INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

Stamp / Signature of the Invigilator

EXAMINATION (~~Mid Semester~~ / End Semester)	SEMESTER (~~Autumn~~ / Spring)

Roll Number										Section		Name	
Subject Number	C	S	3	1	2	0	2	Subject Name					*Operating Systems*
Department / Center of the student												Additional Sheets	

Important Instructions and Guidelines for Students

1. You must occupy your seat as per the Examination Schedule/Sitting Plan.
2. Do not keep mobile phones or any similar electronic gadgets with you even in the switched off mode.
3. Loose papers, class notes, books or any such materials must not be in your possession, even if they are irrelevant to the subject you are taking examination.
4. Data book, codes, graph papers, relevant standard tables/charts or any other materials are allowed only when instructed by the paper-setter.
5. Use of instrument box, pencil box and non-programmable calculator is allowed during the examination. However, exchange of these items of any other papers (including question papers) is not permitted.
6. Write on both sides of the answer script and do not tear off any page. **Use last page(s) of the answer script for rough work.** Report to the Invigilator if the answer script has torn or distorted page(s).
7. It is your responsibility to ensure that you have signed the Attendance Sheet. Keep your Admit Card/Identity Card on the desk for checking by the invigilator.
8. You may leave the examination hall for wash room or for drinking water for a very short period. Record your absence from the Examination Hall in the register provided. Smoking and the consumption of any kind of beverages are strictly prohibited inside the Examination Hall.
9. Do not leave the Examination Hall without submitting your answer script to the invigilator. **In any case, you are not allowed to take away the answer script with you.** After the completion of the examination, do not leave the seat until the invigilators collect all the answer scripts.
10. During the examination, either inside or outside the Examination Hall, gathering information from any kind of sources or exchanging Information with others or any such attempt will be treated as '**unfair means**'. Do not adopt unfair means and do not indulge in unseemly behavior.

Violation of any of the above instructions may lead to severe punishment.

Signature of the Student

To be filled in by the examiner

Question Number	1	2	3	4	5	6	7	8	9	10	Total
Marks obtained											

Marks Obtained (in words)	Signature of the Examiner	Signature of the Scrutineer

CS31202 Operating Systems

End-Semester Examination

25-Apr-2025, 02:00pm–05:00pm

Maximum Marks: 100

Instructions to students

- Please write in the spaces provided in the question paper itself. Be brief and precise.
- For rough work, you can use the extra blank pages provided at the end. If you need additional space for rough work, please ask for supplementary sheets from the invigilators.
- Do not write anything on this page. Questions start from the next page (Page 3).

1. [*Process synchronization*]

(a) In a hospital, there are two operating rooms for performing surgeries. Surgeries are of two types: *Emergency* and *Elective*. At any given time, a maximum of two surgeries**,** regardless of type, can be conducted simultaneously. Both the rooms are equally equipped, so any operation can run in any room. However, it must be ensured that emergency surgeries are always given priority over elective ones; that is, if there is an available room and an emergency patient is waiting, the room must be allocated to the emergency patient before considering any elective surgery. Multiple waiting emergency patients may be served in any order. Only if no emergency patient is waiting, an elective operation may go on. Any new patient (emergency or elective) must wait if both the rooms are occupied (for ongoing elective and/or emergency surgeries). The synchronization mechanism must ensure that no more than two surgeries can run at any time, and that elective surgeries may proceed only when no emergency cases are pending and a room is free.

The following pseudocode implements two processes depicting two different surgeries (a) **EmergencySurgery** and (b) **ElectiveSurgery**. The implementation ensures that the processes are free from race condition, and adheres to the required synchronization constraints. Complete the implementation. **[7]**

```
semaphore mutex = ____1____ ;          // Mutual exclusion for shared data
semaphore rooms = ____2____ ;          // Controls access to the two operating rooms
int emergency_waiting = 0;             // Number of emergency patients waiting

ElectiveSurgery() {
    while (true) {
        ____________________wait(mutex)____________________ ;

        if ( ____emergency_waiting == 0____ ) {

            ____signal(mutex)____ ;        // No emergency waiting

            ____wait(rooms)____ ;          // Take a room
            perform_elective_surgery();

            ____signal(rooms)____ ;        // Release operating room
        } else {

            ____signal(mutex)____ ;        // Emergency is waiting – try again later
        }                                  // Busy wait
    }
}

EmergencySurgery() {
    while(true) {
        ____wait(mutex)____ ;

        ____emergency_waiting++____ ;      // Arrival of emergency patient

        ____signal(mutex)____ ;

        ____wait(rooms)____ ;
```

```
        perform_emergency_surgery();
        wait(mutex);

        _____emergency_waiting--_____ ;          // Done with surgery
        signal(mutex);

        _____signal(rooms)_____ ;                // Release operating room
    }
}
```

(b) Suppose that a multiprogramming system runs three concurrent processes P1, P2, and P3 as follows. These three processes share three semaphores a, b, and c initialized as a = 1, b = 0, and c = 0. Compute the maximum number of 0's that will be printed by process P1. Justify your answer. **[3]**

Process P1

```
while (true) {
    wait(a);
    printf("0");
    signal(b);
    signal(c);
}
```

Process P2

```
wait(b);
signal(a);
```

Process P3

```
wait(c);
signal(a);
```

000 (three zeros)

2. [*Deadlock*]

There are five processes P1, P2, P3, P4, P5 and four resource types A, B, C, D each with multiple instances. At some point of time, the resource allocation to the processes, the new requests of the processes, and the available resource instances are as follows.

	ALLOCATION					REQUEST					AVAILABLE			
	A	B	C	D		A	B	C	D		A	B	C	D
P1	1	2	0	3	P1	3	0	1	2		1	0	2	0
P2	5	2	3	0	P2	2	0	1	3					
P3	1	2	3	3	P3	4	1	2	3					
P4	0	3	2	6	P4	1	0	3	2					
P5	0	5	5	2	P5	1	0	0	1					

(a) Establish that the system is in a deadlocked state. Show all your calculations. **[4]**

Neither of the following inequalities is valid in the given situation.

Request1 = (3, 0, 1, 2) ⩽ AVAILABLE = (1, 0, 2, 0) [resources A and D]
Request2 = (2, 0, 1, 3) ⩽ (1, 0, 2, 0) [resources A and D]
Request3 = (4, 1, 2, 3) ⩽ (1, 0, 2, 0) [resources A, B, and D]
Request4 = (1, 0, 3, 2) ⩽ (1, 0, 2, 0) [resources C and D]
Request5 = (1, 0, 0, 1) ⩽ (1, 0, 2, 0) [resource D]

So no request can be granted.

(b) In order to recover from the deadlock, the OS plans to kill process(es) and release the resources allocated to these processes to the AVAILABLE pool. In order to maximize the possibility of quickly breaking the deadlock, the OS adopts a greedy strategy. For killing, it first chooses a process which cannot finish (as per the deadlock-detection algorithm), and which holds the maximum number of resource instances (total of the four resource types). If releasing the resources of this process still leaves the system in a deadlocked state, another process which cannot finish and holds the maximum total number of resources (among the remaining processes) is chosen and is killed. This greedy loop continues until the system is no longer in a deadlocked state. Show the working of this algorithm on the situation described above. Show the iterations of the greedy loop one after another along with all relevant calculations. Show also that after the loop terminates, the system does not have a deadlock (clearly work out a sequence in which the remaining processes can finish). **[10]**

Iteration 1

P5 is the holder of maximum total number of resources (0 + 5 + 5 + 2 = 12). Killing it changes

AVAILABLE = (1, 0, 2, 0) + (0, 5, 5, 2) = (1, 5, 7, 2).

Now, P4 can be granted its REQUEST4 = (1, 0, 3, 2). Completion of P4 changes

AVAILABLE = (1, 5, 7, 2) + (0, 3, 2, 4) = (1, 8, 9, 6).

But then, the requests of neither P1 nor P2 nor P3 can be granted (resource A). So the system is still in a deadlock.

Iteration 2

After P5 is killed, we have AVAILABLE = (1, 5, 7, 2). Now, P4 holds the largest amount of total resources. But we have already seen that P4 can be granted its request, that is, P4 is not involved in a deadlock. We need to choose the next victim from P1, P2, P3. Among these, the holder of the maximum number of resources is P2 (5 + 2 + 3 + 0 = 10). Preempting/Killing P2 changes

AVAILABLE = (1, 5, 7, 2) + (5, 2, 3, 0) = (6, 7, 10, 2).

We also have REQUEST1 = (3, 0, 1, 2), REQUEST3 = (4, 1, 2, 3), and REQUEST4 = (1, 0, 3, 2). With the available resources REQUEST1 and REQUEST4 can be granted. If P1 is allowed to complete after its request is served, we have

AVAILABLE = (6, 7, 10, 2) + (1, 2, 0, 3) = (7, 9, 10, 5).

REQUEST3 can now be served. When P3 completes, we have

AVAILABLE = (7, 9, 10, 5) + (1, 2, 3, 3) = (8, 11, 13, 8).

Serving REQUEST4 and allowing P4 to complete gives

AVAILABLE = (8, 11, 13, 8) + (0, 3, 2, 6) = (8, 14, 15, 14).

(c) We know that is real life, greed usually does not pay in the long run. Consider again the original situation described at the beginning of this exercise. Is it possible to recover from the deadlock by killing a single process? If there is one, work out a sequence in which the remaining processes can finish. If there is no such process, show that the killing of every single process leaves the system in a deadlock. **[6]**

Let us kill P1 first (although it holds the minimum total number of resources). We then have

AVAILABLE = (1, 0, 2, 0) + (1, 2, 0, 3) = (2, 2, 2, 3).

Since REQUEST2 = (2, 0, 1, 3) ⩽ (2, 2, 2, 3), P2 can complete, giving

AVAILABLE = (2, 2, 2, 3) + (5, 2, 3, 0) = (7, 4, 5, 3).

Now, REQUEST3 = (4, 1, 2, 3) ⩽ (7, 4, 5, 3), so P3 can finish, giving

AVAILABLE = (7, 4, 5, 3) + (1, 2, 3, 3) = (8, 6, 8, 6).

But then REQUEST4 = (1, 0, 3, 2) ⩽ (8, 6, 8, 6), that is, the completion of P4 gives

AVAILABLE = (8, 6, 8, 6) + (0, 3, 2, 6) = (8, 9, 10, 12).

Finally, REQUEST5 = (1, 0, 0, 1) ⩽ (8, 9, 10, 12). Therefore P5 can finish too giving

AVAILABLE = (8, 9, 10, 12) + (0, 5, 5, 2) = (8, 14, 15, 14).

Since all the processes can finish, the system is not in a deadlock after the P1 is killed.

3. **[*Main memory*]**

(a) A system implements the contiguous memory allocation scheme. At some point of time, there are only two available holes of sizes 300MB and 500MB. Then, three processes P1, P2, and P3 arrive (in that order) with memory requirements x MB, y MB, and z MB, respectively. Supply explicit integer values to x, y, and z in order to demonstrate each of the following two situations. In each case, also explain the details how hole allocations proceed or fail. No credit for trying to prove that some values of x, y, z (may) exist. Assume that when a hole is given to a process which does not fill the entire hole, only one new hole is created (at one end of the old hole).

Situation 1: The best-fit strategy can give the required memory to all of the three processes, whereas the worst-fit strategy can give the required memory only to P1 and P2 (P3 has to wait). **[3]**

$x =$ 200 , $y =$ 100 , $z =$ 450

Explanation:

Best-fit: P1 is given memory from the 300MB hole, resulting in two holes of sizes 100MB, 500MB. P2 is now given the 100MB hole, and P3 is allocated in the 500MB hole (leaving a single hole of size 50MB).

Worst-fit: P1 is given memory from the 500MB hole, resulting in two holes of sizes 300MB each. P2 can now be placed in any of the two holes, leaving two remaining holes of sizes 200MB and 300MB. Neither is big enough to accommodate P3.

Situation 2: The worst-fit strategy can give the required memory to all of the three processes, whereas the best-fit strategy can give the required memory only to P1 and P2 (P3 has to wait). **[3]**

$x =$ 200 , $y =$ 300 , $z =$ 250

Explanation:

Best-fit: P1 is given memory from the 300MB hole, resulting in two holes of sizes 100MB, 500MB. P2 is now given space from the 500MB hole, and the two remaining holes are of sizes 100MB and 200MB. Neither can accommodate P3.

Worst-fit: P1 is given memory from the 500MB hole, resulting in two holes of sizes 300MB each. P2 can now be placed in any of the two holes, leaving a single hole of size 300MB. P3 can be accommodated in that hole (leaving a single hole of size 50MB).

(b) Consider the IA32 segmentation scheme along with paging of the linear address space into 4KB pages. Suppose that a process has one thousand 1KB segments numbered 0, 1, 2, . . . , 999, one hundred 10KB segments numbered 1000, 1001, 1002, . . . , 1099, ten 100KB segments numbered 1100, 1101, 1102, . . . , 1109, and one 1MB segment numbered 1110. Assume that all these segments are local to the process, and that the process uses no global segments. In the linear address space, the segments are placed in the sequence of the segment numbers, starting from linear address 0, and are aligned at page boundaries. Suppose that all these pages are loaded to memory frames (no demand paging). In this context, answer the following questions. Show your calculations.

How much space is lost due to internal fragmentation? **[2]**

Each of the 1KB segments is mapped to a page/frame of size 4KB, leading to a total internal fragmentation of 3000KB for these segments.

Each of the 10KB segments needs three 4KB pages/frames, so the total internal fragmentation for these segments is 200KB.

Since 100KB and 1MB are exact multiples of 4KB, no internal fragmentation results from segments of these sizes.

To sum up, total internal fragmentation is 3000KB + 200KB = 3200KB.

What is the total size of the linear address space (including internal fragmentation)? **[2]**

$1000 \times 4KB + 100 \times 12KB + 10 \times 100KB + 1MB = 7224KB$.

The linear address space is paged. What will be the size of the page table (that is, the number of entries in the page table) of the process? **[2]**

7224KB / 4KB = 1806.

IA32 uses two-level hierarchical paging using the 10 + 10 + 12 breakup of the form page-directory (outer-page) index + page-table (inner-page) index + offset. Also suppose that each page-table entry is of size 32 bits. Explain how the linear address space of the process is paged. Clearly calculate the numbers of entries needed in the page directory and in all the (inner) page tables. **[3]**

One frame can store 4KB / 4B = 1024 entries. Two entries are needed in the page directory (outer page table) for 1806 pages. The first entry will point to a page table with 1024 frame pointers, and the second entry will point to a page table with 1806 – 1024 = 782 frame pointers.

Explain how the logical address (1024, 4201) [this is in the format (segment number, offset)] is converted to a linear address and then to a physical address. **[3]**

The segment 1024 is a 10KB segment. Before it, appear 1000 1KB segments (each having one page) and 24 10KB segments (each having three pages). So this segment starts from page 1000 + 3 × 24 = 1072 of the linear address space. The offset is 4201 = 4096 + 105. So the linear address is (1073, 105). As a single 32-bit value, the linear address is 1073 × 4096 + 105 = 4395113.

In order to map this linear address to a physical address, we look at the page directory. Since 1073 > 1024, we follow the second entry of the page directory. From this page table, we look at the entry 1073 – 1024 = 49 (numbering starts from 0). The page-table entry gives the frame number of the address we started with. The offset in that frame will be 105.

4. [*Virtual memory*]

(a) Consider the code snippet below.

```
#define Size 64
int A[Size][Size], B[Size][Size], C[Size][Size];
int register i, j;
for (j = 0; j < Size; j++)
    for (i = 0; i < Size; i++)
        C[i][j] = A[i][j] + B[i][j];
```

The code runs on a virtual-memory-based system (with 1KB page size) implementing the LRU page-replacement algorithm. Assume that the memory-management unit allocates 4 frames to the process running the above code. Frame 0 is allocated to store the code, whereas Frames 1, 2, 3 are used to store the data. Also, two registers are assigned for the indices i and j (so no memory accesses are needed for references to these two variables). Finally, assume that each integer is 4 bytes long.

Compute the total number of page faults that would occur. Show all calculations. **[4]**

Number of pages for an array 64 × 64 × 4 / 1K = 16. Each page stores 4 rows.
As an example, A[0][0] – A[0][63], A[1][0] – A[1][63], A[2][0] – A[2][63], and A[3][0] – A[3][63] will be stored in the first data page.
A similar storage pattern can be derived for the rest of array A and for arrays B and C.

Frame 1 stores A[0][0] to A[3][63]. Same for Frame 2: B[0][0] to B[3][63], Frame 3: C[0][0] to C[3][63]

First iteration of j = 0 access A[0][0], A[1][0], A[2][0], . . . , A[63][0] So, 3 page faults for A[0] – A[3], B[0] – B[3], C[0] – C[3]. Next 3 page faults for A[4] – A[7], B[4] – B[7], C[4] – C[7]. Total 3 × 16 = 48 page faults for j = 0. For 64 iterations of j: Total number of page faults = 48 × 64 = 3072.

Can you modify the program to minimize the number of page faults? Write the revised code. **[1]**

Swap the loops on i and j. That is, make the outer loop on i, and the inner loop on j.

Compute the number of page faults after your modification. Show all the steps. **[3]**

i = 0 access A[0][0], A[0][1], . . . , A[0][63]. Three page faults for i = 0 to i = 3: A[0][0] – A[3][63], B[0][0] – B[3][63], C[0][0] – C[3][63]
For 64 iterations of i, total number of page faults = 3 × 16 = 48.

(b) Prove or disprove with argument: "LRU approximation algorithm (implemented as the second-chance algorithm) does not suffer from Belady's anomaly." **[2]**

False. Second-chance algorithm may degenerate to FIFO.

(c) Consider a virtual-memory system which implements Enhanced Second-Chance page-replacement algorithm, with one reference bit and one dirty bit per page. The number of page frames is 4 (all initially empty). All the dirty bits and reference bits are initially 0. Consider the page-reference string

0, 1, 3(W), 6, 2, 4, 5, 2(W), 5, 0(W), 3, 1(W)

generated by a process P. In the string, an integer x indicates that the page x is accessed in the read-only mode, whereas x(W) indicates that the process P modifies the page x. Assume that the pointer (clock hand) is initially at Frame 0. The page-replacement algorithm replaces a page with (reference bit, dirty bit) equal to (0, 0) or (0, 1) only. If (0, 0) is found during one full rotation, the corresponding page is replaced. If not, yet another full rotation is made to find a (0, 0) entry for replacement. If that attempt fails too, a third rotation is carried out, and the first (0, 1) entry is chosen for replacement. After a page replacement, the pointer moves to the frame cyclically next to the replacement position.

For each page reference, show the contents of frames 0, 1, 2, 3 just after the reference, along with the reference-bit and dirty-bit values for the pages stored in all frames (so your answer should look like a 2-d table, with 4 columns (one for each frame) and 12 rows (one for each page in the reference string). Identify the page faults, and find the total number of page faults. Show the step-by-step procedure in the table. **[5]**

Ref #	Page	F0	F1	F2	F3	Page fault?
1	0	0 (1, 0)	–	–	–	Yes
2	1	0 (1, 0)	1 (1, 0)	–	–	Yes
3	3	0 (1, 0)	1 (1, 0)	3 (1, 1)	–	Yes
4	6	0 (1, 0)	1 (1, 0)	3 (1, 1)	6 (1, 0)	Yes
5	2	2 (1, 0)	1 (0, 0)	3 (0, 1)	6 (0, 0)	Yes
6	4	2 (1, 0)	4 (1, 0)	3 (0, 1)	6 (0, 0)	Yes
7	5	2 (1, 0)	4 (1, 0)	3 (0, 1)	5 (1, 0)	Yes
8	2	2 (1, 1)	4 (1, 0)	3 (0, 1)	5 (1, 0)	No
9	5	2 (1, 1)	4 (1, 0)	3 (0, 1)	5 (1, 0)	No
10	0	2 (0, 1)	0 (1, 0)	3 (0, 1)	5 (0, 0)	Yes
11	3	2 (0, 1)	0 (1, 0)	3 (0, 1)	5 (0, 0)	No
12	1	2 (0, 1)	0 (1, 0)	3 (0, 1)	1 (0, 0)	Yes

(d) Consider a virtual-memory system with page size 200 bytes, which implements a working-set model with a working-set window of size $\Delta = 4$. A process P generates the following virtual addresses at 11 consecutive time steps:

1020, 1312, 1578, 1110, 1222, 330, 1114, 1362, 1570, 1036, 222.

Only three frames are allocated to the process P for execution. Using the working set, determine when thrashing occurs during the execution of the process P. Clearly show the working set and frame allocation to pages at each time step, and identify the points of thrashing (that is, the references for which the three allocated frames cannot accommodate the entire working set). **[5]**

Reference string:
5, 6, 7, 5, 6, 1, 5, 6, 7, 5, 1

Time 1: Page 5 → Working Set = [5] → No Thrashing	Frames allocated to (5)
Time 2: Page 6 → Working Set = [5, 6] → No Thrashing	Frames allocated to (5, 6)
Time 3: Page 7 → Working Set = [5, 6, 7] → No Thrashing	Frames allocated to (5, 6, 7)
Time 4: Page 5 → Working Set = [5, 6, 7] → No Thrashing	Frames allocated to (5, 6, 7)
Time 5: Page 6 → Working Set = [5, 6, 7] → No Thrashing	Frames allocated to (5, 6, 7)
Time 6: Page 1 → Working Set = [5, 6, 7, 1] → Thrashing	Frames allocated to (1, 5, 6)
Time 7: Page 5 → Working Set = [6, 1, 5] → No Thrashing	Frames allocated to (1, 5, 6)
Time 8: Page 6 → Working Set = [1, 5, 6] → No Thrashing	Frames allocated to (1, 5, 6)
Time 9: Page 7 → Working Set = [1, 5, 6, 7] → Thrashing	Frames allocated to (5, 6, 7)
Time10: Page 5 → Working Set = [5, 6, 7] → No Thrashing	Frames allocated to (5, 6, 7)
Time11: Page 1 → Working Set = [5, 6, 7, 1] → Thrashing	Frames allocated to (1, 5, 7)

5. [*File systems*]

(a) Consider an i-node-based organization of a Unix file system. The i-node is stored inside a disk block, and in this system, an i-node is always accessed from the disk only (that is, an i-node never gets buffered in the memory). Assume that 192 bytes of an i-node are used to store the file attributes. Inside the i-node, there exists one single indirect pointer, one doubly indirect pointer, and one triply indirect pointer. The rest of the i-node stores direct block pointers. Disk block size is of 8KB, disk block pointer is 32 bits long, and disk bandwidth is 16KB/Sec. Estimate the minimum and maximum file sizes, whose random/direct access takes exactly 1.5 secs. **[5]**

Inside i-node, pointer storage space 8192 – 192 = 8000 bytes

Number of pointers inside i-node = 8000 / 4 = 2000 (total)

So, number of direct pointers = 2000 – 3 = 1997

Bandwidth 16KB/sec. Block size 8KB. Direct access takes 1.5 sec means it requires 3 block access (one i- node, one single indirect, one data block)

Minimum and maximum file size which is accessed via single indirect pointers:

Min: 1997 × 8KB + 1 = 1997 × 8KB + 1 byte = 16,358,401 bytes

Max: 1997 × 8KB + 2048 × 8KB = 1997 × 8KB + 2048 × 8KB = 32,360 KB ≈ 31.6 MB

(b) The disk free-space manager can be implemented using either a linked (grouping) or a bit-map scheme. Assume that disk addresses require D bits each, and each block is of size x bits. For a disk with B blocks, F of which are free, deduce a bound on F, under which the linked (grouping) takes less space than the bit map. (Note that the free-block indices are stored in free blocks only, and as such do not *waste* space. However, you need to calculate how many *free* blocks are needed for storing all the indices, because the free-space manager needs to use these blocks.) **[3]**

Approximate analysis:

Grouping: Disk block size is x bits. Disk block address D. Number of free block addresses that can be accommodated in a block is (x / D). Number of free blocks F can be accommodated in F/(x/D) blocks.

Bitmap: B blocks need B bits = B / x blocks

Condition: $F \times D < B$

(c) Prove or disprove with argument: "Linked allocation of files is equally effective as contiguous allocation in terms of (i) sequential access of a file, (ii) storage-space utilization." **[4]**

(d) Suppose that in a Unix system, a file "test.txt" is stored at the location /usr/www/test.txt. A process P1 opens the file and performs read operation.

With the help of a schematic diagram, clearly show the update made to the System Wide Open File Table (SWOFT) and Per-Process Open File Table (PPOFT) during the opening of the file. Moreover, show how exactly P1 locates the data blocks of "test.txt" to read its content. Clearly mark the access of each i-node block and data blocks during each access of the file. **[2]**

Assume that another process P2 opens the same file "test.txt", and then P1 and P2 close the file in that sequence. Show the updates in SWOFT and PPOFT after each of these open/close operations. **[2]**

P2 opens: SWOFT increments the count of attached processes in the entry for test.txt, and the same entry (index) is returned for storage in the PPOFT of P2.

P1 closes: PPOPT of P1 removes the local file descriptor for file.txt. Also, the SWOFT count corresponding to the file is decremented.

P2 closes: PPOPT of P1 removes the local file descriptor for file.txt. Also, the SWOFT count corresponding to the file is decremented. Moreover, if that count reduces to zero, the entry for test.txt is deleted from SWOFT.

(e) Consider a disk of size 256GB, which implements FAT to maintain the file system. The size of the FAT is 16 MB, where each FAT entry is of size 4 bytes, and accessing each FAT entry takes 2ms time (the FAT is buffered in the main memory). Reading each data block takes 100ms. Accessing the directory entry for the file takes 10ms. Compute the time to read a file of size 200KB from the disk. **[4]**

Number of entries in the FAT table: 16MB/4=4M

Disk block size: 256GB / 4M = 64KB

File size: 200KB requires 4 Blocks

File access:

Directory access (10ms) + first data block (100ms) + FAT for second block (2ms) + second data block (100ms) + FAT for third block (2ms) + third data block (100ms) + FAT for forth block (2ms) + forth data block (100ms) + FAT for fifth block (NULL) (2ms)
= 10ms + 100ms × 4 + 2ms × 4 = 418ms

6. [*Storage Management*]

A file is copied from the main hard drive *D* of a computer to an external hard drive *E*. Both these drives are magnetic, but *E* is much slower than *D*. Here are the technical specifications of the two drives.

	Disk *D*	Disk *E*
Rotational speed in rpm (rotations per minute)	6000 rpm	1500 rpm
Seek time for moving to an adjacent cylinder	2 ms	3 ms
Number of cylinders	128	96
Capacity of each sector/block	2 KB	1 KB

The file in question occupies eight blocks on *D* stored in the cylinders

100, 32, 15, 86, 109, 57, 76, 97.

The copy of the file on *E* requires twice as many blocks. Let these be

51, 89, 3, 61, 34, 57, 72, 75, 11, 9, 40, 29, 47, 68, 16, 23, respectively.

There are two OS processes (demons) handling reads from *D* and writes to *E* in parallel. Whenever a 2KB block from *D* is read, it is divided into two 1KB blocks, and these two blocks are stored in the buffer for *E*. The copy processes start at time 0 with the above lists of cylinder numbers available to both the processes. Writing to *E* waits until some blocks are available in the buffer for *E*. Reading from *D* does not need to wait. Write to *E* proceeds in batches of blocks as explained below. To start with, a block is read from *D*, and two blocks are written in the buffer for *E*, so the writing process can start writing its first batch (of two blocks). By the time the batch write finishes, several other blocks are expected to have been made available by the reader demon, in the buffer for *E* (*D* is faster than *E*). The writer demon for *E* reads the list, and writes this second batch to *E*. When writing this batch finishes, the writer demon again looks at the list of blocks in the third batch, waiting for write to *E*. This process continues until all of the 16 blocks are written to *E*. If the reading of a block from *D* finishes at exactly the same time when writing a batch finishes, include the two new blocks for *E* in the next batch for writing.

Blocks from both *D* and *E* are accessed by a variant of the SCAN algorithm explained now. A list of cylinders to read/write is available. For *D*, it is the entire list of eight cylinders. For *E*, it is the list of cylinders corresponding to the next batch to write. There is also a known position of the read/write head. Our SCAN algorithm first decides the direction (to higher- or lower-numbered cylinders) to which the head should move first, in order to minimize the total seek time for the entire list. It then moves to that direction. As soon as it accesses (reads/writes) the last cylinder of the list in that direction (not the end of cylinders in the disk), it turns back, and accesses the remaining cylinders in the list. If there are no cylinders (from the list) left to be accessed after the first movement, processing the batch stops. In any case, the head waits at the position of the last disk access.

Assume that the rotational latency plus the time to transfer data from/to a given cylinder is half the time for a single rotation. Assume also that at the beginning of the copy process, the disk head for *D* is at cylinder 69, and the disk head for *E* is at cylinder 55. Clearly work out the chronological sequence of the reads and writes happening in the copy of the file. At what time (relative to the start at time 0, in ms) does the copy end? **[12]**

First, let us compute the single rotation times of the two disks.

Disk D: 60000 / 6000 = 10 ms. So half rotation time is 5 ms. Disk E: 60000 / 1500 = 40 ms. So half rotation time is 20 ms.

Then, let us devise an algorithm for choosing the direction of first move of the read/write head on a disk. Let h be the cylinder number where the head is at the beginning of processing a batch, l the minimum cylinder number in the batch, and r the maximum cylinder number in the batch. Consider three cases:

Case 1: $h \leqslant l \leqslant r$. The head will move to higher cylinder numbers, and the total seek time will be $r - h$ (in number of cylinder changes).
Case 2: $l \leqslant r \leqslant h$. The head will move to lower cylinder numbers, and the total seek time will be $h - l$.
Case 3: $l \leqslant h \leqslant r$. If the head moves to lower cylinder numbers, total seek time is $(h - l) + (r - l)$. If the head moves to higher cylinder numbers, total seek time is $(r - h) + (r - l)$. We compare $h - l$ with $r - h$. Whichever is smaller will dictate the choice of the initial direction of head movement. In case of a tie, we can choose any direction arbitrarily.

Let us now look at the process that reads from D. Its progress can be independently worked out. The cylinders to access and the head position (underlined) in sorted order is: 15, 32, 57, 69, 76, 86, 97, 100, 109. We have 69 – 15 = 54 > 109 – 69 = 40. So the head will move toward higher cylinder numbers, and will cover the cylinders in the order: **76, 86, 97, 100, 109, 57, 32, 15**. The timeline of the activity of the read head on D therefore looks as follows.

Read from *D*

Head	Seek + rot latency + access	Time	Blocks ready to write
69	–	0	–
76	14 + 5 = 19	19	47, 68
86	20 + 5 = 25	44	72, 75
97	22 + 5 = 27	71	16, 23
100	6 + 5 = 11	82	51, 89
109	18 + 5 = 23	105	11, 9
57	104 + 5 = 109	214	40, 29
32	50 + 5 = 55	269	3, 61
15	34 + 5 = 39	308	34, 57

Finally, let us look at the writer process for E. At time 19, it starts with the head at cylinder 55 and a batch 47, 68. We have 55 – 47 = 8 < 68 – 55 = 13, so the head will move to lower cylinder numbers first.

Write to *E* (Batch 1)

Head	Seek + rot latency + access	Time
55	–	19
47	24 + 20 = 44	63
68	63 + 20 = 83	146

By this time, blocks available for writing are 72, 75, 16, 23, 51, 89, 11, 9. The sorted list with head position is: 9, 11, 16, 23, 51, 68, 72, 75, 89. We have 68 – 9 = 59 > 89 – 68 = 21. So the head will first move to higher cylinder numbers.

Write to *E* (Batch 2)

Head	Seek + rot latency + access	Time
68	–	146
72	12 + 20 = 32	178
75	9 + 20 = 29	207
89	42 + 20 = 62	269
51	114 + 20 = 134	403
23	84 + 20 = 104	507
16	21 + 20 = 41	548
11	15 + 20 = 35	583
9	6 + 20 = 26	609

When this batch finishes, all the remaining blocks are available for writing. In sorted order, the list is: 3, 9, 29, 34, 40, 57, 61. The third batch completes the write process as follows.

Write to *E* (Batch 3)

Head	Seek + rot latency + access	Time
9	–	609
3	18 + 20 = 38	647
29	78 + 20 = 98	745
34	15 + 20 = 35	780
40	18 + 20 = 38	818
57	51 + 20 = 71	889
61	12 + 20 = 32	921

Space for rough work

Space for rough work

Space for rough work