National University of Computer and Emerging Sciences

FAST School of Computing **Fall-2023** **Islamabad Campus**

CS-2006: Operating Systems BSSE-P,Q

Tuesday, 7th November, 2023

Course Instructors

Khwaja Bilal Hassan

Serial No:

2nd Sessional Exam

Total Time: 1 Hour

Total Marks: 50

_____

Signature of Invigilator

_____	_____	_____	_____
Student Name	Roll No.	Course Section	Student Signature

DO NOT OPEN THE QUESTION BOOK OR START UNTIL INSTRUCTED.

Instructions:

1. Attempt on question paper. Attempt all of them. Read the question carefully, understand the question, and then attempt it.
2. No additional sheet will be provided for rough work. Use the back of the last page for rough work.
3. If you need more space, write on the back side of the paper and clearly mark question and part number etc.
4. After asked to commence the exam, please verify that you have **Nine** different printed pages including this title page. There are total of **Six** questions.
5. Calculator sharing is strictly prohibited.
6. Use permanent ink pens only. Any part done using soft pencil will not be marked and cannot be claimed for rechecking.

	Q-1	Q-2	Q-3	Q-4	Q-5	Q-6	Total
Marks Obtained							
Total Marks	**10**	**05**	**08**	**08**	**09**	**10**	**50**

Question 1 [10 Marks]

Select the appropriate answer in the following by encircling them. No cutting or overwriting allowed

a) In the context of scheduling, what does "preemptive" mean?
   a) Once a process starts executing, it cannot be interrupted.
   b) A process can be interrupted and moved out of the CPU.
   c) Processes are scheduled in the order they arrive.
   d) Processes are scheduled based on their size.

b) What is the main advantage of the Round Robin scheduling algorithm?
   a) Minimizes the turnaround time of processes
   b) Reduces the waiting time of processes
   c) Guarantees that high-priority processes always execute first
   d) Ensures that no process monopolizes the CPU for an extended period

c) What is a potential issue with the Round Robin (RR) scheduling algorithm when the time quantum is too large?
   a) It may lead to excessive context switching.
   b) It may result in priority inversion.
   c) It may cause processes to be starved.
   d) It may increase the average turnaround time.

d) A higher value of the smoothing factor (alpha) in exponential averaging for SJF service time prediction results in:
   a) A more stable and less responsive estimate
   b) A more responsive to recent changes
   c) No impact on the estimation process
   d) Perfect estimation without any variations

e) In a(n) _____ temporary queue, the sender must always block until the recipient receives the message.
   a) Zero capacity
   b) Fixed capacity
   c) Variable capacity
   d) Bounded capacity
   e) Unbounded capacity

f) Which of the following is a fundamental problem in multi-threading that synchronization aims to address?
   a) Deadlock
   b) Starvation
   c) Race condition
   d) Thread creation

g) What is the primary advantage of User-Level Threads (ULTs) over Kernel-Level Threads (KLTs)?
   a) ULTs provide better performance and lower overhead.
   b) KLTs allow more fine-grained control over hardware resources.
   c) ULTs are better at handling I/O operations.
   d) KLTs simplify the management of thread-specific data.

h) In Unix-like operating systems, which of the following is a key difference between fork() and clone()?
   a) fork() creates a new process, while clone() creates a new thread.
   b) fork() creates an exact copy of the process, while clone() creates a new process.
   c) fork() creates a new thread, while clone() creates an exact copy of the process.
   d) fork() and clone() are equivalent in their functionality.

i) When passing a pointer to a local variable to a thread, what should you be cautious about?
   a) It's always safe to pass pointers to local variables.
   b) The local variable must be declared as static.
   c) The local variable's scope must outlive the thread's execution.
   d) Local variables cannot be passed to threads.

j) In Peterson's Algorithm, what are the two key variables that each process uses to indicate its interest in entering the critical section?
   a) lock and unlock
   b) interested and turn
   c) entry and exit
   d) wait and signal

Question 2 [1+4 = 5 Marks]

Look at the given piece of Code:

int turn = -1; //assume this is shared	

//Process 0
void insert(int y)
{
turn = 0;
while(turn != 1);
stack[i] = x;
i++;
turn = 1;
}
	//Process 1
	void insert(int y)
	{
	turn=1;
	while (turn != 0);
	stack [i] = x;
	i++;
	turn = 0;
	}

a) Does this software solution solves the Mutual exclusion(Yes/No) [1]

Ans: Yes

b) Provides a scenario to proof your answer above? [4 marks]

The solution provide Mutual exclusion as when one of the process is in its critical section the other one will busy wait until the first is out of its critical section.
E.G.
P0: turn =0, While (0!=1) condition is true it will start circulating here.
P1: turn=1, while(1!=0) the condition is true it will also start circulating here but now p0 can enter its C.S
P0: while(1!=1) condition is false now hence p0 is allowed to its CS. When one is in its CS the other one is not allowed to go into its CS.

Hence it solves the mutual exclusion.

Question 3 [8 Marks]

Consider the following declarations and code segment [8 Marks]

```
#define Size 100
//Assume both arrays are initialized with 1s
```

```
int A[Size];
int B[Size];
void update(int* List, int Start, int Stop)
{
	for (int Idx = Start; Idx <= Stop; Idx++)
	{
		List[Idx]++;
	}
}
int add(int* List, int Start, int Stop)
{
	int Sum = List[Start];
	for (int Idx = Start + 1; Idx <= Stop; Idx++)
	{
		Sum += List[Idx];
	}
	return Sum;
}
```

Suppose two threads are created and execute the function calls shown below. For each case, determine whether the execution of the two threads involves a race condition.

Thread1	Thread2	Race condition?	Reason
add (A, 20, 30)	add (A, 20, 40)	No	Although the threads here are accessing similar data from 20 to 40 but as we are just accessing and both of them are just reading not doing any changes to existing array so no race condition occur here.
update (A, 50, 75)	update (B, 50, 75)	No	As both of threads are working on a separate arrays son no race condition, occur here.
update (B, 0, 20)	update (B, 21, 40)	No	Both of the threads here are accessing different non-overlapping parts so no issue in that.
add (A, 20, 30)	update (A, 25, 35)	Yes	Race condition can occur here as both of them are accessing and one of it is updating the overlapped part. Based on that the final array and final sum will be affected.

Question 4 [8 Marks]

Calculate TAT and AWT for the following Ready Queue scenario by using following algorithms.
Note: The processor time start from 0 and there can be time where CPU may be Idle.

a) RR with Quantum 2

Process	Arrival Time	Service Time	Finish time	TAT	WT
A	0	3	5	5	2
B	2	6	17	15	9
C	4	4	13	9	5
D	6	5	20	14	9
E	8	2	15	7	5
				10	**06**

Ans:

A	B	A	C	B	D	C	E	B	D	D

0 2 4 5 7 9 11 13 15 17 19 20

HRRN

Process	Arrival Time	Service Time	Finish time	TAT	WT
A	0	3	3	3	0
B	2	6	9	7	1
C	4	4	13	9	5
D	6	5	20	14	9
E	8	2	15	7	5
				8	**04**

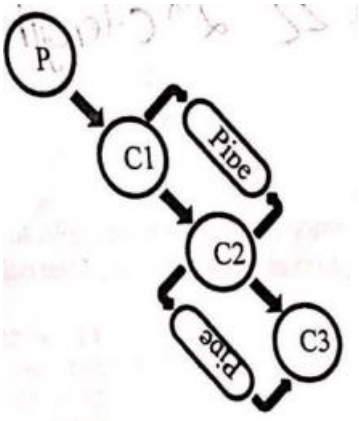
A	B	C	E	D

0 3 9 13 15 20

Question 5 [09 Marks]

Question:

Write a C/C++ program to implement the below given scenario.
There are four processes: Parent P and Children's C1, C2 and C3 as shown in the below hierarchy. C1 communicates with C2 and then C2 communicates with C3.



Having a structure of Rect shown below:

struct Rect

```
{
    int length, int width;
}
```

- Parent process create the object of Rect and fill the values of length and width from user.
- C1 Process send Rect object to C2.
- C2 Receives the object and print the area of rectangle and forward that object to C3.
- C3 Receives the object and print the perimeter of rectangle.

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

struct Rect {
    int length;
    int width;
};

int main() {
    int pipeC1toC2[2], pipeC2toC3[2];
    pipe(pipeC1toC2);
    pipe(pipeC2toC3);

    pid_t pidC1, pidC2, pidC3;
    Rect rect;

    std::cout << "Enter length: ";
    std::cin >> rect.length;
    std::cout << "Enter width: ";
    std::cin >> rect.width;

    pidC1 = fork();
    if (pidC1 == 0) {
        // C1 process
        close(pipeC1toC2[0]); // Close read end
        write(pipeC1toC2[1], &rect, sizeof(Rect));
        close(pipeC1toC2[1]); // Close write end

        pidC2 = fork();
        if (pidC2 == 0) {
            // C2 process
            close(pipeC1toC2[1]); // Close write end
            close(pipeC2toC3[0]); // Close read end

            Rect receivedRect;
            read(pipeC1toC2[0], &receivedRect, sizeof(Rect));
            close(pipeC1toC2[0]); // Close read end

            int area = receivedRect.length * receivedRect.width;
            std::cout << "Area of rectangle: " << area << std::endl;
```

```
            write(pipeC2toC3[1], &receivedRect, sizeof(Rect));
            close(pipeC2toC3[1]); // Close write end

            pidC3 = fork();
            if (pidC3 == 0) {
                // C3 process
                close(pipeC2toC3[1]); // Close write end

                Rect receivedRect;
                read(pipeC2toC3[0], &receivedRect, sizeof(Rect));
                close(pipeC2toC3[0]); // Close read end

                int perimeter = 2 * (receivedRect.length + receivedRect.width);
                std::cout << "Perimeter of rectangle: " << perimeter << std::endl;
            }
        }
    }

    return 0;
}
```

Question 6 [10 Marks]

Write a C/C++ program to simulate the following scenario. The program should:

- Take a large unsorted array from user.
- Divide the array into smaller chunks given by user, assigning each chunk to a worker thread for parallel sorting.
- Each worker thread works on chunk of an array and sort it out.
- In the main thread to obtain the final sorted array while waiting for all the worker threads to finish their work.
- Print the final sorted array.

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

#define MAX_ARRAY_SIZE 10000

// Struct to pass chunk information to the thread
typedef struct {
    int* arr;
    int start;
    int end;
} ChunkInfo;
```

```
int cmpfunc(const void* a, const void* b) {
   return (*(int*)a - *(int*)b);
}

// Function to sort a chunk of the array
void* sortChunk(void* arg) {
   ChunkInfo* chunkInfo = (ChunkInfo*)arg;
   qsort(chunkInfo->arr + chunkInfo->start, chunkInfo->end - chunkInfo->start, sizeof(int),
&cmpfunc);
   return NULL;
}


int main() {
   int array[MAX_ARRAY_SIZE];
   int arraySize, chunkSize;

   printf("Enter the size of the array: ");
   scanf("%d", &arraySize);

   printf("Enter the elements of the array: ");
   for (int i = 0; i < arraySize; ++i) {
      scanf("%d", &array[i]);
   }

   printf("Enter the number of chunks: ");
   scanf("%d", &chunkSize);

   int chunkLength = arraySize / chunkSize;
   pthread_t threads[chunkSize];

   // Create threads for sorting chunks
   for (int i = 0; i < chunkSize; ++i) {
      ChunkInfo* chunkInfo = (ChunkInfo*)malloc(sizeof(ChunkInfo));
      chunkInfo->arr = array;
      chunkInfo->start = i * chunkLength;
      chunkInfo->end = (i == chunkSize - 1) ? arraySize : (i + 1) * chunkLength;
      pthread_create(&threads[i], NULL, sortChunk, (void*)chunkInfo);
   }

   // Join all threads to wait for their completion
   for (int i = 0; i < chunkSize; ++i) {
      pthread_join(threads[i], NULL);
   }

   // Merge and sort the chunks to obtain the final sorted array
   qsort(array, arraySize, sizeof(int), &cmpfunc);

   // Print the final sorted array
```

```
    printf("Final sorted array: ");
    for (int i = 0; i < arraySize; ++i) {
        printf("%d ", array[i]);
    }
    printf("\n");

    return 0;
}
```

Another solution

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

typedef struct {
    int* array;
    size_t start;
    size_t end;
} ChunkInfo;

void quicksort(int* array, size_t start, size_t end) {
    if (start < end) {
        int pivot = array[end];
        size_t i = start - 1;

        for (size_t j = start; j < end; j++) {
            if (array[j] <= pivot) {
                i++;
                int temp = array[i];
                array[i] = array[j];
                array[j] = temp;
            }
        }

        int temp = array[i + 1];
        array[i + 1] = array[end];
        array[end] = temp;

        size_t partition = i + 1;

        // Recursively sort the two halves
        quicksort(array, start, partition - 1);
        quicksort(array, partition + 1, end);
    }
}
```

```
void* workerThread(void* arg) {
    ChunkInfo* chunk = (ChunkInfo*)arg;
    quicksort(chunk->array, chunk->start, chunk->end);
    pthread_exit(NULL);
}

int main() {
    int n, numChunks;

    printf("Enter the size of the array: ");
    scanf("%d", &n);

    printf("Enter the number of chunks: ");
    scanf("%d", &numChunks);

    int* array = (int*)malloc(n * sizeof(int));
    printf("Enter the elements of the array:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &array[i]);
    }

    pthread_t threads[numChunks];
    ChunkInfo chunks[numChunks];

    size_t chunkSize = n / numChunks;
    for (int i = 0; i < numChunks; i++) {
        chunks[i].array = array;
        chunks[i].start = i * chunkSize;
        chunks[i].end = (i == numChunks - 1) ? (n - 1) : (chunks[i].start + chunkSize - 1);

        pthread_create(&threads[i], NULL, workerThread, (void*)&chunks[i]);
    }

    for (int i = 0; i < numChunks; i++) {
        pthread_join(threads[i], NULL);
    }

    // Merge the sorted chunks (not needed in this case since the chunks are already sorted)

    printf("Final sorted array:\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", array[i]);
    }
    printf("\n");

    free(array);

    return 0;
}
```

R/W