Name - Vinayak Sethi   Roll No - COE18B061

OPERATING SYSTEMS

END ASSESSMENT

Part B

Q1.) Claim(C)

	R_0	R_1	R_2
P_0	4	1	2
P_1	1	5	1
P_2	1	2	3

Allocation

R_0	R_1	R_2
1	0	2
0	3	1
1	0	2

Available / free

R_0	R_1	R_2
2	2	0

1.) Can P_0 be scheduled?

Future Req $\Rightarrow$ $\boxed{3 \mid 1 \mid 0}$ > Free $\Rightarrow$ not allowed

$\therefore$ The future req cannot be fulfiled as there is no free resources available, hence this cannot be processed.

2.) Can P_1 be scheduled?

Future Req $\Rightarrow$ $\boxed{1 \mid 2 \mid 0}$ < Free $\Rightarrow$ Allowed

so, required resources are allocated to P_1 $\Rightarrow$ Free Before completion (P_1) $\Rightarrow$ $\boxed{1 \mid 0 \mid 0}$

P_1 runs till completion and frees its resource

Claim (C)

	R_0	R_1	R_2
P_0	4	1	2
P_1	0	0	0
P_2	1	2	3

Allocation

R_0	R_1	R_2
1	0	2
0	0	0
1	0	2

Free After completion of P_1 $\boxed{2 \mid 5 \mid 1}$

$\therefore$ Free was $\boxed{1 \mid 0 \mid 0}$ when P_1 processed and released $\boxed{2 \mid 5 \mid 1}$ after completion

$P_1 \rightarrow P_0$?

$\Rightarrow$ $\boxed{3 \mid 1 \mid 0}$ > Free $\Rightarrow$ not allowed

$P_1 \rightarrow P_2$?

$\Rightarrow$ $\boxed{0 \mid 2 \mid 1}$ < free $\Rightarrow$ Allowed

P_2 runs till completion and frees its resource

Free before completion of P_2 $\Rightarrow$ $\boxed{2 \mid 3 \mid 0}$

Free after completion of P_2 $\Rightarrow$ $\boxed{3 \mid 5 \mid 3}$

Claim

	R_0	R_1	R_2
P_0	4	1	2
✓P_1	0	0	0
✓P_2	0	0	0

Allocation

R_0	R_1	R_2
1	0	2
0	0	0
0	0	0

Free

$\boxed{3 \mid 5 \mid 3}$

$P_1 \rightarrow P_2 \rightarrow P_0$? $\boxed{3 \mid 1 \mid 0}$ < free $\Rightarrow$ Allowed

P_0 runs till completion and frees its resource

Free before completion of P_0 $\Rightarrow$ $\boxed{0 \mid 4 \mid 3}$

Free after completion of P_1 $\Rightarrow$ $\boxed{4 \mid 5 \mid 5}$

All process complated running $\Rightarrow$ $P_1 \rightarrow P_2 \rightarrow P_0$

Claim

	R_0	R_1	R_2
✓P_0	0	0	0
✓P_1	0	0	0
✓P_2	0	0	0

Allocation

R_0	R_1	R_2
0	0	0
0	0	0
0	0	0

Free

$\boxed{4 \mid 5 \mid 5}$

$\Rightarrow$ $\boxed{\text{Safe state}}$

(b) P_0 requests $1 \rightarrow R_1$ $\Rightarrow$ $\boxed{0 \mid 1 \mid 0}$ < Free $\Rightarrow$ Allowed

(∴ This will not run till complation, it just allocate it)

	R_0	R_1	R_2
P_0	4	1	2
P_1	1	5	1
P_2	1	2	3

Allocation

R_0	R_1	R_2
1	1	2
0	3	1
1	0	2

This is allocation

Free after processing request of P_0

2	1	0

$P_0 \, (0,1,0) \Rightarrow$ [3 | 0 | 0] > free $\Rightarrow$ Not allowed

$P_1 \Rightarrow$ [1 | 2 | 0] > Free $\Rightarrow$ Not allowed

$P_2 \Rightarrow$ [0 | 2 | 1] > Free $\Rightarrow$ Not allowed

From the free resource, there is no process which can have ~~access~~ access to them (resources) at fullest (i.e their claim to run and complete).

$\therefore$ This gets us into unsafe state.

$\Rightarrow$ ($P_0 \, (0,1,0) \Rightarrow$ Anything) cannot be safe order

$\Rightarrow$ Unsafe order $\rightarrow$ Unsafe state

Q2-) The statement referred above is Amdahl's law. This law gives the theoritical speedup in latency of the execution of a task at a fixed workload. that can be expected of a system whose resources are improved.

→ The speedup due to addition of processor is not linear because of serialness in the program. This can be improved as below.

Let n be a multiprocessor setup

$T(n)$ time to run a program on n processor setup

$T(1)$ time to run on single process setup.

Speedup $S(n) \Rightarrow S(n) = \frac{T(1)}{T(n)}$

As every code has some serialness in it.

Let $T(1)$ has x amount of serialness.

$\Rightarrow$ It takes $x . T(1)$ ~~for~~ to run serial part and

$[(1-x) \times T(1)/n]$ to run parallel part.

$$\Rightarrow T(n) \Rightarrow T(1)\left(x + \frac{(1-x)}{n}\right)$$

$$\Rightarrow S(n) = T(1) / T(n)$$

$$\Rightarrow S(n) = \frac{1}{\left(x + \frac{(1-x)}{n}\right)}$$

Let say theoretically $n \to \infty$

$$\Rightarrow S(n) = \frac{1}{x}$$

$\Rightarrow$ Max speedup is inverse of serialness.

Enhanced new CPU is 50 times faster than old CPU

older CPU $\Rightarrow$ 60% of time processing

40% I/O $\Rightarrow$ this is serial component in the program.

Infinite computing capacity means $n \to \infty$

$$\text{so speed up} = \frac{1}{0.4} = 2.5$$

So maximum speedup achieved is 2.5 times.

Q3.) Dekker's Algorithm - It was first provably - correct solution to the critical section problem.

Pseudocode:

```
var flag : array [0...1] of boolean;
turn : 0...1;

repeat

    flag[i] = true
    while flag[j] do
        if turn = j then
        begin
            Flag[i] = false ;
            while turn = j do no-op;
            Flag[i] = true ;
        end
        critical section
    turn = j;
    flag[i] = false

        remainder section
    until false;
```

Proof of correctness for Dekker's Algorithm for synchronization

The required conditions for synchronization are:-

1.) Mutual Exclusion

2.) Progress

3.) Bounded Wait

1.) To show that mutual exclusion is enforced

Note : We are taking 2 process in consideration P_0, P_1

→ Here when process P_1 enters critical section iff flag[0] == false (P_1 can modify only flag[1] also it checks flag[0] if flag[0] == true)

The case is when P_1 enters the critical section, flag[1] & !flag[0] = true

[Note: P_0 can only change flag [0] and P_1 can only change flag[1]]

2.) To show there is no indefinite delay for acquiring the critical section.

↳ (a) When one process tries to access CS (critical section)

If P_i attemps to access critical section, it will check flag[1-i] set to false and enter critical section without any problem.

[Note: P_0 will find flag [1] is false and move to critical section, P_1 will find flag [0] is false and move to critical section]

(b) When both process try to access critical section

When both attempt to enter critical section and turn = 0 (or 1) (similar for both cases), both enter the while loop and value of turn is modified only when one process enter critical section.

[Note: Turn is made 1 or 0 either by P_0 or P_1. If turn is same for both, then one of them enters critical section, the value of turn will change for other process and hence allowing it to exit CS]

↳ (i) Both attempting to enter critical section when turn = 0 and flag [0] = false

Here as soon as P_1 find flag [0] is false, it enters the critical section which ensures <u>Progress</u>.

↳ (ii) Both attempting to enter critical section when turn = 0 and flag [0] = true

Here P_0 will wait in external loop till flag [1] = false (value of flag [0] not being modified].

P_1 will set flag [1] = false and wait in internal loop (as turn = 0). As soon as this is done, P_0 enters critical section.

Pseudocode for producer consumer using dekker's algorithm

```
void producer ()
{
    count = 1;
    while (1) {
            int data, added = 1, err;
            if (added) {
                    data = rand ()
                    added = 0;
        // Dekker algorithm
                    c1 = 1;
~~void consumer()~~
                    while (c2) {
~~{~~
                        if (turn == 2) {
                            c1 = 0;
                            while (turn == 2)
                            c1 = 1;
                        }
                    }
            err = addQ(&q, data)
            turn = 2;
            c1 = 0;
            if (err = ok) {
                    added = 1;
                    print (producer data);
            }
        }
}
```

```
void consumer () {
    int count = 1;
    while (1) {
        int data, err;
// Dekker algorithm
      c2 = 1;

      while (c1){
          if (turn = = 1){
              c2 = 0;
              while (turn == 1);
              c2 = 1;
          }
      }

        err = front Q (& Q ,& data)
         iff. (err = = ok) err = delete Q(&q);

        turn = 1;
         c2 = 0;

         if (err = = ok)
             printf (Consumed data);
        }
}
```

~~Part~~

Q4-) Interrupts are not used to implement synchronization. It basically depends on how we tend to use them but regardless of that this is really poor choice of technique that can be considered.

Certain problems we will face if they are to be implemented

1.) <u>Disabled on one core</u>

This could lead to nothing but ignorance as the threads running on other cores will still be using the shared data and ignore the synchronization primitives.

2.) <u>Disabled on all cores</u>

i.) <u>Priority</u> - If something of high priority (urgent) occurs, kernel might need interrupt but this cannot happen as they are disabled and hence the high priority task will be ignored.

ii.) <u>Lack of interrupts</u> : Context switching, I/O operations or any other operations will not be used by os as if requires trap in kernel.

iii.) <u>Clock lag</u> : If clock relies on interrupt, it will start logging.

iv.) <u>Early exit</u> : If user app is exited due to an error or exception it will not be freed / released synchronization primitive as interrupt is disabled.

<u>Part C</u>

PART C

Q1.) A *pipe* is a form of *redirection* that is used in Linux and other Unix-like operating systems to send the output of one program to another program for further processing.

Redirection is the transferring of *standard output* to some other destination, such as another program, a file or a printer, instead of the display monitor (which is its default destination). Standard output, sometimes abbreviated *stdout*, is the destination of the output from *command line* (i.e., all-text mode) programs in Unix-like operating systems.

Pipes are used to create what can be visualized as *a pipeline of commands*, which is a temporary direct connection between two or more simple programs. This connection makes possible the performance of some highly specialized task that none of the constituent programs could perform by themselves. A *command* is merely an instruction provided by a user telling a computer to do something, such as launch a program. The command line programs that do the further processing are referred to as *filters*.

This direct connection between programs allows them to operate simultaneously and permits data to be transferred between them continuously rather than having to pass it through temporary text files or through the display screen and having to wait for one program to be completed before the next program begins.

Code:

```
#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<sys/wait.h>
```

```c
#include<sys/types.h>

#include<string.h>

#include<ctype.h>

#include<fcntl.h>


void print_uppercase(char buff);


int main(int argc,char *argv[])

{

   if(argc == 1)

   {

       printf("Invalid arguments !!\n try : ./<%s> filename\n",argv[0]);

       exit(1);

   }

   int fd1[2];

   char str[100];

   pid_t pid;

   if((pipe(fd1))== -1)

   {

       fprintf(stderr,"pipe failed");

       return 1;

   }

   FILE *fp;

   fp = fopen(argv[1],"r");
```

```c
pid = fork();

if(pid == -1)

{

    printf("fork failed\n");

    exit(1);

}

else if (pid == 0)

{

    dup2(fd1[1],1);

    fp = fopen(argv[1],"r");

    close(fd1[0]);

    char ch;

    while ((ch = fgetc(fp)) != EOF)

    {

        write(fd1[1],&ch,sizeof(ch));

    }

    close(fd1[1]);

    exit(1);

}

else

{

    close(fd1[1]);

    char buf;
```

```c
        while(read(fd1[0],&buf,sizeof(buf))>0)

        {

            print_uppercase(buf);

        }

    }

    return 0;

}

void print_uppercase(char buff)

{

        printf("%c",toupper(buff));

}
```

Output:

```
File   Edit   View   Search   Terminal   Help
vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/EndSem$ gcc Q1_AllCaps.c -o Q1_AllCaps
vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/EndSem$ ./Q1_AllCaps
Invalid arguments !!
 try : ./<./Q1_AllCaps> filename
vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/EndSem$ ./Q1_AllCaps Input.txt
HELLO!
MY NAME IS VINAYAK SETHI
I AM 3RD YR CSE STUDENT AT IIITDM KANCHEEPURAM
BYEEE
vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/EndSem$ cat Input.txt
hello!
my name is vinayak sethi
i am 3rd yr cse student at iiitdm kancheepuram
byeee
vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/EndSem$
```

Q2.) Coke Machine Problem

Logic:

Listing:

```
import random

class Shared:
    def __init__(self, start=5):
        self.cokes = start

def consume(shared):
    shared.cokes -= 1
    print shared.cokes

def produce(shared):
    shared.cokes += 1
    print shared.cokes

def loop(shared, f, mu=1):
    while True:
        t = random.expovariate(1.0/mu)
        time.sleep(t)
        f(shared)

shared = Shared()
fs = [consume]*2 + [produce]*2
threads = [Thread(loop, shared, f) for f in fs]
for thread in threads: thread.join()
```

The capacity is 10 cokes, and that machine is initially half full. So the shared variable cokes is 5. The program creates 4 threads, two producers and two consumers. They both run loop, but producers invoke produce and consumers invoke consume. These functions make unsynchronized access to a shared variable, which is a no-no. Each time through the loop, producers and consumers sleep for a duration chosen from an exponential distribution with mean mu. Since there are two producers and two consumers, two cokes get added to the machine per second, on average, and two get removed. So on average the number of cokes is constant, but in the short run in can vary quite widely

Initialization:

Listing:

```
class Shared:
    def __init__(self, start=5, capacity=10):
        self.cokes = Semaphore(start)
        self.slots = Semaphore(capacity-start)
        self.mutex = Semaphore(1)
```

Producer, Consumer pseudocode:

Listing:

```
def consume(shared):
    shared.cokes.wait()
    shared.mutex.wait()
    print shared.cokes.value()
    shared.mutex.signal()
    shared.slots.signal()

def produce(shared):
    shared.slots.wait()
    shared.mutex.wait()
    print shared.cokes._Semaphore__value
    shared.mutex.signal()
    shared.cokes.signal()
```

Code:

```
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#include<semaphore.h>
#include<unistd.h>
#include<time.h>

sem_t coke, slots, mutex;
int ncoke = 5; //count of cokes

void *producer(void *arg);
```

```c
void *consumer(void *arg);

int main()
{
   pthread_t p[2], c[2]; //returns the thread id of thread created

   //initializes the unnamed semaphore at the address pointed to
   sem_init(&mutex, 0, 2);
   sem_init(&coke, 0, 1);
   sem_init(&slots, 0, 1);

   //creates a new thread
   for(int i=0; i<2; i++)
   {
       pthread_create(&p[i], NULL, producer, NULL);
       pthread_create(&c[i], NULL, consumer, NULL);
   }

   //wait for termination of the thread
   for(int i=0; i<2; i++)
   {
       pthread_join(p[i], NULL);
       pthread_join(c[i], NULL);
   }

   //destroys the unnamed semaphore at the address pointed to
   sem_destroy(&coke);
   sem_destroy(&slots);
   sem_destroy(&mutex);
   return 0;
}

void *producer(void *arg)
{
   while(1)
   {
       sem_wait(&slots);
       sem_wait(&mutex);
       if (ncoke == 10)
           continue;
```

```
        ncoke++;
        printf("\n[PRODUCER] Number of cokes in the machine currently after 
production: %d\n", ncoke);
        sem_post(&mutex);
        sem_post(&slots);
        sleep(1);
    }

    pthread_exit(0); //to exit a thread
}

void *consumer(void *arg)
{
    while(1)
    {
        sem_wait(&coke);
        sem_wait(&mutex);
        if(ncoke == 0)
            continue;
        ncoke--;
        printf("[CONSUMER] Number of cokes in the machine currently after 
consumption: %d\n", ncoke);
        sem_post(&mutex);
        sem_post(&coke);
        sleep(1);
    }

    pthread_exit(0); //to exit a thread
}
```

Output:

```
vinayak@vinayak-Swift-SF315-52G:
File  Edit  View  Search  Terminal  Help
vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/EndSem$ gcc Q2_CokeMachineProblem.c -o Q2_CokeMachineProblem -lpthread
vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/EndSem$ ./Q2_CokeMachineProblem

[PRODUCER] Number of cokes in the machine currently after production: 6
[CONSUMER] Number of cokes in the machine currently after consumption: 5
[CONSUMER] Number of cokes in the machine currently after consumption: 4

[PRODUCER] Number of cokes in the machine currently after production: 5

[PRODUCER] Number of cokes in the machine currently after production: 6
[CONSUMER] Number of cokes in the machine currently after consumption: 5
[CONSUMER] Number of cokes in the machine currently after consumption: 4

[PRODUCER] Number of cokes in the machine currently after production: 5

[PRODUCER] Number of cokes in the machine currently after production: 6
[CONSUMER] Number of cokes in the machine currently after consumption: 5
[CONSUMER] Number of cokes in the machine currently after consumption: 4

[PRODUCER] Number of cokes in the machine currently after production: 5

[PRODUCER] Number of cokes in the machine currently after production: 6
[CONSUMER] Number of cokes in the machine currently after consumption: 5
[CONSUMER] Number of cokes in the machine currently after consumption: 4

[PRODUCER] Number of cokes in the machine currently after production: 5

[PRODUCER] Number of cokes in the machine currently after production: 6
[CONSUMER] Number of cokes in the machine currently after consumption: 5

[PRODUCER] Number of cokes in the machine currently after production: 6
[CONSUMER] Number of cokes in the machine currently after consumption: 5

[PRODUCER] Number of cokes in the machine currently after production: 6
[CONSUMER] Number of cokes in the machine currently after consumption: 5

[PRODUCER] Number of cokes in the machine currently after production: 6
[CONSUMER] Number of cokes in the machine currently after consumption: 5

[PRODUCER] Number of cokes in the machine currently after production: 6

[PRODUCER] Number of cokes in the machine currently after production: 7
[CONSUMER] Number of cokes in the machine currently after consumption: 6
[CONSUMER] Number of cokes in the machine currently after consumption: 5

[PRODUCER] Number of cokes in the machine currently after production: 6

[PRODUCER] Number of cokes in the machine currently after production: 7
[CONSUMER] Number of cokes in the machine currently after consumption: 6
[CONSUMER] Number of cokes in the machine currently after consumption: 5

[PRODUCER] Number of cokes in the machine currently after production: 6
[CONSUMER] Number of cokes in the machine currently after consumption: 5

[PRODUCER] Number of cokes in the machine currently after production: 6
[CONSUMER] Number of cokes in the machine currently after consumption: 5
^C
```
