# 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**

   Inode 8000's relative position = 8000*128 = 1024000
   Block number = 1024000/4k = 1024000/4000 = 256
   Block number if 1KB = 1024 B is used, = 1024000/4*1024 = 1024000/4096 = 250
   Sector = (1024000 + 0)/512 = 2000

   2.5 pts for block number and 2.5 pts for the sector number

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

   The sequence in which requests will be resolved is as follows.

   Track 45 → Track 20 → Track 150 → Track 240 → Track 450

   Seek time = 10 ms
   RPS is 5400/60 = 90
   Time for one rotation: 1/90 = 0.01111 s = 11.11 ms
   Average rotational latency = 0.01111/2 = 0.005 = 5 ms
   Rotational Latency per sector = 11.11 / 100 = 0.1111 ms

   Either average rotational latency or rotational latency per sector can be used for further calculations.

(i) Track 45, Sector 65

- **Seek Time**: |100 - 45| × 10 = 55 × 10 = 550 ms
- **Rotational Latency (a)**: |65 - 50| × 0.1111 = 15 × 0.1111 = 1.67 ms
- **Rotational Latency (b)**: |65 - 50| × 5 = 15 × 5 = 75 ms
- **Total Time**: 550 + 1.67 = 551.67 ms or 550+75 = 625 ms

(ii) Track 20, Sector 25

- **Seek Time**: |45 - 20| × 10 = 25 × 10 = 250 ms
- **Rotational Latency (a)**: |25 - 65| × 0.1111 = 40 × 0.1111 = 4.44 ms
- **Rotational Latency (b)**: |25 - 65| × 5 = 200 ms
- **Total Time**: 250 + 4.44 = 254.44 ms or 250+200 = 450 ms

(iii) Track 150, Sector 50

- **Seek Time**: (|20 - 0| + |150 - 0|) × 10 = 170×10 = 1700 ms
- **Rotational Latency (a)**: |50 - 25| × 0.1111 = 25 × 0.1111 = 2.778 ms
- **Rotational Latency (b)**: |50 - 25| × 5 = 125 ms
- **Total Time**: 1500 + 2.78 = 1502.78 ms or 1500+125 = 1625 ms

(iv) Track 240, Sector 99

- **Seek Time**: |240 - 150| × 10 = 90×10 = 900 ms
- **Rotational Latency (a)**: |99 - 50| × 0.1111 = 49 × 0.1111 = 5.44 ms
- **Rotational Latency (b)**: |99 - 50| × 5 = 245
- **Total Time**: 900 + 5.44 = 905.44 ms or 900 + 245 = 1145 ms

(v) Track 450, Sector 70

- **Seek Time**: |450 - 240| × 10 = 210 × 10 = 2100 ms
- **Rotational Latency (a)**: |70 - 99| × 0.1111 = 29 × 0.1111 = 3.22 ms
- **Rotational Latency (b)**: |70 - 99| × 5 = 145
- **Total Time**: 2100 + 3.22 = 2103.22 ms or 2100 + 145 = 2245 ms

5 pts for identifying the correct sequence and 5 pts for calculating each of the track/sector request.

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**

```
write(fd, buffer, 4);
lseek(fd, 10, SEEK_SET);
write(fd, buffer, 7);
```

```
lseek(fd, 5, SEEK_CUR);
read(fd, buffer, 8);
lseek(fd, -10, SEEK_END);
write(fd, buffer, 5);
lseek(fd, 10, SEEK_END);
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?

(a) Offset after each operation:

1. write(fd, buffer, 4) → **4** (writes 4 bytes, moves forward)
2. lseek(fd, 10, SEEK_SET) → **10** (sets offset to 10)
3. write(fd, buffer, 7) → **17** (writes 7 bytes, moves forward)
4. lseek(fd, 5, SEEK_CUR) → **22** (moves forward by 5)
5. read(fd, buffer, 8) → **30** (reads 8 bytes, moves forward)
6. lseek(fd, -10, SEEK_END) → **42** (sets offset to 42)
7. write(fd, buffer, 5) → **47** (writes 5 bytes, moves forward)
8. lseek(fd, 10, SEEK_END) →62 (sets offset to 10 past the EOF)
9. write(fd, buffer, 5) →67 (writes 5 bytes, moves forward)

(b) read() call beyond the EOF returns 0.

9 points for the (a) part (1 point for each) and 1 point for the (b) part.

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)                                     = 1
close(3)                                       = 0
openat(AT_FDCWD, "o2.txt", O_WRONLY|O_CREAT|O_TRUNC, 0644) = 3
dup2(3, 2)                                     = 2
close(3)                                       = 0
write(2, "Hello strace! \n", 15)       = 15
write(1, "Hello World!\n", 13)         = 13
```

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

```
#include <unistd.h>
#include <fcntl.h>

int main() {
        int stdout_fd, stderr_fd;

        stdout_fd = open("o1.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);   – 1
        if (stdout_fd < 0) {
        perror("Error opening o1.txt");
        exit(EXIT_FAILURE);
        }

        if (dup2(stdout_fd, STDOUT_FILENO) < 0) {                          – 2
        perror("Error");
        close(stdout_fd);
        exit(EXIT_FAILURE);
        }

        close(stdout_fd);                                                  – 3

        stderr_fd = open("o2.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);   – 4
        if (stderr_fd < 0) {
        perror("Error opening o2.txt");
        exit(EXIT_FAILURE);
        }

        if (dup2(stderr_fd, STDERR_FILENO) < 0) {                          – 5
        perror("Error");
        close(stderr_fd);
        exit(EXIT_FAILURE);
        }

        close(stderr_fd);                                                  – 6

        printf("Hello World!\n");                                          – 7
        fprintf(stderr, "Hello strace! \n");                               – 8

        return 0;
}
```

1. **All the 8 system calls are identified from strace and are put in the code (8 pts)**
2. **Appropriate file descriptors are identified and used in all the function calls (8 pts)**
3. **At least some (3 to 4) error handling has been done (4 pts)**

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;}
```

Issues with the given code:

1. **Deadlock:** The agent waits for a smoker (specifically smoker 0) to finish making the cigarette before placing new ingredients.

   ```
   pthread_cond_wait(&condition[0], &mutex);
   ```

   But the smoker cannot proceed because the agent is not placing ingredients. This creates a deadlock, where the agent and the smoker are waiting on each other, introducing a circular dependency.

2. **Efficiency:** Furthermore, the ingredients need not be selected inside the critical section as this is a non-critical task. Only the buffer values must be placed inside the critical section for maximal efficiency.

3. **Semantic Error:** `pthread_cond_signal(&condition[i])` must be used instead of waiting when signaling the appropriate smoker.

The code may be rewritten as follows:

```
void* agent(void* arg) {
    while (1) {

        //NO NEED TO PLACE INGREDIENT INITIALIZATION IN CRITICAL
SECTION

        // Randomly choose two different ingredients
        int ing1 = rand() % 3;
        int ing2;
        do { ing2 = rand() % 3; } while (ing1 == ing2);

       //AGENT DOES NOT WAIT FOR SMOKER

        pthread_mutex_lock(&mutex);

        // 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) {
                // SIGNAL THE SMOKER, DO NOT WAIT
                pthread_cond_signal(&condition[i]);
            }
        }
        pthread_mutex_unlock(&mutex);

        usleep(1000);  // Simulate time taken by the agent to prepare
the next ingredients
    }
    return NULL;
      }
```

```
1. Identify the deadlock issue, explain it and fix the issue
   in the code - 6 points
2. Identify the efficiency issue, explain it and fix the issue
   in the code - 3 points
3. Identify the semantic erro, explain it and fix the issue in
   the code - 6 points.
```

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**

Ans: One possible race condition is as follows. n1’s next is set to top, then n2’s next is set to top. So both n1 and n2 are pointing to the old top. Then top is set to n1 by T1, and then top is set to n2 by T2. So, finally, top points to n2, and n2’s next points to old top. But now, n1 is not accessible by traversing the list from top, and n1 remains on a side branch of the list.

Figures that illustrate the change of top by T1 and T2 work.

Describing a scenario correctly gets 6 points, and 4 points are for the figures describing the race condition.

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**

```
sem_wait(mutex)
N = min(waiting_count, K)
for i= 1 to N
        sem_post(bus_arrived)
        sem_wait(passenger_boarded)
waiting_count = waiting_count - N
sem_post(mutex)
depart()
```

1. If the pseudocode matches exactly, give 20 pts.
2. If a loop is used, along with decrementing the waiting_count, and depart() call, give 12 pts
3. For the correct sequence of sem_wait and sem_post calls, along with the correct semaphore names, give 8-10 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--;
}
```

Printf is executed **14** times ( 8 + 4 + 2)
Iteration 1: 2 processes print "hello" -> 2 calls.
Iteration 2: 4 processes print "hello" -> 4 calls.
Iteration 3: 8 processes print "hello" -> 8 calls.

```
If the answer (14) is correct, 5 pts can be given. Otherwise, 0
pts would be given.
```

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

```
<-------------- 39-bit virtual address -------------->
 _____________________________________________________
|   Level 1   |  Level 2    |  Level 3    |    Page     |
|   offset    |  offset     |  offset     |    offset   |
|_____________|_____________|_____________|_____________|
    9 bits       9 bits         9 bits         12 bits
```

The page size is 4 KB = (1KB = $2^{10}$ bytes), and the page table entry size at every level is 8 bytes. A process P is currently using 2 GB (1 GB = $2^{30}$ 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**

**Virtual address size:** 39 bits.
**Page size:** 4 KB = $2^{12}$ bytes
Offset bits = 12 bits12 \, \text{bits}12bits.

**Page table entry (PTE) size:** 8 bytes
**Virtual memory usage by process P:** 2 GB = $2^{31}$ bytes
**Physical memory mapped:** 2 GB = $2^{31}$ bytes

**1. Number of pages in use:**

Each page is 4 KB = $2^{12}$ bytes, so the number of pages required for 2 GB of virtual memory is:

Number of pages = $2^{31}/2^{12} = 2^{19}$

**2. Page table structure:**

The page table is a **3-level hierarchical table**, with 9 bits used at each level for indexing. Each level contains $2^9$= 512 entries.

**Level 3 (leaf level):**

- Each level-3 entry maps a single page.
- The total number of level-3 entries needed equals the total number of pages: Level-3 entries required = $2^{19}$
- Each entry is 8 bytes, so the memory required for level-3 page tables is: Memory for level-3 tables = $2^{19} \times 8 = 2^{22}$ bytes = 4 MB

**Level 2:**

- Each level-2 entry points to a level-3 table.
- Each level-3 table covers $2^9$ pages because each level-3 table has 512 entries, and each entry maps one page.
- The total number of level-3 tables required is: Number of level-3 tables = $2^{19}/2^9 = 2^{10}$
- Thus, the number of level-2 entries required is $2^{10}$, and each entry is 8 bytes: Memory for level-2 tables = $2^{10} \times 8 = 2^{13}$ bytes = 8 KB.

**Level 1:**

- Each level-1 entry points to a level-2 table.
- Each level-2 table covers $2^9 \times 2^9 = 2^{18}$ pages, so the total number of level-2 tables required is $2^{10}$.
- Each level-1 table covers $2^9 \times 2^9 \times 2^9 = 2^{27}$ pages, which means only one level-1 table is required.
- Memory for level-1 table = 512×8 = $2^9 \times 8 = 2^{12}$ bytes = 4 KB

**3. Total memory required for page tables:**

Adding the memory used across all levels:

Total memory = Level-1 + Level-2 + Level-3.
Total memory = 4 KB + 8 KB + 4 MB.

Convert everything to KB:
Total memory = 4 KB + 8 KB + 4096 KB = 4108 KB

Give 1 point each for calculating the memory requirements at each of the three levels. 1 point for calculating the number of pages and 1 point for computing the total memory.