Q1. [10 pts] Consider the following program pseudo-code. Find out all the values printed out by the program. Order is not important.

```
 int x = 2000
 int y = 1000
 main() {
          n = fork ()
          if (n == 0) {
                    x = 2500
                    y = 1200
          }
          else {
                    n = fork()
                    if n == 0 {
                              x = 2200
                              y = 1300
                    }
                    x = 2100
                    print (x)
          }
          print (x,y)
}
```

Answer:

```
X=2000
Y=1000
|------------------------------|
                          X = 2500
                          Y=1200
                          Print (2500,1200)
|
|
-------------------------|
                        X = 2200
                        Y = 1300
                        X = 2100
                        Print (2100)
                        Print (2100, 1300)
X=2100
Print (2100)
Print (2100, 1000)
```

Q2. [10 pts] An application (a multi-threaded process) has a part that must be executed serially and the rest is parallelizable (with multiple threads) depending on the number of CPUs (i.e., the process is using as many threads as there are CPUs). In a single-CPU computer, it runs in 4 hours. When we run the application in a computer with 4 CPUs, it takes 2 hours for the application to complete. How long will it take if we run the application in a computer with 8 CPUs?

Let us call the fraction of the program that must be run serially as x.

1 / (x + (1-x)/4) = 4/2 = 2 → **x = 1/3**

Speedup in 8 CPU compared to 1 CPU?

1 / (x + (1-x/8)) = 1 / (x + ((1-x)/8) = 1 / (8x + 1 – x) / 8 = 1 / (7x + 1) / 8 = 8 / (7x + 1) = 8 / (7/3 + 1) = 8 / (10 / 3) = **24 / 10** is the speed up.

4 / 24 / 10 = **40 / 24 = 10 / 6** hours = **5/3 hours; It** will take.

Q3. [20 pts] The following 5 processes are given. Schedule the processes (i.e., draw the Gantt chart) using FCFS, SJF, SRTF, and RR (q = 20) scheduling. For each algorithm, find out the finish time (i.e., the time the burst execution will complete) of each process.

Process	Arrival Time (ms)	CPU Burst Length (ms)
A	0	80
B	10	60
C	45	90
D	55	50
E	70	40

Answer:

a) AAAAAAAA BBBBBB CCCCCCCCC DDDDD EEEE
A=80, B=140, C=230, D=280, E=320

b) AAAAAAAA EEEE DDDDD BBBBBB CCCCCCCCC
A=80, E=120 , D=170, B=230, C=320.

c) ABBBBBBEEEEDDDDDAAAAAAACCCCCCCCC
each letter 10 units.
B=70, E=110, A=230, D=160, C=320

d) AABBAABBCCDDAAEEBBCCDDAAEECCDCCC
B=180, A=240, E=260, C=320, D=290.

Q4. [15 pts] A system is using 36 bits virtual addresses. The physical memory is 512 MB. Page size is 4096 bytes and a page table entry is 4 bytes long no matter which page table structure is used. A process X is using 64 MB at the start of its virtual memory and 32 MB at the end of its virtual memory. That means total used virtual memory of the process is 96 MB. The rest of the virtual memory of the process is unused.

Answer the following questions.

a) How much physical memory (RAM) will be occupied by the page table information if *single-level page table* scheme is used?

2^36 / 2^12 = 2^24. 2^24 x 2^2 = 2^26 = **64 MB**

b) How much physical memory will be occupied by the page table information if *two-level page table* scheme with address division [12, 12, 12] is used?

A second level table can map:
2^12 x 2^12 = 2^24 = 16 MB.

64/16 = 4 and 32 / 16 = 2 ; therefore we need 6 inner tables.
Each table size is: 2^12 * 2^2 = 2^14. 16 KB.

We need 6 inner and 1 top level table; 7 tables. 7 x 16 = **112 KB.**

c) How much physical memory will be occupied by the page table information if *inverted page table* is used?

512 MB = 2^29. 2^29 / 2^12 = 2^17 entries needed. 2^17 x 2^2 = 2^19 = **512 KB.**

Q5. [10 pts] In a system that uses deadlock avoidance, we have the following Maximum Demand and Allocation Matrices for 5 processes. There are 3 resource types in the system: A, B, C. There exist 6 A, 5 B and 6 C (Existing = [6, 5, 6]). The current available resource instances are: 1 A, 1 B, 0 C (Available = [1, 1, 0]). Is this state safe or not? Prove your answer. Otherwise, no points.

	MaxDem	Allocation
	A B C	A B C
P1	3 5 1	2 0 1
P2	1 2 2	0 1 2
P3	3 1 4	1 0 1
P4	2 3 4	1 2 1
P5	1 3 3	1 1 1

Answer:
Safe:
Sequence = P2 P5 P3 P4 P1
Ya da
Sequence = P2 P5 P4 P1 P3
Ya da
Sequence = P2 P5 P4 P3 P1

Q6. [20 pts]. Answer the following questions.

a) In our application programs, when we make a system call, we are usually not using a system call number. Why?

It is because we call library functions. They call the system call directly using the system call number.

b) Write down two advantages of kernel-level thread implementation? Briefly explain.

1) Blocking system calls are not problem.
2) Multiple CPUs can be utilized since scheduling is done by the kernel.

c) Write down the possible methods that can be used to pass parameters to kernel.

There are 3 methods:

1) Use of a register or registers
2) Put the parameters into a table and passing the address of the table using a register.
3) Using the stack. Putting the parameters into a stack from where kernel can pop the parameters.

d) Write the 3 main events that cause kernel start executing in CPU.

Hw interrupts
Exceptions
System calls

e) Write one advantage and one disadvantage of shared memory as IPC mechanism compared to message passing.

Advantage: it is fast
Disadvantage: hard to program, we need to be careful about synchronization.

f) Briefly compare fork() and exec() system calls.

Fork is used to create a new process that is executing the same program that the parent is executing.

Exec is not creating a new process. It is changing the program that the process is running.

g) Which one of the following is not kept in PCB? Why?
a) process state, b) program counter value, c) process heap, d) process id.

Process heap is not kept in the PCB. It is not in kernel space.

Q7. [15 pts] Consider a file X that is to be accessed by two types of processes: type P and Q. The rule is that one or more P·processes can access the file at the same time, and one or more Q processes can access the file at the same time. But if a P process is accessing the file, no Q process can access it at the same time. Similarly, if a Q process is accessing the file, no P process can access it at the same time. P processes and Q processes are arriving at random times. For example: Q1, Q2, P1, P2, P3, Q3, P4, Q4, Q5, ..., can be a sequence of processes that may arrive.

Using *shared variables* and *binary semaphores* (i.e., semaphores that can take values 0 or 1), develop the following four functions that will enforce the access rules described above: P_lock(), P_unlock(), Q_lock(), and Q_unlock(). For example, if one or more P processes are accessing the file at the moment, a Q process that wants to access the file needs to be waited (blocked) in function Q_lock(). *Your solution does not have to be starvation-free.*

Below are pseudo-codes for P and Q processes, showing how these functions are used.

```
P Process:                     Q Process:
...                            ...
P_lock()                       Q_lock()

// access the file X           // access the file X

P_unlock()                     Q_unlock()
...                            ...
```

Answer:

We will have the following shared variables:
int countP = 0 // num of P processing accessig the file
int count Q = 0 // num of Q processes accessin the file
semaphore semP initialized to 1; // semaphore protecting countP
semaphore semQ initialized to 1; // semaphore protecting countQ
semaphore semX initialized to 1; // semaphore for file X

```
P_lock() {                         Q_lock() {
        wait(semP);                        wait(semQ);
        countP = countP + 1;               countQ = countQ + 1;
        if countP == 1 then                if countQ == 1 then
                wait(semX)                         wait(semX)
        signal(semP)                       signal(semQ)
}                                  }
```

```
P_unlock()                         Q_unlock()
{                                  {
        wait(semP)                         wait(semQ)
        countP = countP – 1                countQ = countQ – 1
        if countP == 0 then                if countQ == 0 then
                signal(semX)                       signal(semX)
        signal(semP)                       signal(semQ)
}                                  }
```