Nov 13, 2024

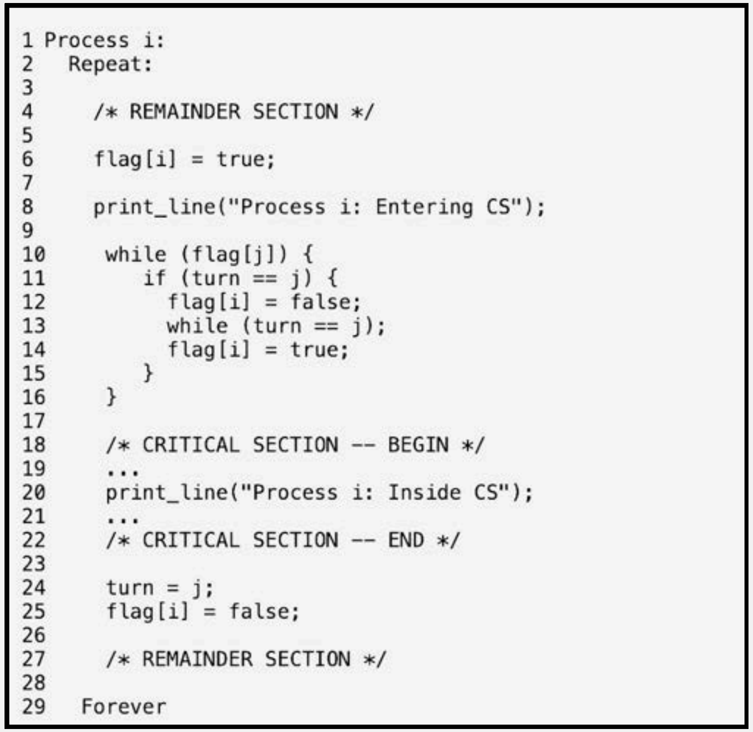
CS 350 DISCUSSION 10

Mutual Exclusion and CFQ

Let’s Practice

Exercise 1

Consider the code for 2-party mutual exclusion between two processes, namely Process i and Process j, reported below. Here, mutual exclusion is performed so that only one process at a time can issue a print_line() command. The code of Process j is identical, except for all the occurrences of “i” having been replaced with “j” and vice-versa. The flag array is shared among all the processes, and so is the variable turn. Always assume an in-order CPU. The turn variable is initialized to i at the start.



```
Process i:
  Repeat:

    /* REMAINDER SECTION */

    flag[i] = true;

    print_line("Process i: Entering CS");

     while (flag[j]) {
        if (turn == j) {
          flag[i] = false;
          while (turn == j);
          flag[i] = true;
        }
     }

     /* CRITICAL SECTION -- BEGIN */
     ...
     print_line("Process i: Inside CS");
     ...
     /* CRITICAL SECTION -- END */

     turn = j;
     flag[i] = false;

     /* REMAINDER SECTION */

   Forever
```

Problem 19.1 from the book

Question 1

What is the value to which we should initialize the items in the flag array for the algorithm to guarantee 2-party mutual exclusion?

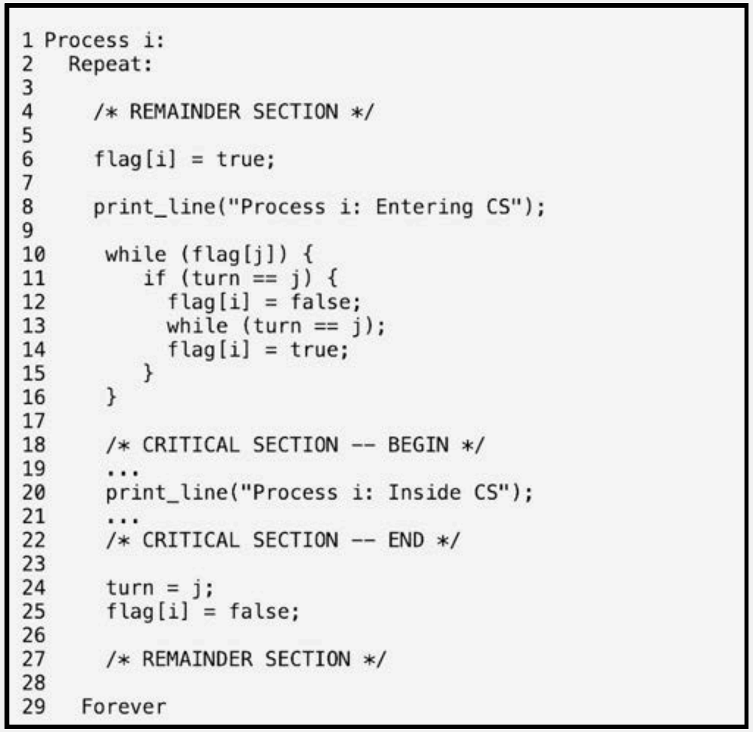
flag[i] = false , flag[j] = false

turn = j ;

Let’s Practice

Exercise 1

Consider the code for 2-party mutual exclusion between two processes, namely Process i and Process j, reported below. Here, mutual exclusion is performed so that only one process at a time can issue a print_line() command. The code of Process j is identical, except for all the occurrences of “i” having been replaced with “j” and vice-versa. The flag array is shared among all the processes, and so is the variable turn. Always assume an in-order CPU. The turn variable is initialized to i at the start.



```
Process i:
  Repeat:

    /* REMAINDER SECTION */

    flag[i] = true;

    print_line("Process i: Entering CS");

     while (flag[j]) {
        if (turn == j) {
          flag[i] = false;
          while (turn == j);
          flag[i] = true;
        }
     }

     /* CRITICAL SECTION -- BEGIN */
     ...
     print_line("Process i: Inside CS");
     ...
     /* CRITICAL SECTION -- END */

     turn = j;
     flag[i] = false;

     /* REMAINDER SECTION */

   Forever
```

Problem 19.1 from the book

Question 1

What is the value to which we should initialize the items in the flag array for the algorithm to guarantee 2-party mutual exclusion?

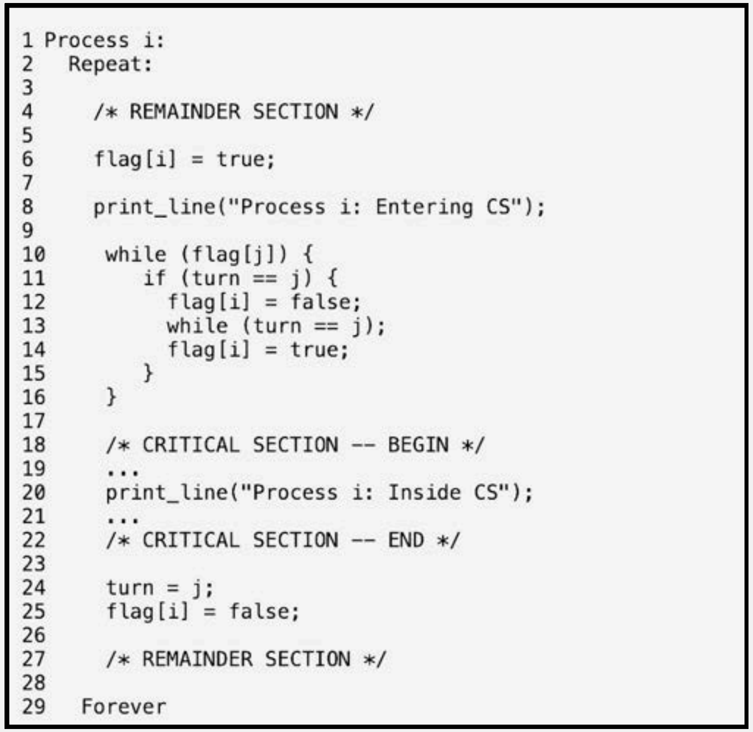
Answer

All items in the flag array must be initialized to False.

Let’s Practice

Exercise 1

Consider the code for 2-party mutual exclusion between two processes, namely Process i and Process j, reported below. Here, mutual exclusion is performed so that only one process at a time can issue a print_line() command. The code of Process j is identical, except for all the occurrences of “i” having been replaced with “j” and vice-versa. The flag array is shared among all the processes, and so is the variable turn. Always assume an in-order CPU.



```
Process i:
  Repeat:

    /* REMAINDER SECTION */

    flag[i] = true;

    print_line("Process i: Entering CS");

     while (flag[j]) {
        if (turn == j) {
          flag[i] = false;
          while (turn == j);
          flag[i] = true;
        }
     }

     /* CRITICAL SECTION -- BEGIN */
     ...
     print_line("Process i: Inside CS");
     ...
     /* CRITICAL SECTION -- END */

     turn = j;
     flag[i] = false;

     /* REMAINDER SECTION */

   Forever
```

Problem 19.1 from the book

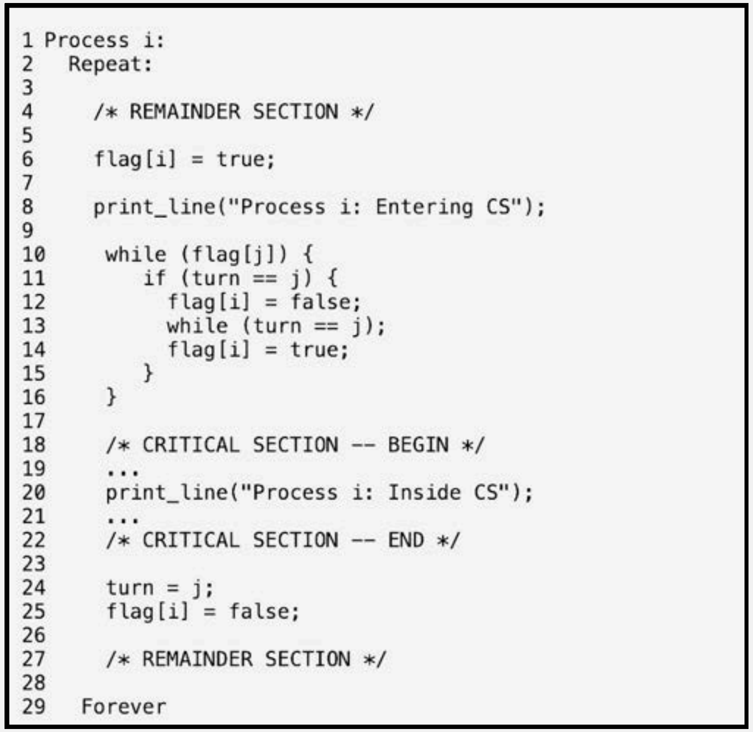
Question 2

Explain the implications of initializing turn to j if only Process i will attempt to acquire the resource?

Let’s Practice

Exercise 1

Consider the code for 2-party mutual exclusion between two processes, namely Process i and Process j, reported below. Here, mutual exclusion is performed so that only one process at a time can issue a print_line() command. The code of Process j is identical, except for all the occurrences of “i” having been replaced with “j” and vice-versa. The flag array is shared among all the processes, and so is the variable turn. Always assume an in-order CPU.



```
Process i:
  Repeat:

    /* REMAINDER SECTION */

    flag[i] = true;

    print_line("Process i: Entering CS");

     while (flag[j]) {
        if (turn == j) {
          flag[i] = false;
          while (turn == j);
          flag[i] = true;
        }
     }

     /* CRITICAL SECTION -- BEGIN */
     ...
     print_line("Process i: Inside CS");
     ...
     /* CRITICAL SECTION -- END */

     turn = j;
     flag[i] = false;

     /* REMAINDER SECTION */

   Forever
```

Problem 19.1 from the book

Question 2

Explain the implications of initializing turn to j if only Process i will attempt to acquire the resource?

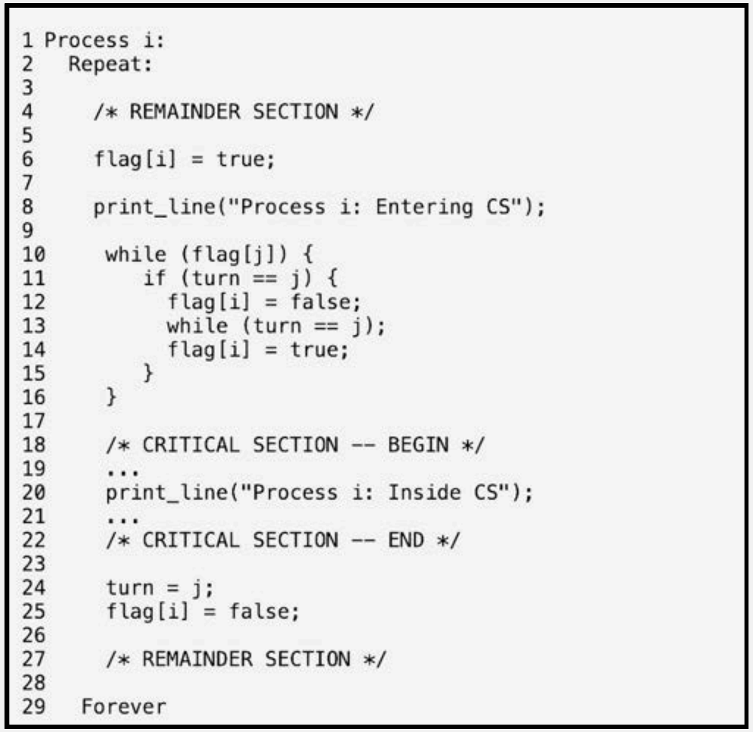
Answer

It has no impact. Process i still can access the critical section.

Let’s Practice

Exercise 1

Consider the code for 2-party mutual exclusion between two processes, namely Process i and Process j, reported below. Here, mutual exclusion is performed so that only one process at a time can issue a print_line() command. The code of Process j is identical, except for all the occurrences of “i” having been replaced with “j” and vice-versa. The flag array is shared among all the processes, and so is the variable turn. Always assume an in-order CPU.



```
1 Process i:
2   Repeat:
3
4     /* REMAINDER SECTION */
5
6     flag[i] = true;
7
8     print_line("Process i: Entering CS");
9
10     while (flag[j]) {
11        if (turn == j) {
12          flag[i] = false;
13          while (turn == j);
14          flag[i] = true;
15        }
16     }
17
18     /* CRITICAL SECTION -- BEGIN */
19     ...
20     print_line("Process i: Inside CS");
21     ...
22     /* CRITICAL SECTION -- END */
23
24     turn = j;
25     flag[i] = false;
26
27     /* REMAINDER SECTION */
28
29   Forever
```

Problem 19.1 from the book

Question 3

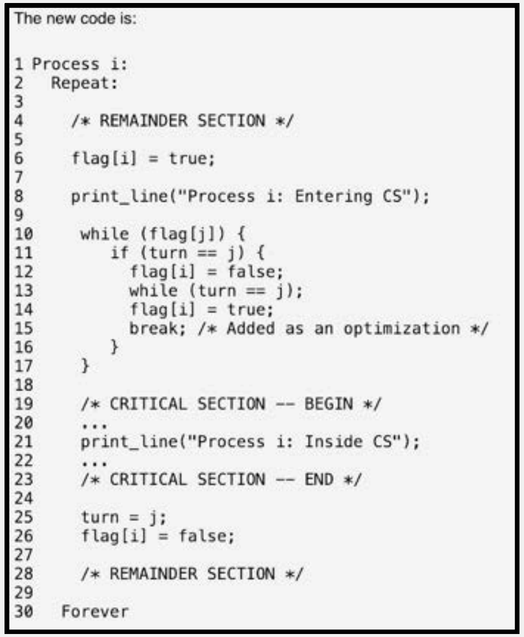
Your colleague has modified this code to perform an optimization. They have inserted a break command between line 14 and line 15. Is this going to be a problem? If you believe that there is a problem, produce a sample of execution interleaving that illustrates the problem.

Let’s Practice

Exercise 1

Question 3

Your colleague has modified this code to perform an optimization. They have inserted a break command between line 14 and line 15. Is this going to be a problem? If you believe that there is a problem, produce a sample of execution interleaving that illustrates the problem.



The new code is:

```
1 Process i:
2   Repeat:
3
4     /* REMAINDER SECTION */
5
6     flag[i] = true;
7
8     print_line("Process i: Entering CS");
9
10     while (flag[j]) {
11        if (turn == j) {
12          flag[i] = false;
13          while (turn == j);
14          flag[i] = true;
15          break; /* Added as an optimization */
16        }
17     }
18
19     /* CRITICAL SECTION -- BEGIN */
20     ...
21     print_line("Process i: Inside CS");
22     ...
23     /* CRITICAL SECTION -- END */
24
25     turn = j;
26     flag[i] = false;
27
28     /* REMAINDER SECTION */
29
30   Forever
```

Yes ,

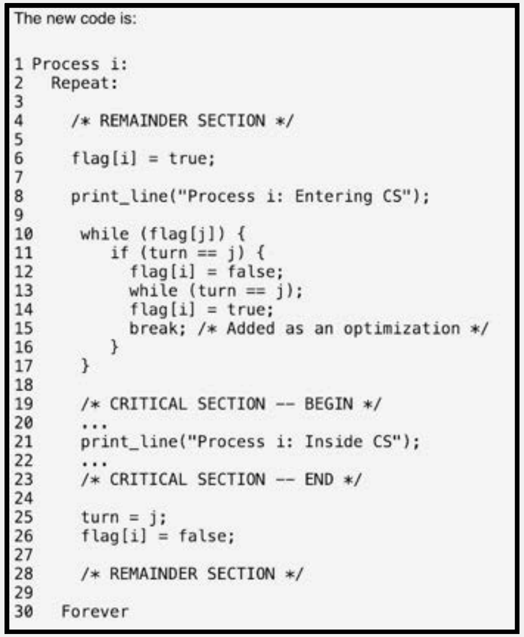
Problem 19.1 from the book

Let’s Practice

Exercise 1

Question 3

Your colleague has modified this code to perform an optimization. They have inserted a break command between line 14 and line 15. Is this going to be a problem? If you believe that there is a problem, produce a sample of execution interleaving that illustrates the problem.

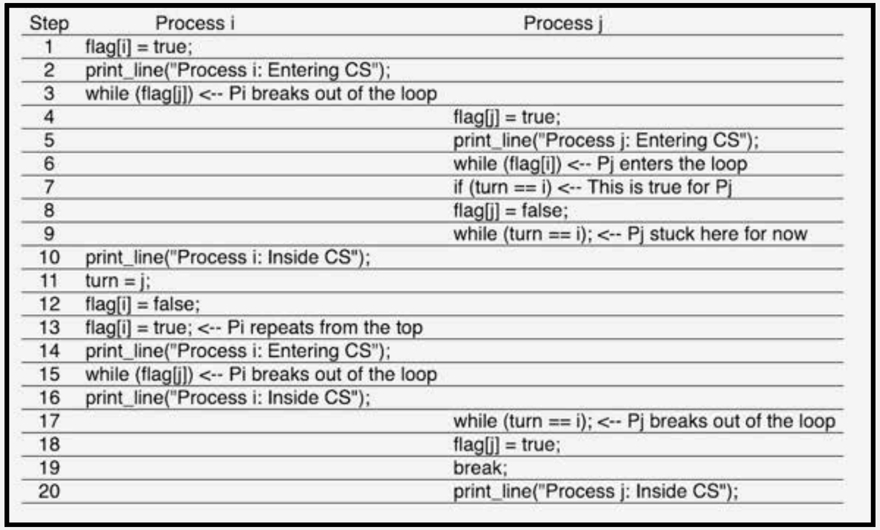


The new code is:

```
1 Process i:
2   Repeat:
3
4     /* REMAINDER SECTION */
5
6     flag[i] = true;
7
8     print_line("Process i: Entering CS");
9
10     while (flag[j]) {
11        if (turn == j) {
12          flag[i] = false;
13          while (turn == j);
14          flag[i] = true;
15          break; /* Added as an optimization */
16        }
17     }
18
19     /* CRITICAL SECTION -- BEGIN */
20     ...
21     print_line("Process i: Inside CS");
22     ...
23     /* CRITICAL SECTION -- END */
24
25     turn = j;
26     flag[i] = false;
27
28     /* REMAINDER SECTION */
29
30   Forever
```

Answer

Yes, this is going to be a problem. It will result in both processes being in a critical section at the same time as shown below.



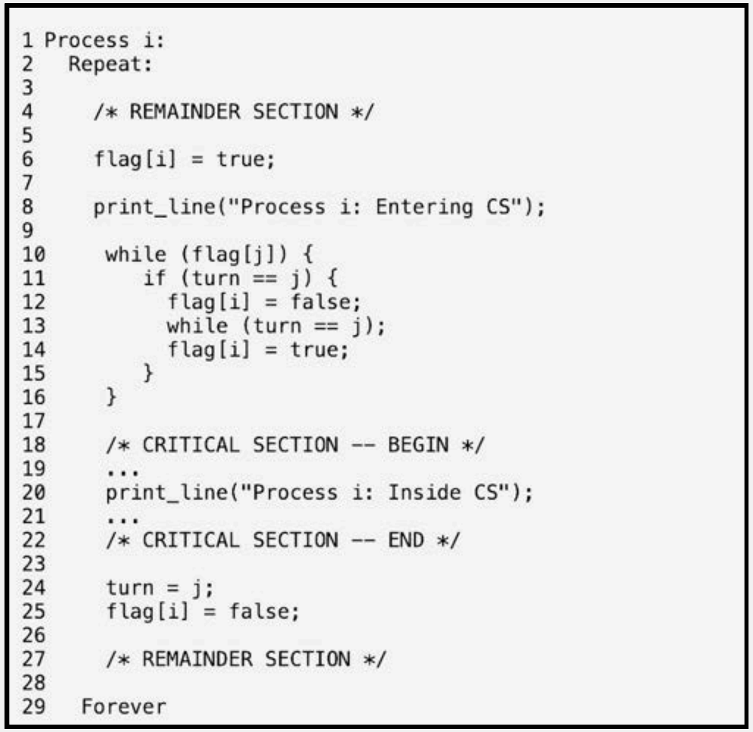
Step	Process i	Process j
1	flag[i] = true;	
2	print_line("Process i: Entering CS");	
3	while (flag[j]) <-- Pi breaks out of the loop	
4		flag[j] = true;
5		print_line("Process j: Entering CS");
6		while (flag[i]) <-- Pj enters the loop
7		if (turn == i) <-- This is true for Pj
8		flag[j] = false;
9		while (turn == i); <-- Pj stuck here for now
10	print_line("Process i: Inside CS");	
11	turn = j;	
12	flag[i] = false;	
13	flag[i] = true; <-- Pi repeats from the top	
14	print_line("Process i: Entering CS");	
15	while (flag[j]) <-- Pi breaks out of the loop	
16	print_line("Process i: Inside CS");	
17		while (turn == i); <-- Pj breaks out of the loop
18		flag[j] = true;
19		break;
20		print_line("Process j: Inside CS");

Problem 19.1 from the book

Let’s Practice

Exercise 1

Consider the code for 2-party mutual exclusion between two processes, namely Process i and Process j, reported below. Here, mutual exclusion is performed so that only one process at a time can issue a print_line() command. The code of Process j is identical, except for all the occurrences of “i” having been replaced with “j” and vice-versa. The flag array is shared among all the processes, and so is the variable turn. Always assume an in-order CPU.



```
Process i:
  Repeat:

    /* REMAINDER SECTION */

    flag[i] = true;

    print_line("Process i: Entering CS");

     while (flag[j]) {
        if (turn == j) {
          flag[i] = false;
          while (turn == j);
          flag[i] = true;
        }
     }

     /* CRITICAL SECTION -- BEGIN */
     ...
     print_line("Process i: Inside CS");
     ...
     /* CRITICAL SECTION -- END */

     turn = j;
     flag[i] = false;

     /* REMAINDER SECTION */

   Forever
```

Problem 19.1 from the book

Question 4

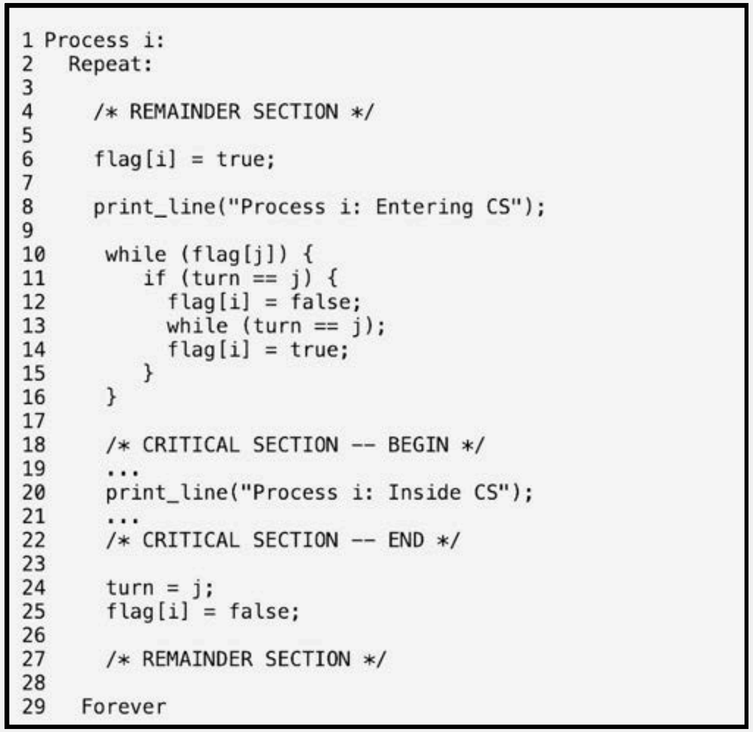
Change the code to leverage semaphores instead of the software-based approach currently employed in the code. Specify how you are going to initialize any semaphore you decide to use.

Let’s Practice

Exercise 1

Question 4

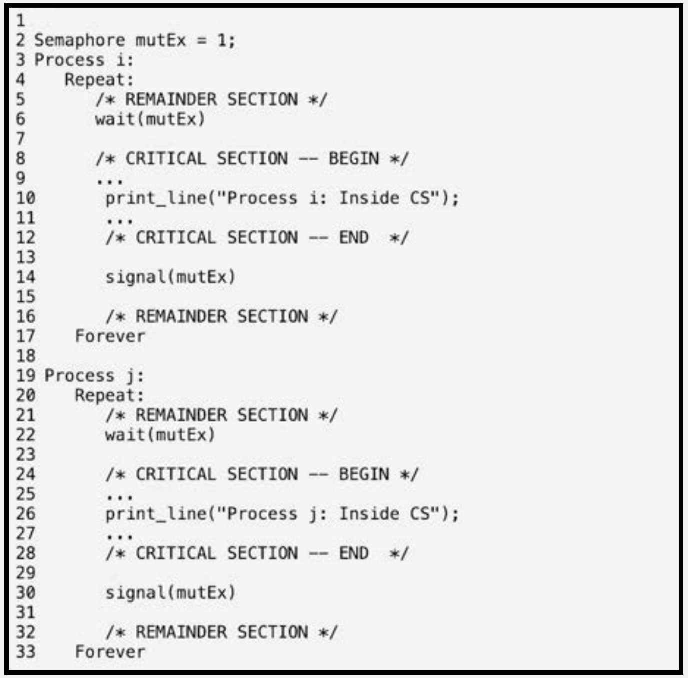
Change the code to leverage semaphores instead of the software-based approach currently employed in the code. Specify how you are going to initialize any semaphore you decide to use.



```
Process i:
  Repeat:

    /* REMAINDER SECTION */

    flag[i] = true;

    print_line("Process i: Entering CS");

     while (flag[j]) {
        if (turn == j) {
          flag[i] = false;
          while (turn == j);
          flag[i] = true;
        }
     }

     /* CRITICAL SECTION -- BEGIN */
     ...
     print_line("Process i: Inside CS");
     ...
     /* CRITICAL SECTION -- END */

     turn = j;
     flag[i] = false;

     /* REMAINDER SECTION */

   Forever
```

Problem 19.1 from the book

Answer



```

Semaphore mutEx = 1;
Process i:
   Repeat:
      /* REMAINDER SECTION */
      wait(mutEx)

      /* CRITICAL SECTION -- BEGIN */
      ...
       print_line("Process i: Inside CS");
       ...
       /* CRITICAL SECTION -- END  */

       signal(mutEx)

       /* REMAINDER SECTION */
    Forever

Process j:
    Repeat:
       /* REMAINDER SECTION */
       wait(mutEx)

       /* CRITICAL SECTION -- BEGIN */
       ...
       print_line("Process j: Inside CS");
       ...
       /* CRITICAL SECTION -- END  */

       signal(mutEx)

       /* REMAINDER SECTION */
    Forever
```

Let's Practice

Exercise 2

You have 3 applications running on your machine that have their own queue each. Each queue has a quantum of 2 requests. The applications that are running are 1) Call of Duty, 2) Spotify and 3) Zoom in order of their priorities of meeting their deadline. Table below has its requests and the position in the disk along with it. Your laptop happens to be using Linux and you, as a CS350 student wants to find out how requests are handled in your linux system. What scheduling do you go with?

$$q_{CFQ} = 2$$

	Disk head for Request 1	Disk head for Request 2	Disk head for Request 3	Disk head for Request 4	Disk head for Request 5	Disk head for Request 6
Call of Duty	3	4	12	2	5	2
Spotify	5	1	7	8	5	4
Zoom	9	2	5	9	2	1

Use the table to put the request into each queue and then to the single dispatch queue. The total number of disk positions in the circular disk are 12

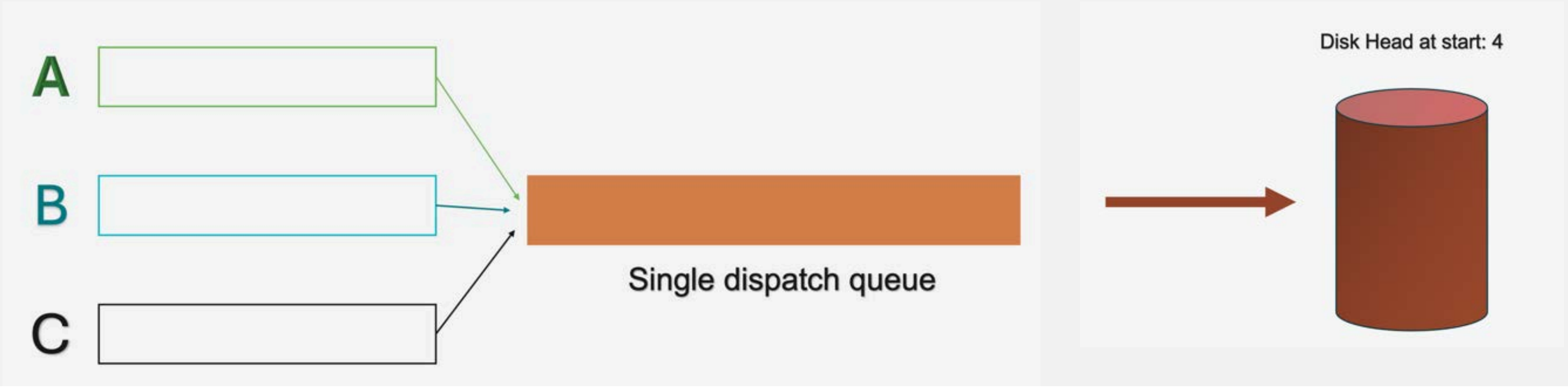
	Disk head for Request 1	Disk head for Request 2	Disk head for Request 3	Disk head for Request 4	Disk head for Request 5	Disk head for Request 6
Call of Duty	3	4	12	2	5	2
Spotify	5	1	7	8	5	4
Zoom	9	2	5	9	2	1

A) Call of Duty

B) Spotify

C) Zoom

$$q_{CFQ} = 2$$

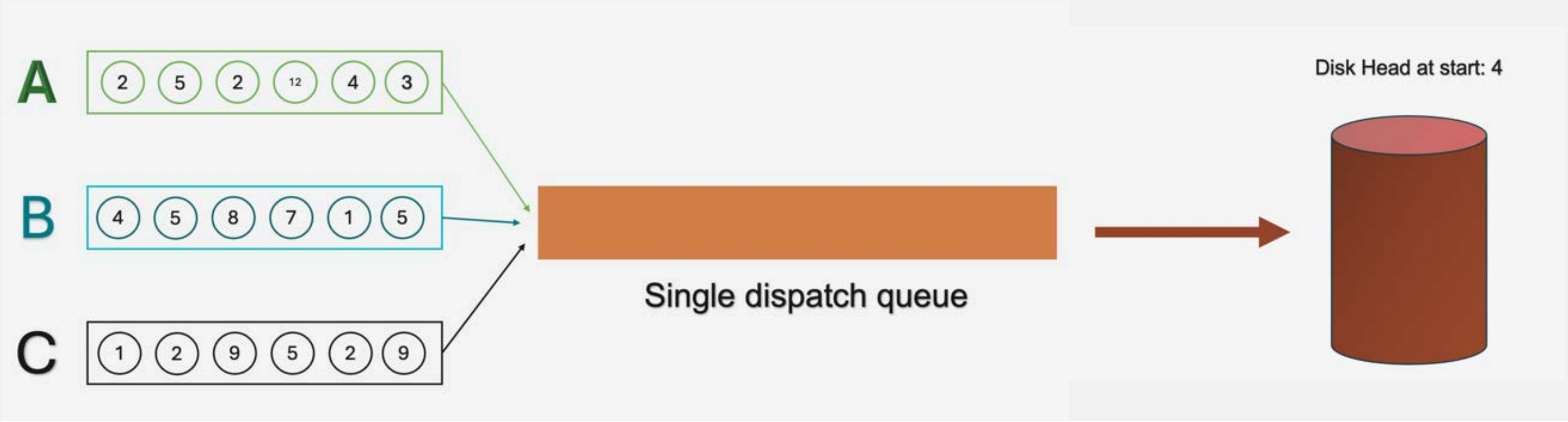


Use the table to put the request into each queue and then to the single dispatch queue.

A) Call of Duty
B) Spotify
C) Zoom

$$q_{CFQ} = 2$$

	Disk head for Request 1	Disk head for Request 2	Disk head for Request 3	Disk head for Request 4	Disk head for Request 5	Disk head for Request 6
Call of Duty	3	4	12	2	5	2
Spotify	5	1	7	8	5	4
Zoom	9	2	5	9	2	1



Use the table to put the request into each queue and then to the single dispatch queue.

A) Call of Duty

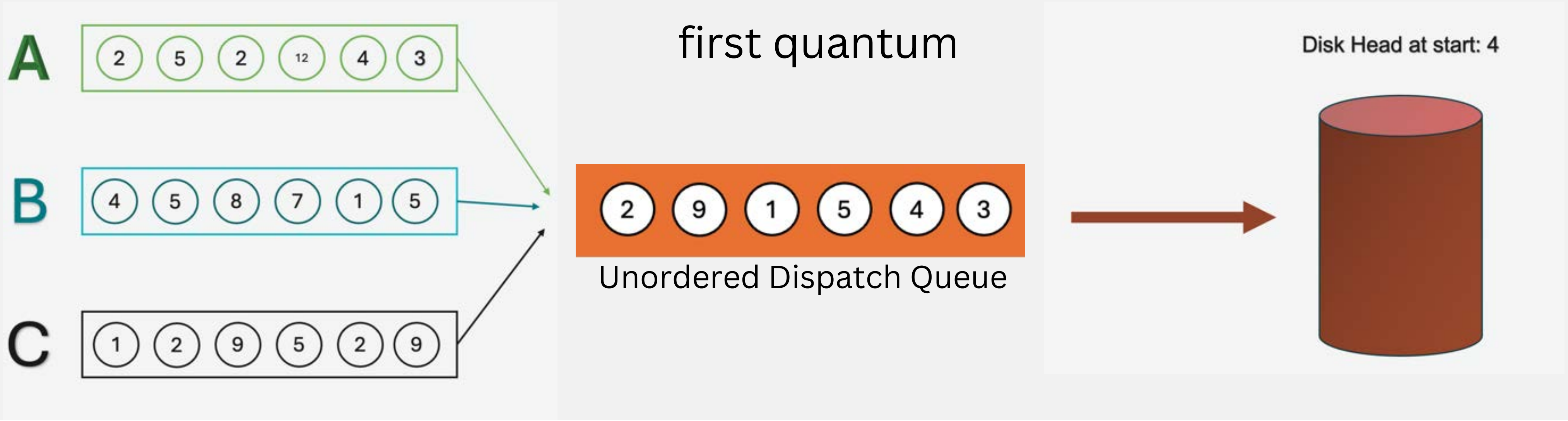
B) Spotify

C) Zoom

$$q_{CFQ} = 2$$

	Disk head for Request 1	Disk head for Request 2	Disk head for Request 3	Disk head for Request 4	Disk head for Request 5	Disk head for Request 6
Call of Duty	3	4	12	2	5	2
Spotify	5	1	7	8	5	4
Zoom	9	2	5	9	2	1

first quantum



Use the table to put the request into each queue and then to the single dispatch queue.

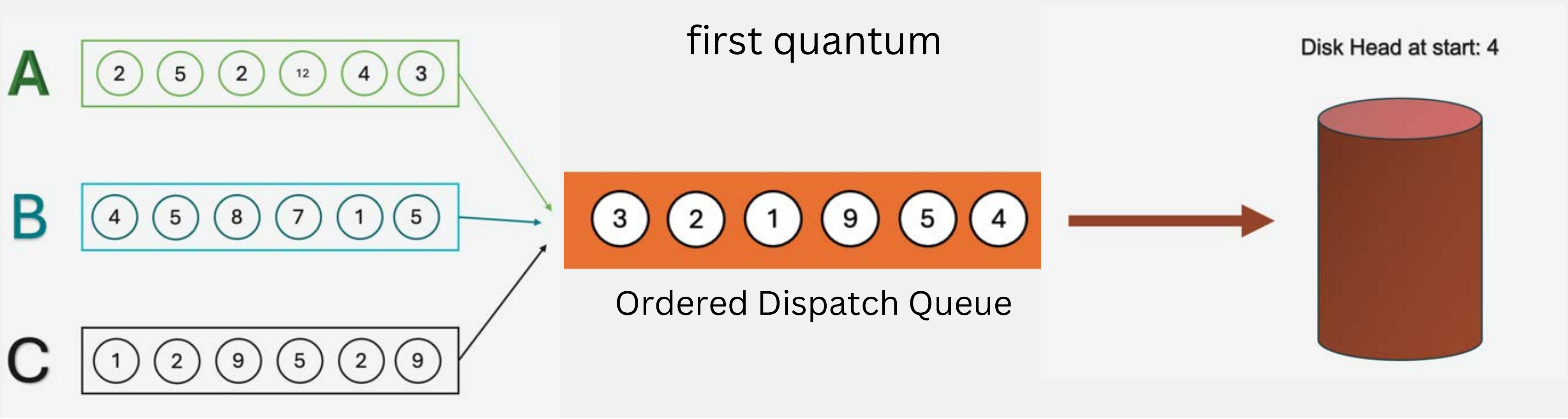
A) Call of Duty

B) Spotify

C) Zoom

$$q_{CFQ} = 2$$

	Disk head for Request 1	Disk head for Request 2	Disk head for Request 3	Disk head for Request 4	Disk head for Request 5	Disk head for Request 6
Call of Duty	3	4	12	2	5	2
Spotify	5	1	7	8	5	4
Zoom	9	2	5	9	2	1



Use the table to put the request into each queue and then to the single dispatch queue.

A) Call of Duty

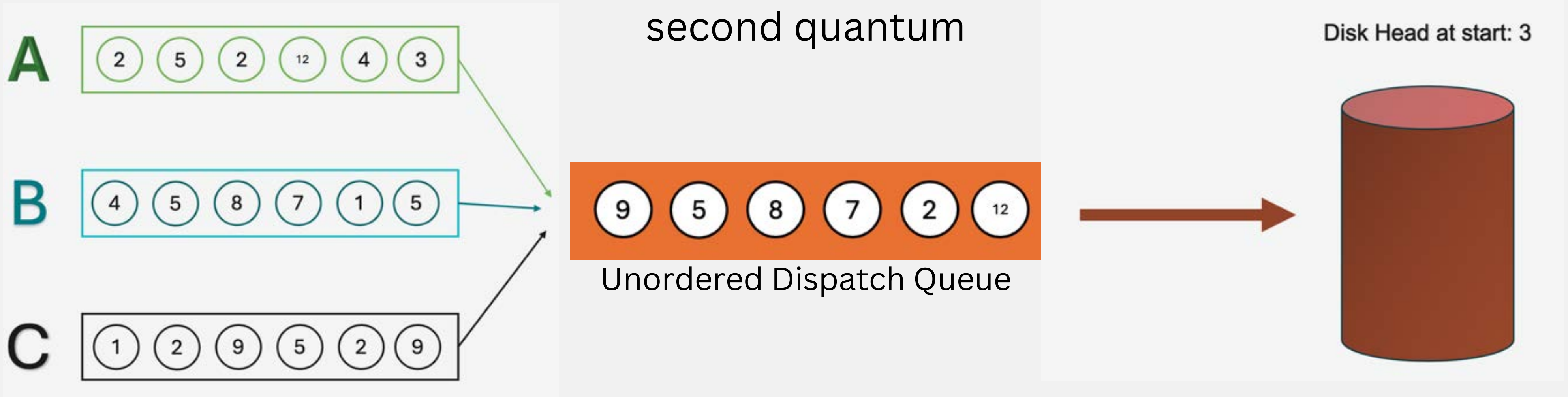
B) Spotify

C) Zoom

$$q_{CFQ} = 2$$

	Disk head for Request 1	Disk head for Request 2	Disk head for Request 3	Disk head for Request 4	Disk head for Request 5	Disk head for Request 6
Call of Duty	3	4	12	2	5	2
Spotify	5	1	7	8	5	4
Zoom	9	2	5	9	2	1

second quantum



Use the table to put the request into each queue and then to the single dispatch queue.

A) Call of Duty

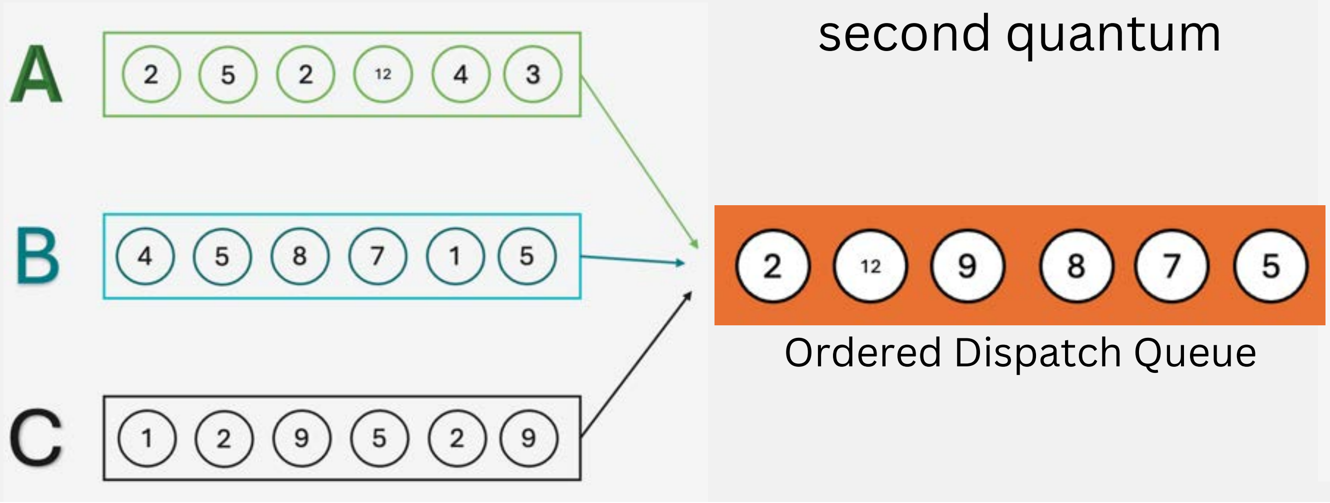
B) Spotify

C) Zoom

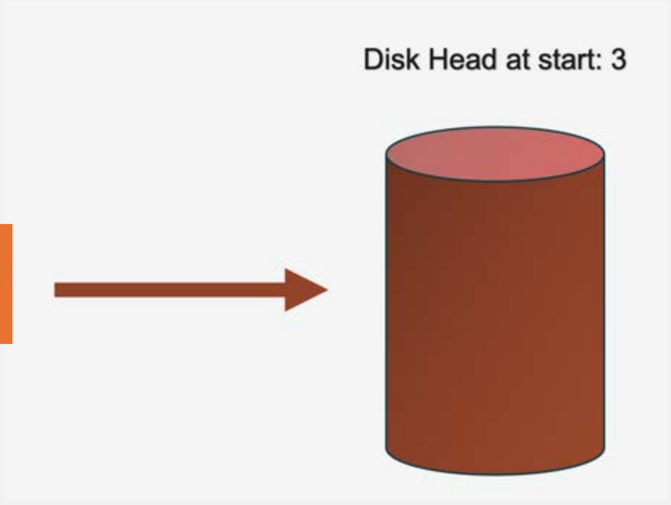
	Disk head for Request 1	Disk head for Request 2	Disk head for Request 3	Disk head for Request 4	Disk head for Request 5	Disk head for Request 6
Call of Duty	3	4	12	2	5	2
Spotify	5	1	7	8	5	4
Zoom	9	2	5	9	2	1

$$q_{CFQ} = 2$$

second quantum



Ordered Dispatch Queue



Use the table to put the request into each queue and then to the single dispatch queue.

A) Call of Duty

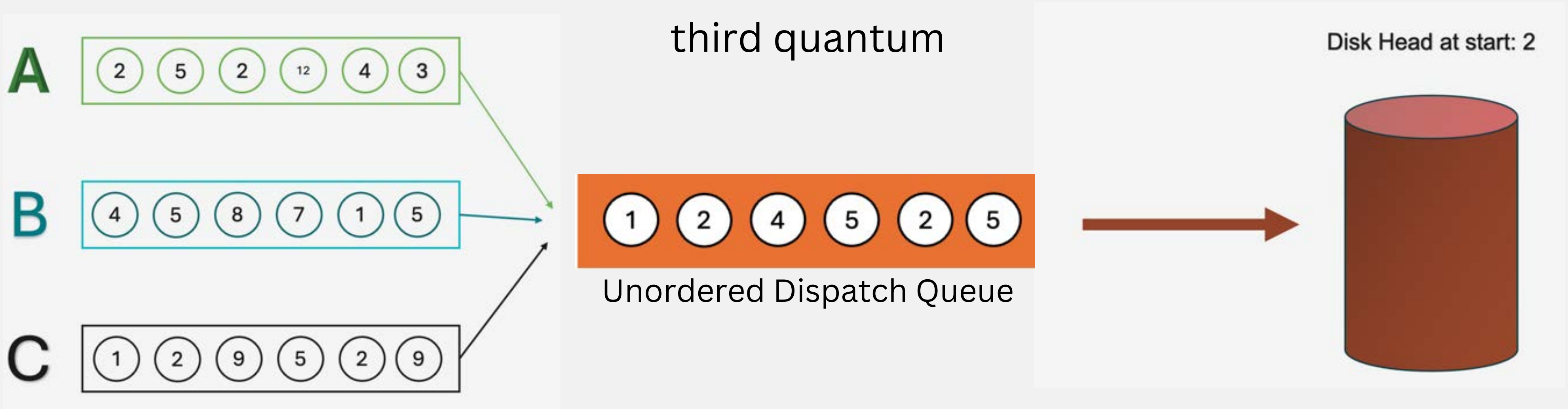
B) Spotify

C) Zoom

$$q_{CFQ} = 2$$

	Disk head for Request 1	Disk head for Request 2	Disk head for Request 3	Disk head for Request 4	Disk head for Request 5	Disk head for Request 6
Call of Duty	3	4	12	2	5	2
Spotify	5	1	7	8	5	4
Zoom	9	2	5	9	2	1

third quantum



Use the table to put the request into each queue and then to the single dispatch queue.

A) Call of Duty

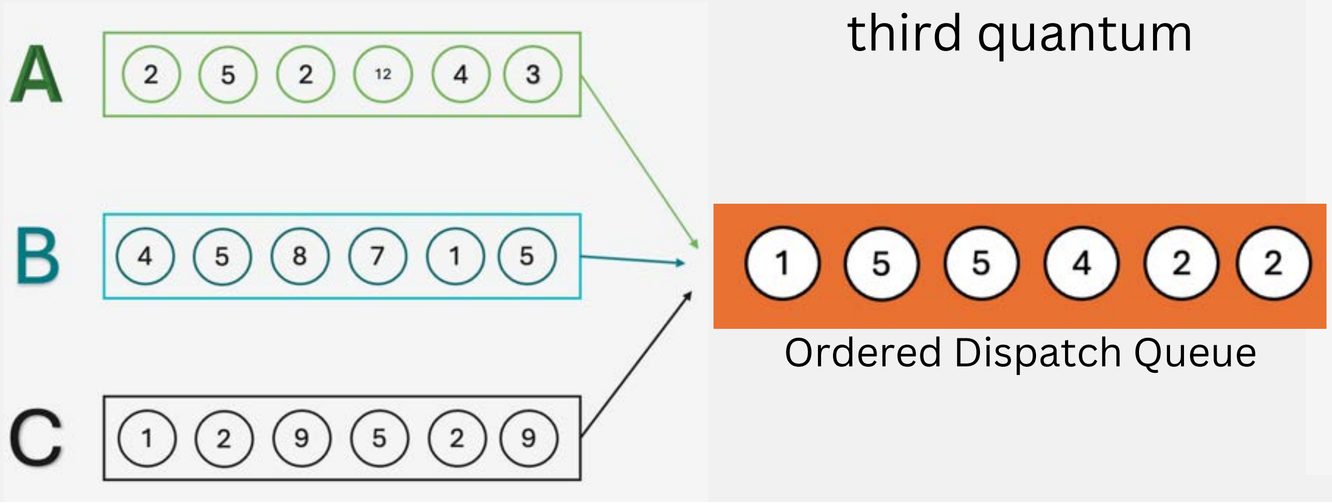
B) Spotify

C) Zoom

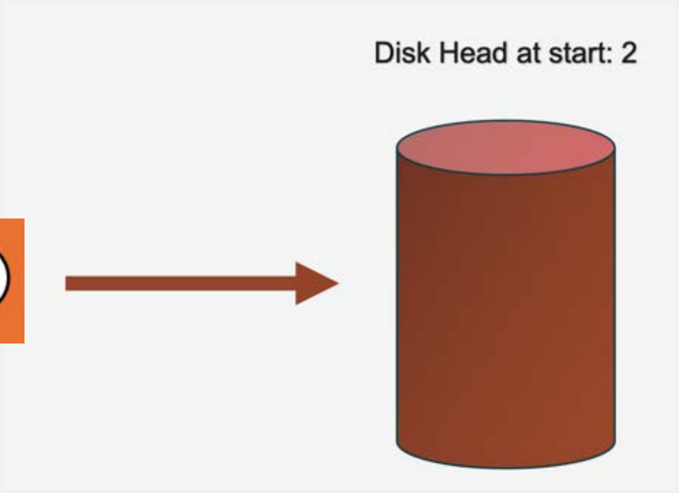
$$q_{CFQ} = 2$$

	Disk head for Request 1	Disk head for Request 2	Disk head for Request 3	Disk head for Request 4	Disk head for Request 5	Disk head for Request 6
Call of Duty	3	4	12	2	5	2
Spotify	5	1	7	8	5	4
Zoom	9	2	5	9	2	1

third quantum



Ordered Dispatch Queue

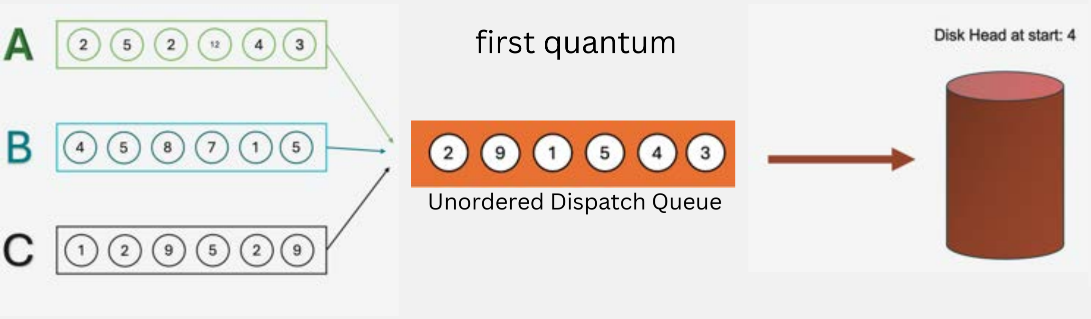


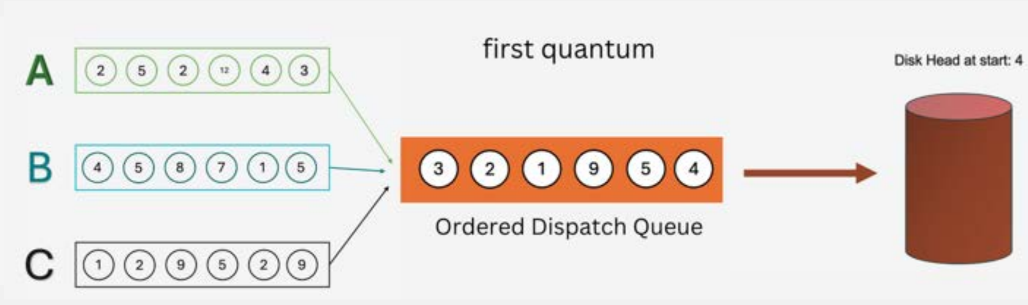
"If the disk head starts at position 4, calculate the total head movement required to service all requests in:

$$q_{CFQ} = 2$$

a) The original dispatch queue for the first quantum

b) The reordered dispatch queue for the first quantum



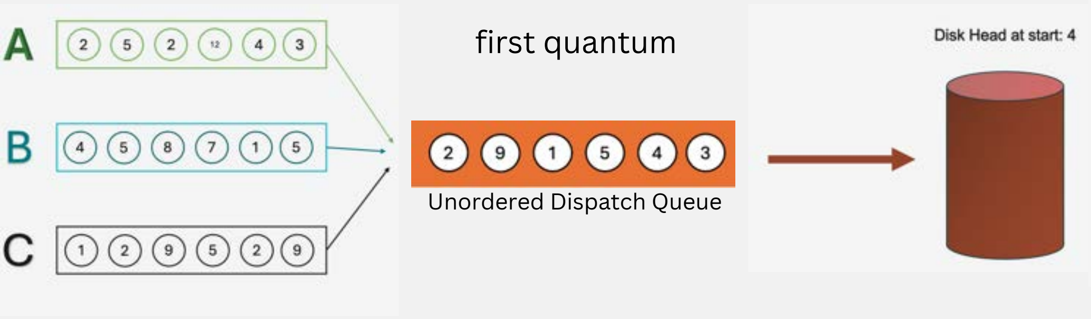


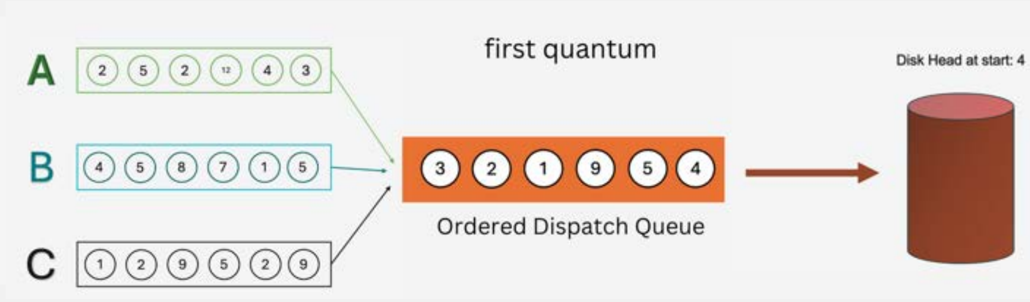
"If the disk head starts at position 4, calculate the total head movement required to service all requests in:

$q_{CFQ} = 2$

a) The original dispatch queue for the first quantum

b) The reordered dispatch queue for the first quantum





22 for Original Queue

15 for Ordered Queue if we use C-SCAN variant (not down to 0)