INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

Stamp / Signature of the Invigilator

EXAMINATION (End Semester)										**SEMESTER (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 or 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/CS30002 Operating Systems, Spring 2023–2024

End-Semester Test

24–April–2024 | 02:00pm–05:00pm | Maximum marks: 60

[*Write your answers in the question paper itself. Be brief and precise. Answer all questions.*]

Do not write anything on this page.

Questions start from the next page.

1. [Process synchronization]

(a) Assume that one or more producer process(es) insert items to a shared buffer of bounded size, and one or more consumer process(es) extract items from that buffer. Prof. Foo proposes the following solution to this problem. Prove/Disprove with proper justification whether this solution works. **(2)**

```
shared semaphore n = 0;
shared semaphore s = 1;
void producer()
{
    while (true) {
       item = produce_item();
       wait(s);
       insert_to_buffer(item);
       signal(s);
       signal(n);
    }
}
```

```
void consumer()
{
    while (true) {
        wait(n);
        wait(s);
        item = extract_from_buffer();
        signal(s);
        consume(item);
    }
}
```

Solution The code does not take into account the number of items stored in the buffer. As a result, buffer overflow may happen. Note that this solution works for the unbounded-buffer variant of the producer-consumer problem.

(b) Four concurrent processes P, Q, R, S are running on a system. All these four processes access a shared variable x initialized to zero. These four processes are implemented as follows. Each of the processes P and Q reads x from memory, increments by one, stores it to memory, and then terminates. Each of the processes R and S reads x from memory, decrements by two, stores it to memory, and then terminates. Each process invokes the `wait()` operation on a counting semaphore `count` before reading x, and invokes the `signal()` operation on the semaphore `count` after storing x to memory. The semaphore `count` is initialized to two (2). What is the maximum possible value of x after all processes complete execution? Explain your answer. **(4)**

Solution Processes can run in many ways. Below is one of the cases in which x attains max value.

Semaphore `count` is initialized to 2.

Process P executes making $count = 1$ and $x = 1$, but it does not update the x variable.

Then, process R executes making $count = 0$. It decrements x, stores $x = -2$, and signals semaphore making $count = 1$.

Now, process S runs completely making $count = 0$, $x = -4$, and finally $count = 1$ (after signal).

Now, process P stores $x = 1$ and signals the semaphore to $count = 2$.

Finally, process Q making $x = 2$.

Evidently, x cannot store a value larger than 2, since only two processes increment it (by 1).

(c) In the sleeping barber's problem, consider the following code snippet of the barber and each customer. The barber waits on the counting semaphore **`customers`** (initialized to 0), and each customer waits on the counting semaphore **`barbers`** (initialized to 0). The binary semaphore **`mutex`** (initialized to 1) is used for the mutually exclusive access of the shared variable **`waiting`**.

```
void barber ( void )
{
    while (true) {
        wait(&customers);
        wait(&mutex);
        waiting = waiting - 1;
        signal(&barbers);
        signal(&mutex);
        cut_hair();
    }
}
```

```
void customer ( void )
{
    wait(&mutex);
    if (waiting < CHAIRS) {
        waiting = waiting + 1;
        signal(&customers);
        signal(&mutex);
        wait(&barbers);
        get_haircut();
    } else {
        signal(&mutex);
    }
}
```

Explain the impact of swapping **`wait(&customers)`** with **`wait(&mutex)`** in the barber code (the customer code remains as shown above). **(2)**

Solution Suppose that at some point of time, there are no customers, so we have the semaphore **`customers`** $= 0$. Now, the barber first locks **`mutex`**, and starts waiting on **`customers`**. Subsequently, any new customer cannot lock **`mutex`**, and keeps on waiting. So the program hangs.

Explain the impact of swapping **`signal(&mutex)`** with **`wait(&barbers)`** in the customer code (the barber code remains as shown above). **(2)**

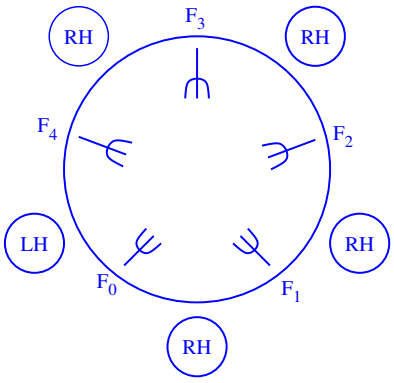
Solution Suppose again that at some point of time, there are no customers, so the barber is waiting on **`customers`**. A new customer comes, locks **`mutex`**, and wakes up the barber by signaling **`customers`**. Then, the customer waits on **`barbers`** without releasing **`mutex`**, so the barber after waking up cannot lock **`mutex`**, and waits indefinitely. That is, the program hangs again.

2. [Deadlock]

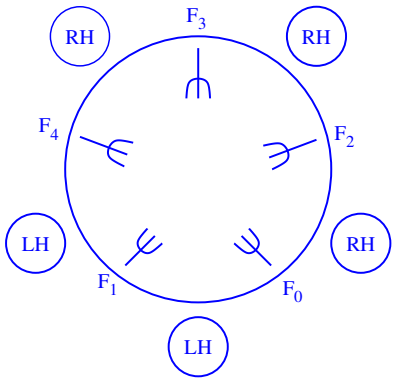
(a) Consider the dining philosophers problem with five philosophers. Some of the philosophers (at least one) is/are right-handed, and the other(s) (at least one) is/are left-handed. A right-handed philosopher first picks the left fork and then the right fork, whereas a left-handed philosopher first picks the right fork and then the left fork. With proper justification, prove/disprove whether deadlock is possible in this case. If deadlock is possible, you must clearly mention a sequence which leads to the deadlock. If deadlock is not possible, furnish a formal proof (handling only some specific situations does not construct a proof). **(4)**

Solution [*Deadlock is not possible*]

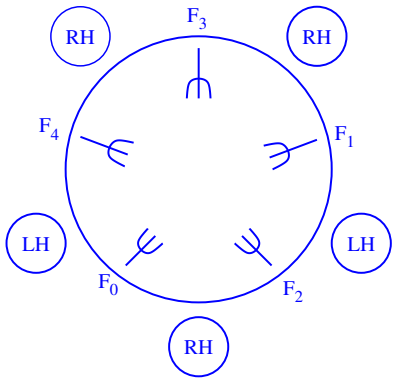
The numbers of left-handed and right-handed philosophers can be one of $1+4$, $2+3$, $3+2$, and $4+1$. We give a proof for the first two cases. Because of circular symmetry, the other two cases can be analogously handled. In the case $2+3$, there are two sub-cases based on whether the two left-handed philosophers sit adjacent to each other or not. In each of these cases, we number the forks as in the figure below.



One left–handed philosopher



Adjacent

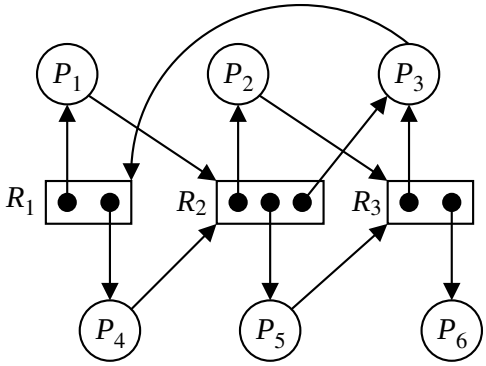


Not adjacent

Two left–handed philosophers

With this numbering, each philosopher first picks the smaller-numbered fork and then the larger-numbered fork adjacent to him/her. But then, deadlock is not possible. A proof is given in the book. In short, if a deadlock involves k philosophers for $2 \leqslant k \leqslant 5$, then there is a cycle in the resource-allocation graph, and as we traverse along the cycle, the fork numbers increase monotonically, which is impossible.

(b) On the next two pages, two resource-allocation graphs are given. For each graph, find the *Allocation* and *Need* matrices and the *Available* vector corresponding to the graph. Subsequently, run the deadlock-detection algorithm to determine whether each graph corresponds to a deadlock situation or not. Solving the problem using any other method will deserve no credit. **(3 + 3)**



Solution [*No deadlock*]

We have

	Allocation		
	R_1	R_2	R_3
P_1	1	0	0
P_2	0	1	0
P_3	0	1	1
P_4	1	0	0
P_5	0	1	0
P_6	0	0	1

	Need		
	R_1	R_2	R_3
P_1	0	1	0
P_2	0	0	1
P_3	1	0	0
P_4	0	1	0
P_5	0	0	1
P_6	0	0	0

Available		
R_1	R_2	R_3
0	0	0

We start with $Finish[i] = False$ for $1 \leqslant i \leqslant 6$.

$Need_6 \leqslant Available$, so we set $Finish[6] = True$ and $Available = [0 \quad 0 \quad 1]$.

$Need_2 \leqslant Available$, so we set $Finish[2] = True$ and $Available = [0 \quad 1 \quad 1]$.

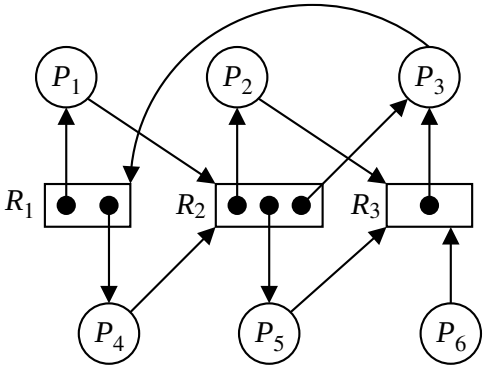
$Need_1 \leqslant Available$, so we set $Finish[1] = True$ and $Available = [1 \quad 1 \quad 1]$.

$Need_3 \leqslant Available$, so we set $Finish[3] = True$ and $Available = [1 \quad 2 \quad 2]$.

$Need_4 \leqslant Available$, so we set $Finish[4] = True$ and $Available = [2 \quad 2 \quad 2]$.

$Need_5 \leqslant Available$, so we set $Finish[5] = True$ and $Available = [2 \quad 3 \quad 2]$.

Since $Finish[i] = True$ for all $1 \leqslant i \leqslant 6$, the system is not in a deadlock state.



Solution [*Deadlock*]

We have

Allocation

	R_1	R_2	R_3
P_1	1	0	0
P_2	0	1	0
P_3	0	1	1
P_4	1	0	0
P_5	0	1	0
P_6	0	0	0

Need

	R_1	R_2	R_3
P_1	0	1	0
P_2	0	0	1
P_3	1	0	0
P_4	0	1	0
P_5	0	0	1
P_6	0	0	1

Available

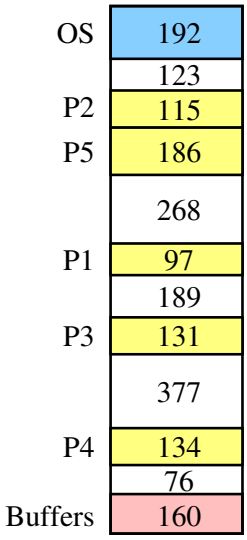
R_1	R_2	R_3
0	0	0

We start with $Finish[i] = False$ for $1 \leqslant i \leqslant 6$.

For no i, we have $Need_i \leqslant Allocation$, so the deadlock-detection algorithm terminates. Since $Finish[i] = False$ for many processes, the system is in a deadlock state.

3. [Physical memory]

(a) Prof. Barivas's laptop computer uses an old version of FooOS. The computer has a total memory of 2 GB. Out of this, 192 MB is reserved for OS codes and kernel data structures, and 160 MB is reserved for storing run-time data (like buffers) of user processes. The remaining part of the memory can be allocated to user processes. The old version of FooOS does contiguous memory allocation. At some point of time, there are five running processes P1, P2, P3, P4, and P5. Their memory requirements and the holes are shown in the figure below.



At this point of time, the following events happen in the given sequence.

A new process P6 of memory requirement 180 MB arrives.
A new process P7 of memory requirement 70 MB arrives.
A new process P8 of memory requirement 210 MB arrives.
The process P2 leaves.

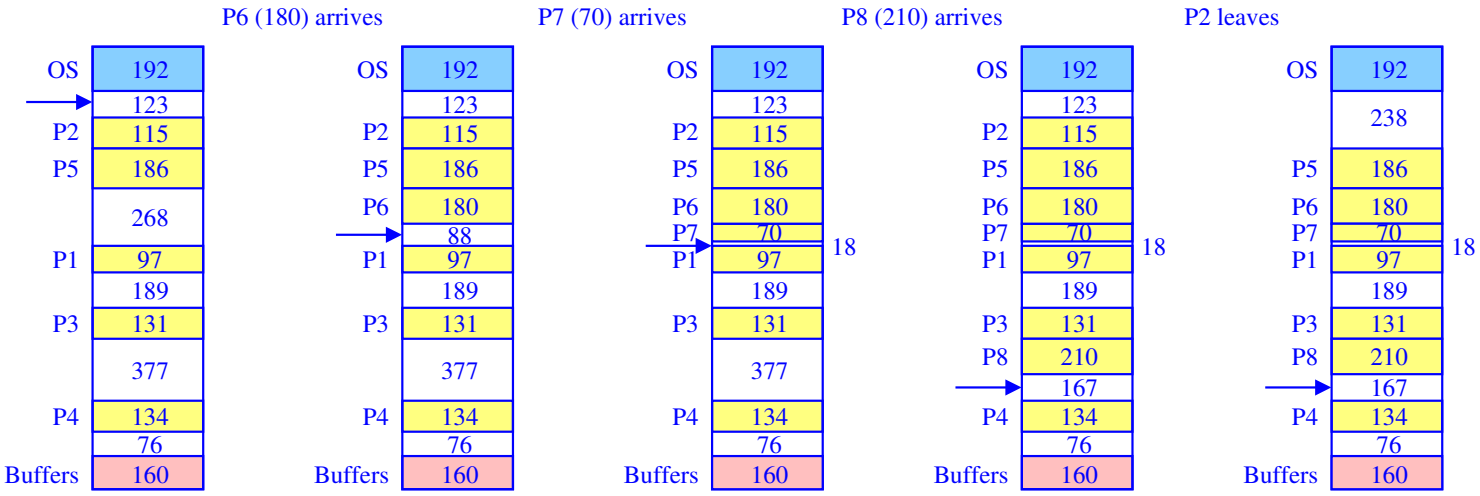
No memory compaction (that is, defragmentation) is done by the OS. Instead, a process which does not fit in any of the existing holes waits until some running process(es) terminate(s) freeing enough memory to create a hole big enough to accommodate the newly arrived process.

Clearly mention, for each of the following three contiguous memory-allocation strategies, how the above four events are handled by FooOS. After each event, you may redraw the picture or write the memory configuration as a sequence of items. The initial configuration shown in the adjacent picture can be written in text as: OS(192), Hole(123), P2(115), P5(186), Hole(268), P1(97), Hole(189), P3(131), Hole(377), P4(134), Hole(76), Buffers(160). Assume that a process is always allocated to the top side (the OS side) of an appropriate hole.

First-fit strategy

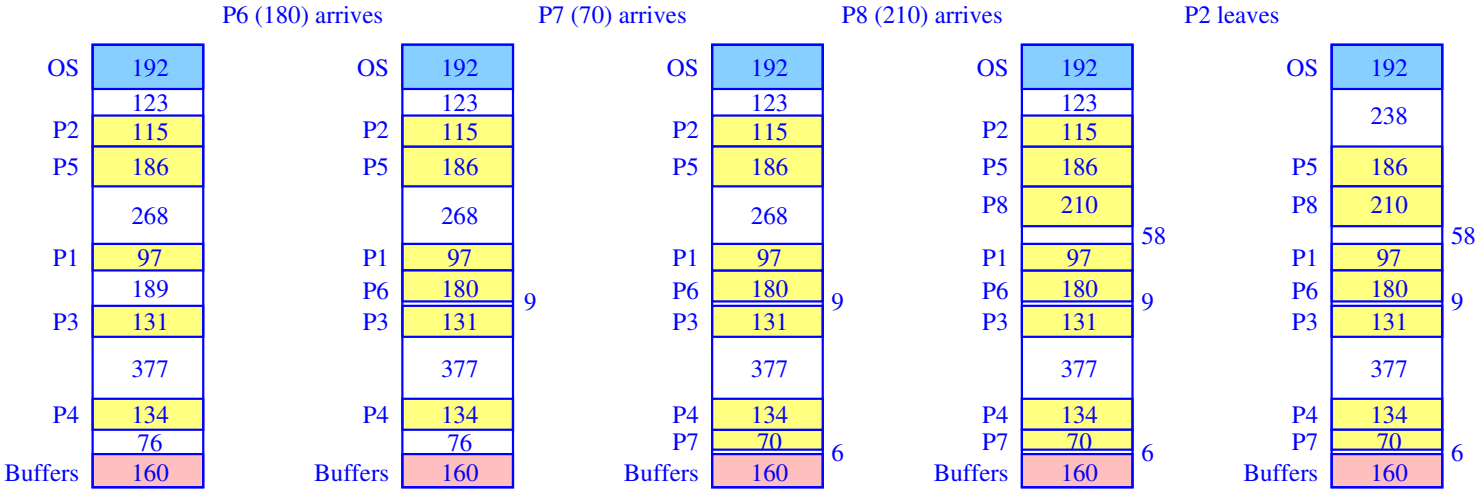
Assume that the first-fit pointer points to the 123 MB hole just before the above sequence of four events. **(2)**

Solution



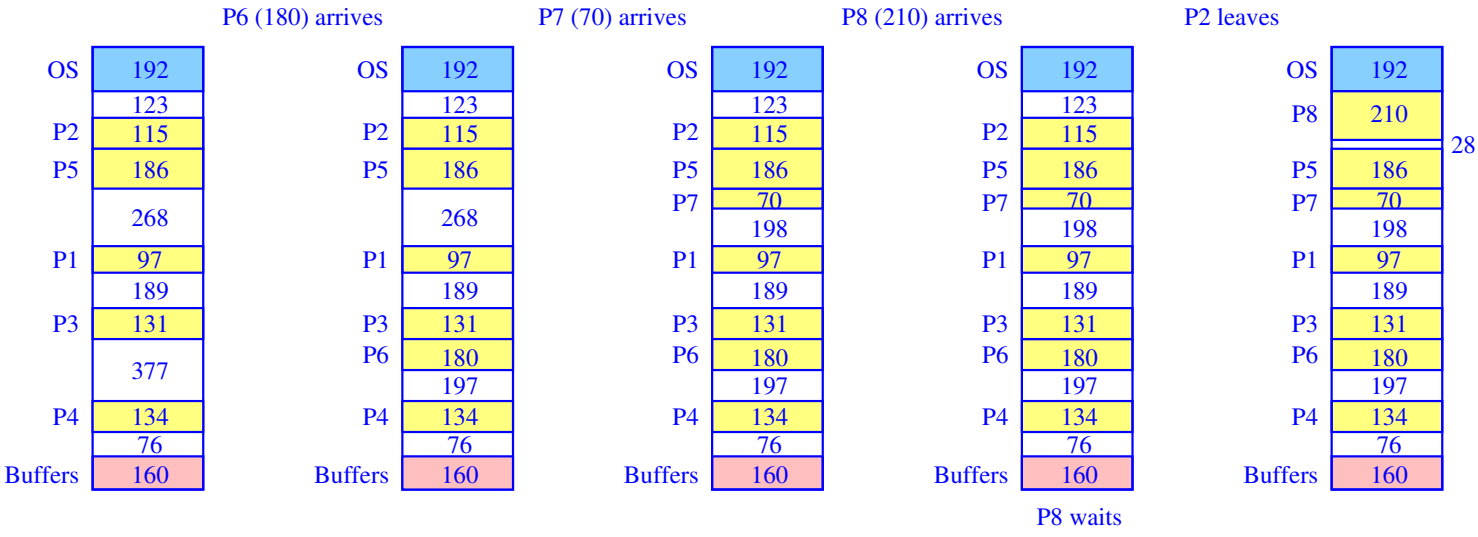
Best-fit strategy **(2)**

Solution



Worst-fit strategy **(2)**

Solution



(b) Prof. Barivas installs a new version of FooOS on his laptop computer (the memory remains the same: 2 GB). The new version of FooOS supports paging, where each page (or frame) is of size 8 KB. Each process is restricted to have a logical address space of maximum size 256 MB. As in Part (a), assume that the OS reserves 192 MB + 160 MB for its operation. In the context of this new FooOS version, answer the following questions. Assume that each page-table entry is of size 4 bytes. Show all your calculations.

How many frames can be allocated to user processes? **(1)**

Solution $\dfrac{(2048 - 192 - 160)\text{ MB}}{8\text{ KB}} = 212\text{ K} = 217,088$

What is the maximum size (in MB) of a process whose page table fits in a single frame? **(1)**

Solution Each frame can store $\dfrac{8\text{ KB}}{4\text{ B}} = 2\text{ K}$ page-table entries. Since each page is of size 8 KB, the maximum size of a process whose page table fits in a single frame is $2\text{ K} \times 8\text{ KB} = 16\text{ MB}$.

If hierarchical paging is used, what is the maximum number of frames needed to store the page table of a single process? Use the minimum possible number of levels in the hierarchy, given the 256 MB restriction on the logical address space of each process. **(2)**

Solution Two-level hierarchical paging suffices. By the above calculations, each frame can point to 16 MB of data. Therefore a file of size 256 MB requires $\dfrac{256}{16} = 16$ second-level page-table frames. The addresses of these 16 frames can be stored in a single first-level page-table frame. Therefore the maximum storage needed for storing the page table of a process is $1 + 16 = 17$.

4. [Virtual memory]

(a) The page-reference string of a process with seven pages (numbered 0–6) is

$$0,1,3,6,2,1,3,5,3,0,4,1,0,5,3,2,0.$$

The number of memory frames allocated to the process is four (initially empty). Let p_0, p_1, p_2, p_3 (integers in the range 0–6) be the pages of the process, currently in memory. The *second-chance page-replacement algorithm* is used, with one reference bit per page (all initialized to 0). The starting position of the pointer is at p_0. Immediately after a page replacement, the reference bit of the newly loaded page is set to 1. For each page reference in the above string, show the pages p_0, p_1, p_2, p_3 loaded to memory just after the reference, along with the reference-bit values for these pages. Present your answer as a 2-d table with 4 rows p_0, p_1, p_2, p_3 and 17 columns (one for each page in the reference string). Identify the page faults, and find the total number of page faults. **(3)**

Solution In the following table, the page numbers p_i are shown along with the reference bits in parentheses.

	0	1	3	6	2	1	3	5	3	0	4	1	0	5	3	2	0
p_0	0(1)	0(1)	0(1)	0(1)	2(1)	2(1)	2(1)	2(1)	2(1)	2(0)	4(1)	4(1)	4(1)	4(1)	4(0)	2(1)	2(1)
p_1		1(1)	1(1)	1(1)	1(0)	1(1)	1(1)	1(0)	1(0)	0(1)	0(1)	0(0)	0(1)	0(1)	0(0)	0(0)	0(1)
p_2			3(1)	3(1)	3(0)	3(0)	3(1)	3(0)	3(1)	3(1)	3(0)	1(1)	1(1)	1(1)	1(0)	1(0)	1(0)
p_3				6(1)	6(0)	6(0)	6(0)	5(1)	5(1)	5(1)	5(0)	5(0)	5(0)	5(1)	3(1)	3(1)	3(1)
	↑	↑	↑	↑	↑			↑		↑	↑	↑			↑	↑	

The pointer location where the last change was made is marked in red. The next page fault initiates a search from the circularly next position. The page faults are shown by vertical arrows. There are 11 of them.

(b) An OS implements virtual memory using the *LRU-approximation page-replacement algorithm* based on reference bits. A small process is allocated four frames. Initiated by timer interrupts, the reference bits are checked at regular intervals. Initially, the reference bits are 0111 (page 0 is 0, the rest are 1). At the four subsequent timer interrupts, the values are $1011, 1010, 1101, 0010$. If the page-replacement algorithm is used with 8-bit counters (all initialized to 0), show the contents (bit-wise) of the four counters after the last of the above timer interrupts. Assume that no page fault occurred during the above intervals. **(3)**

Solution

	Timer INT 0	Timer INT 1	Timer INT 2	Timer INT 3	Timer INT 4
Reference bits	0111	1011	1010	1101	0010
Page 0	00000000	10000000	11000000	11100000	01110000
Page 1	10000000	01000000	00100000	10010000	01001000
Page 2	10000000	11000000	11100000	01110000	10111000
Page 3	10000000	11000000	01100000	10110000	01011000

(c) A working-set model (with window size 4) is implemented to allocate frames to a process P. The page references of the process P from time $t = 1$ onward are

$$c, c, d, b, c, e, c, a, a, d.$$

The initial working set at time $t = 0$ is $\{e, d, a\}$, where page a was referenced at time $t = 0$, page d was referenced at time $t = -1$, and page e was referenced at time $t = -2$. In a table, show, for each page reference: (i) the time t, (ii) the working set at time t, (iii) whether there is a page fault, and (iv) the number of frames to be allocated to process P at time t. **(3)**

Solution

Page reference	Time	Working set	Hit/Miss
c	1	$\{e,d,a,c\}$	M
c	2	$\{d,a,c\}$	H
d	3	$\{a,c,d\}$	H
b	4	$\{c,d,b\}$	M
c	5	$\{d,b,c\}$	H
e	6	$\{d,b,c,e\}$	M
c	7	$\{b,e,c\}$	H
a	8	$\{e,c,a\}$	M
a	9	$\{e,c,a\}$	H
d	10	$\{c,a,d\}$	M

The number of pages allocated at any time equals the size of the working set at that time.

(d) A memory-management module implements a local page-replacement policy. Prof. Foo claims that unlike the global page-replacement policy, here the execution time of a process P is determined only by the execution sequence and the behavior of the process P, and no other process in any way can affect the execution time of the process P. Prove/Disprove (with proper justification) the claim of Prof. Foo. **(2)**

Solution The claim is false. For example, if one or more processes thrash, a non-thrashing process needs to wait more than the usual time in the queue for doing disk I/O.

(e) Prove/disprove with proper justification: *The optimal page-replacement algorithm may suffer from Belady's anomaly.* **(2)**

Solution False. See book for justification.

5. [Storage management]

A lecture by Prof. Sad is stored in the file *combinatorics.mkv*. The size of this file is 567,890,123 bytes. The HDD that stores this file has capacity 1 TB, and uses 1 KB blocks. In connection with the storage of Prof. Sad's video in the HDD, answer the following parts. In each part, show your calculations.

(a) What is the smallest number of bits (should be a multiple of 8), that can be used to address the blocks of the HDD? **(1)**

Solution The number of blocks in the HDD is $\frac{1\text{ TB}}{1\text{ KB}} = 1\text{ G} = 2^{30}$. Therefore, 32 bits (that is, 4 bytes) are needed to address the blocks.

(b) How many data blocks are needed in the HDD to store the content of the file? **(1)**

Solution $\left\lceil \frac{567,890,123}{1,024} \right\rceil = 554,581$ data blocks are needed.

(c) Suppose that the blocks are addressed as derived in Part (a). If a linked allocation of storage is used with the next-block links stored in the blocks themselves, how many blocks are needed to store *combinatorics.mkv*? **(1)**

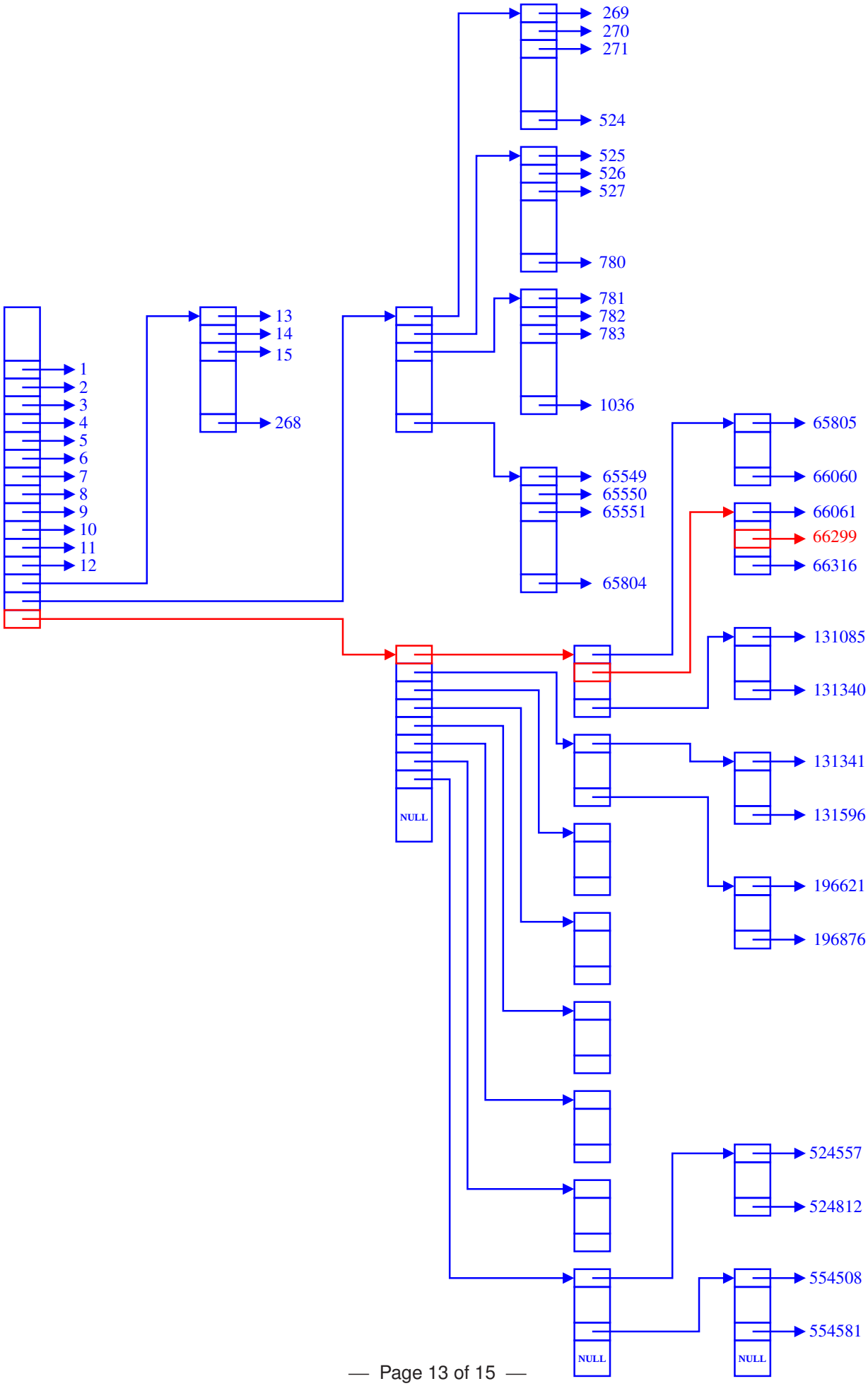
Solution The amount of data stored in each block is $1024 - 4 = 1020$ bytes. Therefore Prof. Sad's lecture video requires $\left\lceil \frac{567,890,123}{1020} \right\rceil = 556,756$ blocks.

(d) Explain how the file is stored if the HDD uses MSDOS's FAT file system. Clearly mention the index blocks and their linking in the file allocation table. **(2)**

Solution The FAT format does not store the links in the blocks, so 554,581 data blocks are needed. Let the addresses of these blocks be $b_1, b_2, \ldots, b_{554581}$. Then, the global file-allocation table T stores $T[b_i] = b_{i+1}$ for $i = 1, 2, \ldots, 554580$ and $b_{554581} = \text{NULL}$.

(e) In this part, assume that the HDD uses the Unix inode format with a total of 15 pointers in the top-level index block. Clearly mention the index blocks (along with their links to index or data blocks) that are used for locating the data blocks of the lecture video. Explain how many disk-block accesses are required to read the 67,890,123-th byte of the file. **(4 + 1)**

Solution The detailed organization of the $554,581$ data blocks and the index blocks at various levels is given in the following figure. The $67,890,123$-th byte is in the block $1 + \lfloor 67890123/1024 \rfloor = 66299$. The block storing this byte can be located along the path marked red. Five disk accesses are required to retrieve this byte.



6. [Mass-storage structure]

(a) Consider a disk where each movement of the arm to the adjacent cylinder takes a seek time of 6 ms, the rotation speed is 3600 rpm, and each track holds 1 MB of data. Each disk block is of size 4 KB. The SCAN algorithm is used for disk scheduling. A file of size 32 KB is stored on the disk. The i-th block of the file (i starts from 0) is stored on cylinder $(30 \times i)\ \%\ n$, where $n = 100$ is the total number of cylinders (cylinder numbers: $0-99$). Compute the total time required to read the complete file. Assume that the disk head is initially on cylinder 25 moving up (that is, towards larger cylinder numbers). Assume also that after the correct cylinder is located, there is a rotational latency of half of a single revolution time, before the transfer of the data block can start from that cylinder. **(4)**

Solution Seek time / track = 6 ms

Block size = 4 KB, file size = 32 KB, total 8 blocks

Blocks for storing the file are $0, 30, 60, 90, 20, 50, 80, 10$

Seek movements for SCAN: 155 $(25 \rightarrow 90 \rightarrow 0)$

Total seek time = $155 \times 6 = 930$ ms

One complete revolution time = $60000/3600 \approx 16.67$ ms.

One half revolution time = $(60000/3600)/2 \approx 8.33$ ms.

Total rotational latency = $8 \times 8.33 = 66.64$ ms.

Data transfer time for one block = $((16.67 \text{ ms}/1 \text{ MB}) \times 4$ KB.

Data transfer time for the entire file = $((16.67 \text{ ms}/1 \text{ MB}) \times 32 \text{ KB} = 0.52$ ms.

Total time to read file = $930 + 66.64 + 0.52 = 997.16$ ms.

(b) A hard disk with 16 recording surfaces (0–15) has 16384 cylinders (0–16383), and each cylinder contains 64 sectors (0–63). The data storage capacity in each sector is 512 bytes. Each sector is addressed by a triple $\langle S,C,T\rangle$, where S,C,T stand for the surface number, the cylinder (or track) number, and the sector number, respectively. For a sector $\langle S,C,T\rangle$ (not the last one), the next sector is defined to be

$$\text{nextof}\langle S,C,T\rangle = \begin{cases} \langle S,C,T+1\rangle & \text{if } T < 63 \\ \langle S,C+1,0\rangle & \text{if } T = 63 \text{ and } C < 16383 \\ \langle S+1,0,0\rangle & \text{if } T = 63 \text{ and } C = 16383 \end{cases}$$

A file of size 243987 KB is stored in the disk, and the starting disk location of the file is $\langle 9,12000,30\rangle$. What is the address of the last sector of the file if the file is stored in a contiguous manner? **(3)**

Solution In the mixed-radix system, $\langle S,C,T\rangle$ stands for the sector numbered $(16384\times 64)\times S+64\times C+T$. The file of size 243987 KB needs $2\times 243987 = 487974 = 64\times 7624+38$ sectors. Thus, the last sector has the number

$$\begin{aligned} & \Big((16384\times 64)\times 9+64\times 12000+30\Big)+\Big(64\times 7624+38\Big)-1 \\ = \;& (16384\times 64)\times 9+64\times(12000+7624)+(30+38-1) \\ = \;& (16384\times 64)\times 9+64\times(12000+7624)+67 \\ = \;& (16384\times 64)\times 9+64\times(12000+7624)+64+3 \\ = \;& (16384\times 64)\times 9+64\times(12000+7624+1)+3 \\ = \;& (16384\times 64)\times 9+64\times 19625+3 \\ = \;& (16384\times 64)\times 9+64\times(16384+3241)+3 \\ = \;& (16384\times 64)\times 10+64\times 3241+3. \end{aligned}$$

Therefore, the last sector of the file has address $\langle 10,3241,3\rangle$.