OPERATING SYSTEMS

END ASSESS MENT

Part B

Qi) (laim(c)

	Ro	P.	R2
Po	4	1	2
ρ,	1	5	1
P2		2	3

Allocation

R	,	R,	R2	
	1	0	2	7
	0	3	1	
L	1	0	2	

Available Ifrea

2	R	RI	Ro
	O	2	2
	0	2	2

1) (an Po be scheduled ?

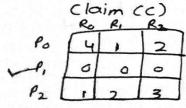
Future Rear = 3 1 0 > Free > not allowed

.. The future requannot be fulfilled as there is no free resources available, hence this cannot be processed.

2.) Can P, be scheduled?

Future Req = [120] < Free = Allowed So, required resources are allocated to P, => Free Before completion (p) => [100

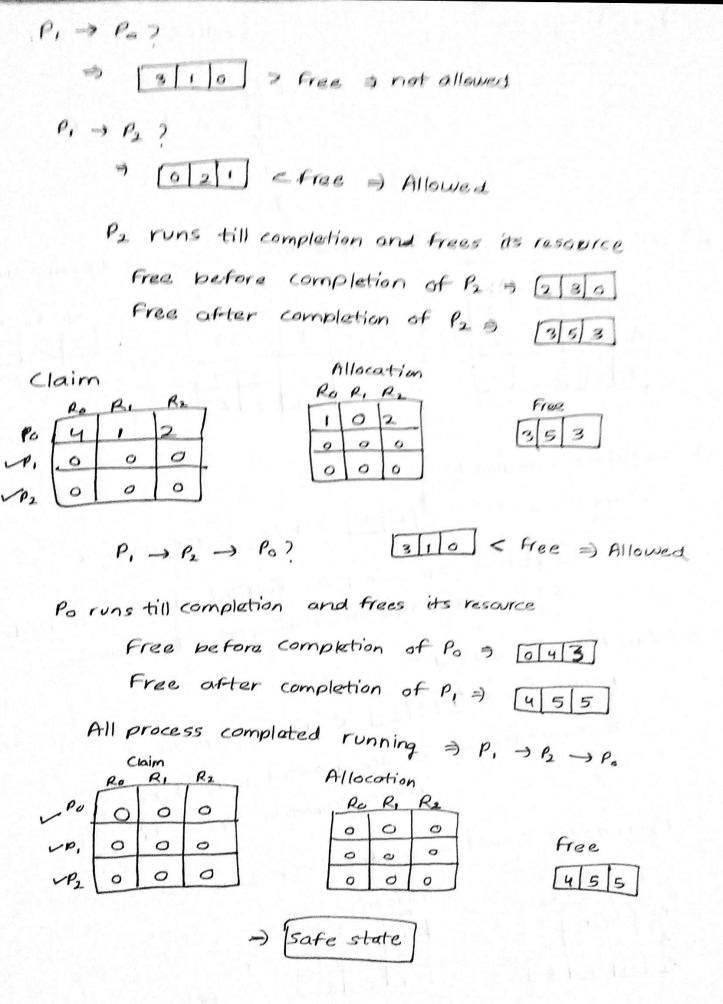
P, runstill Completion and frees its resource



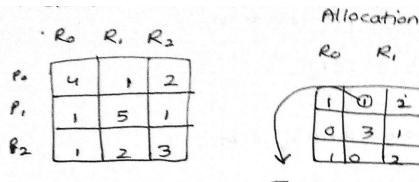
Ro	ation	LR.
_	0	2
0	0	0
,	0	12

Free After completion 251 of P,

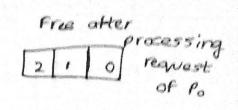
: Free was olo when P, processed and released 2 5 1 after completion



(: This will not run till completion, it just allocate it)



	1	0	2
	0	3	1
,	L	0	2



3 0 0 > free > Not allowed 120 > Free => Not allowed 021 > Free => Not allowed

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

.. This gets us into unsafe state.

=) Anything) cannot be safe order -) Unsafe order -> Unsafe state

- Q2-) The statement referred above is Andahi'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

Let nbe a multiprocessor setup T(n) time to run a program on n processor setup

Speedup
$$S(n) \Rightarrow S(n) = T(i)$$
 $T(n)$

As every code has some serialness in it.

Let T(1) has x amount of serialness.

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

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

$$\Rightarrow S(n) = \frac{1}{(x + \frac{(1-3c)}{n})}$$

Let say theoretically
$$n \rightarrow \infty$$

 $\Rightarrow S(n) = \bot$

=) Max speedup is inverse of serialness.

Enhanced new CPU is 50 times faster than old CPU

older CPU => 60% of time processing

40% Ilo => this is serial component in the program.

Infinite computing capacity means
$$n \to \infty$$

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 [i] do if turn = j then begin flag(i) = false; while turn = j do no-op; flag [i] = true;

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 Po, P,

-> Here when process P, enters critical section iff flag[0] = = false (P, can modify only flag [1] also it checks flag [0] if flag[i] ==

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

- Po can only change flag (0) and P, can only change flag [1] [Note:
- 2.) To show there is no indefinite delay for acquiring the critical section.
 - (a) When one process tries to access (s (critical section) If Pi attemps to access critical section, it will check flog[i-1]
 - set to false and enter critical section without any problem.
 - [Note: Po will find flag [1] is false and move to critical section, P, 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 to lord either by Po or Pi. It 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 (5)
 - (i) Both attempting to enter critical section when turn eq

Here as soon as P. findflag (0) is false, it enters the Critical section which ensures progress

scii) Both attempting to enter critical section when turn = 0

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

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

```
void producer ()
   count = 1;
   while (1) {
            int data, added = 1, err;
             if (added) {
                      data = rand ()
                       added = 0;
        11 Dekker algorithm
                     C1 = 1;
                     while (c) {
                          if (turn == 2) {
 $
                             CI = 0;
                             while (turn == 2)
                              c1 = 1;
                     3
             err = add Q(ba, data)
             turn = 2;
              C1=0;
              if Cerr = OK) &
                    added=1;
                     print (producerdata);
                 3
```

Pseudocode for producer consumer using dekker's algorithm

```
void consumer () {
          int count = 1;
           while (1) 5
                   int data, err;
      11 Dekter algorithm
            C2 = 1;
             while (c,) &
                     if (turn = = 1){
                             (2-0;
                              while (turn == 1);
                              C2 = 1;
                       3
                  err = front a (& a, & data)
                   iff. (err = = 0k) err = delete Q(29);
                  turn = 1;
                   Cz = 0;
                   if(err = = 0k)
                      printf (Consumed data);
                  3
```

Part

(14-) 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

Certain problems we will face if they are to be implemented

1.) Disabled on one core

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

2.) Disabled on all cores

- i.) Priority If something of high priority (urgent) occurs, kernel might need interrupt but this cannot happen as they are disabled and hence the nigh priority task will be
- ii.) Lack of interrupts: Context switching, I/O operations or any other operations will not be used by os as if requires trap in kernel.
- iii.) Clock lag: If clock relies on interrupt, it will start lagging.
- iv.) Early exit: If user app is exited due to an error or exception it will not be freed / released synchronization primitive as interrupt is disabled.

Part C

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>
```

```
#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");
```

```
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;
```

```
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
2
3 class Shared:
     def __init__(self, start=5):
           self.cokes = start
def consume(shared):
      shared.cokes -= 1
       print shared.cokes
10
def produce(shared):
      shared.cokes += 1
12
      print shared.cokes
13
14
  def loop(shared, f, mu=1):
15
      while True:
         t = random.expovariate(1.0/mu)
17
          time.sleep(t)
18
          f(shared)
19
21 | shared = Shared()
22 | fs = [consume] *2 + [produce] *2
threads = [Thread(loop, shared, f) for f in fs]
```

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,

for thread in threads: thread.join()

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()
2
       shared.mutex.wait()
3
       print shared.cokes.value()
       shared.mutex.signal()
       shared.slots.signal()
  def produce(shared):
      shared.slots.wait()
9
       shared.mutex.wait()
10
       print shared.cokes._Semaphore__value
11
       shared.mutex.signal()
       shared.cokes.signal()
```

Code:

```
#include<stdio.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);
```

```
void *consumer(void *arg);
int main()
  pthread t p[2], c[2]; //returns the thread id of thread created
  sem init(&coke, 0, 1);
  for(int i=0; i<2; i++)
      pthread create(&p[i], NULL, producer, NULL);
      pthread create(&c[i], NULL, consumer, NULL);
      pthread join(p[i], NULL);
      pthread join(c[i], NULL);
  sem destroy(&coke);
  sem destroy(&slots);
  sem destroy(&mutex);
void *producer(void *arg)
  while (1)
```

```
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(&mutex);
      printf("[CONSUMER] Number of cokes in the machine currently after
      sem post(&mutex);
      sem post(&coke);
      sleep(1);
  pthread exit(0); //to exit a thread
```

Output:

```
vinayak@vinayak-Swift-SF315-52G
vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/EndSem$ gcc Q2_CokeMachineProblem.c -o Q2_CokeMachineProblem -lpthread
 rinayak@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