 0. The file size is 52 bytes. **10 pts**

```
1 a write(fd, buffer, 4); +4
2 b lseek(fd, 10, SEEK_SET); 10
3 c write(fd, buffer, 7); 17
4 d lseek(fd, 5, SEEK_CUR); 22
5 e read(fd, buffer, 8); 30
6 f lseek(fd, -10, SEEK_END); 42
7 g write(fd, buffer, 5); 47
8 h lseek(fd, 10, SEEK_END); 62
9 i write(fd, buffer, 5); -
```

(a) What is the offset of the file after each of the above operations? Show your calculations for each step.

(b) What happens if a read operation is attempted beyond the end of the file in this sequence?

4. Using the partial strace output of a program, reconstruct the corresponding C program that produces this output. Ensure the program handles potential errors during system calls gracefully. Do not use additional libraries except the standard C headers. **20 pts**

```

openat(AT_FDCWD, "o1.txt", O_WRONLY|O_CREAT|O_TRUNC, 0644) = 3
dup2(3, 1)
close(3)
openat(AT_FDCWD, "o2.txt", O_WRONLY|O_CREAT|O_TRUNC, 0644) = 3
dup2(3, 2)
close(3)
write(2, "Hello strace! \n", 15) = 15
write(1, "Hello World!\n", 13) = 13

```

5. The following is a description of a common synchronization problem: The Smoker's Problem. There are three smokers and a single agent (a "dealer"). The agent has a set of ingredients (tobacco, paper, and matches), and can place any two of these ingredients on the table. The smokers need all three ingredients to make a cigarette (tobacco, paper, and matches), but each smoker has a particular ingredient they need: One smoker has only tobacco, another has only paper, and the third has only matches. The agent will place two ingredients on the table (so the smokers can take what they need), and then one smoker can take the ingredients and make a cigarette. The problem is to ensure that only one smoker takes the ingredients at a time and that all smokers eventually get a chance to make their cigarette. Given here is the code for the agent function. Explain and address the concurrency and efficiency issues in the code. 15 pts

```

void* agent(void* arg) {
    while (1) {
        int ing1, ing2;
        pthread_mutex_lock(&mutex);

        printf("Agent waiting for smoker to make a cigarette\n");
        //Wait for condition 0 to be signalled
        pthread_cond_wait(&condition[0], &mutex);
        // Randomly choose two different ingredients
        ing1 = rand() % 3;
        do {ing2 = rand() % 3;} while (ing1 == ing2);
        // Update the buffer with the two ingredients
        buffer[0] = ing1; buffer[1] = ing2;
        printf("Agent placed ingredients %d and %d on the
table.\n", ing1, ing2);

        // Signal the appropriate smoker to make the cigarette
        for (int i = 0; i < 3; i++) {
            if (i != ing1 && i != ing2) {
                pthread_cond_wait(&condition[i]);}
            pthread_mutex_unlock(&mutex);
            // Simulate time taken by agent to prepare next
            ingredients
            usleep(1000);
        }
        return NULL;
    }
}

```

6. Consider a process where multiple threads share a common Last-In-First-Out data structure. The data structure is a linked list of "struct node" elements, and a pointer "top" to the top element of the list is shared among all threads. To push an element onto the list, a thread dynamically allocates memory for the struct node on the heap and pushes a pointer to this struct node into the data structure as follows.

```
void push(struct node *n) {
    n->next = top;
    top = n;
}
```

A thread that wishes to pop an element from the data structure runs the following code.

```
struct node *pop(void) {
    struct node *result = top;
    if(result != NULL) top = result->next;
    return result;
}
```

A programmer who wrote this code did not add any kind of locking when multiple threads concurrently access this data structure. As a result, when multiple threads try to push elements onto this structure concurrently, race conditions can occur, and the results are not always what one would expect. Suppose two threads T1 and T2 try to push two nodes, n1 and n2, respectively, onto the data structure at the same time. If all went well, we would expect the top two elements of the data structure would be n1 and n2 in some order. However, this correct result is not guaranteed when a race condition occurs. Describe how a race condition can occur when two threads simultaneously push two elements onto this data structure. Describe the exact interleaving of executions of T1 and T2 that causes the race condition, and illustrate with figures how the data structure would look like at various phases during the interleaved execution.

10 pts

7. Consider a scenario where a bus picks up waiting passengers from a bus stop periodically. The bus has a capacity of K. The bus arrives at the bus stop, allows up to K waiting passengers (fewer if less than K are waiting) to board, and then departs. Passengers have to wait for the bus to arrive and then board it. Passengers who arrive at the bus stop after the bus has arrived should not be allowed to board and should wait for the next time the bus arrives. The bus and passengers are represented by threads in a program. The passenger thread should call the function board() after the passenger has boarded, and the bus should invoke depart() when it has boarded the desired number of passengers and is ready to depart. The threads share the following variables, none of which are implicitly updated by functions like board() or depart().

```
mutex = semaphore initialized to 1.
bus_arrived = semaphore initialized to 0.
passenger_boarded = semaphore initialized to 0.
waiting_count = integer initialized to 0.
```

Below is given synchronized code for the passenger thread. You should not modify this

in any way.

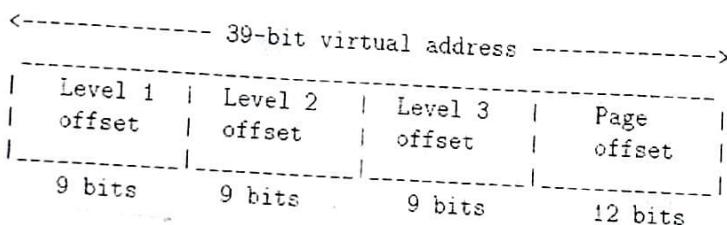
```
sem_wait(mutex)
waiting_count++
sem_post(mutex)
sem_wait(bus_arrived)
board()
sem_post(passenger_boarded)
```

Write the corresponding synchronized code and explain your answer for the bus thread that achieves the correct behaviour specified above. The bus should board the correct number of passengers, based on its capacity and the number of those waiting. The bus should correctly board these passengers by calling sem_post/sem_wait on the semaphores suitably. The bus code should also update the waiting count as required. Once boarding is completed, the bus thread should call depart(). You can use any extra local variables in the code of the bus thread, like integers, loop indices and so on. However, you must not use any other extra synchronization primitives. **20 pts**

8. In the code snippet given below, how many times would printf be executed? **5 pts**

```
int x = 3;
while(x > 0) {
    fork();
    printf("hello");
    wait(NULL);
    x--;
}
```

8. Consider a three-level page table to translate a 39-bit virtual address to a physical address, as shown below:



The page size is $4\text{ KB} = (1\text{KB} = 2^{10}\text{ bytes})$, and the page table entry size at every level is 8 bytes. A process P is currently using 2 GB ($1\text{ GB} = 2^{30}\text{ bytes}$) virtual memory which OS mapped to 2 GB of physical memory. The minimum amount of memory required for the page table of P across all levels is _____ KB. **5 pts**