4.8 Provide two programming examples in which multithreading does not provide better performance than a single-threaded solution.

1. Any kind of sequential program is not a good candidate to be threaded. An example of this is a program that calculates an individual tax return.
2. Another example is a "shell" program such as the C-shell or Korn shell. Such a program must closely monitor its own working space such as open files, environment variables, and current working directory.

4.9 Under what circumstances does a multithreaded solution using multiple kernel threads provide better performance than a single-threaded solution on a single-processor system?

When a kernel thread suffers a page fault, another kernel thread can be switched in to use the interleaving time in a useful manner. A single-threaded process, on the other hand, will not be capable of performing useful work when a page fault takes place. Therefore, in scenarios where a program might suffer from frequent page faults or has to wait for other system events, a multithreaded solution would perform better even on a single-processor system.

4.10 Which of the following components of program state are shared across threads in a multithreaded process?
a.Register values
b.Heap memory
c.Global variables
d.Stack memory

The threads of a multithreaded process share heap memory and global variables. Each thread has its separate set of register values and a separate stack.

4.13 Is it possible to have concurrency but not parallelism? Explain

Yes, It is possible to have concurrency but not parallelism.

Concurrency is a condition that exists when at least two threads are making progress. A more generalized form of parallelism that can include time-slicing as a form of virtual parallelism.'

Parallelism is a condition that arises when at least two threads are executing simultaneously". It is possible for two threads to make progress, though not at the same instant.

for example, consider a single-core processor running two threads. An operating system will normally switch back and forth between the two threads very quickly, giving the appearance of parallelism. the two threads are each taking turns executing their respective instructions on the same cpu core. Though it is not possible to have parallelism without concurrency, it is possible to have concurrency but not parallelism.

4.17

Consider the following code segment:

pid t pid;

pid = fork();

if (pid == 0) { /* child process */

fork();

thread create(. . .);

}

fork();

a.How many unique processes are created?

b.How many unique threads are created?

```
The statement pid = fork(); before the if statement creates one process.
The parent process say p creates this process. Let it be p1.

The statement fork(); in the if statement creates one process. The parent
process p creates this process. Let it be p2.
After the if statement, parent process p, process p1 and process p2 will
execute fork(); creating three new processes.
One process is created by parent process p.
One process is created by process p1.
One process is created by process p2.
Hence, 5 unique processes (p1, p2, p3, p4, p5) will be created. If the
parent process is also considered, then 6 unique processes (p, p1, p2, p3,
p4, p1, p5) will be created.
Thread creation is done in if block. Only child process p1 is executed in
the if block. Therefore, process p1 will be created one thread.
In the if block one process p2 is created using fork(). Therefore, process
p2 will also create a thread.
Hence, 2 unique threads will be created.
```

4.19 The program shown in Figure 4.23 uses the Pthreads API. What would be the output from the program at LINE C and LINE P?

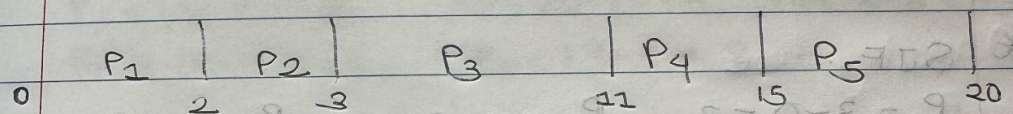
```
The output is CHILD: value = 5
The child process in the thread is forked by parent process and child process each
have its own memory space.
After forking, the parent process waits for the completion of child process.
New thread is created for child process and the runner() function is called which
set the value of the global vairlable to 5.
Thus, after execution of this line, the value displayed will be 5.
The output of LINE P in the program:
The output is PARENT: value = 0
After completing the child process, the value of the global variable present in
parent process remains 0.
Thus, after execution of this line, the value displayed will be 0.
```

5.4

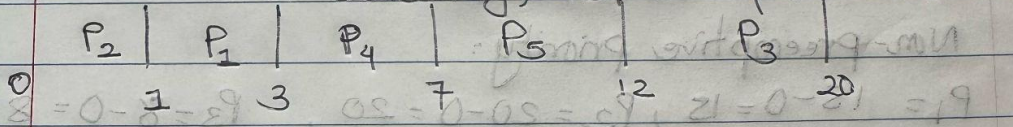
5.4

Process	Burst-time	Priority
P_1	2	2
P_2	1	1
P_3	8	4
P_4	4	2
P_5	5	3

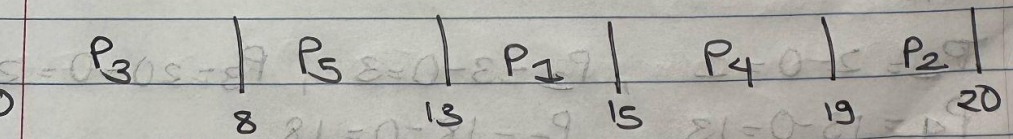
* Gantt charts for FCFS:
Here, the process comes first in the queue will be scheduled first



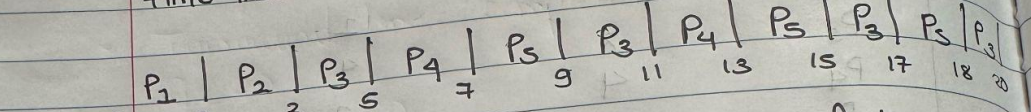
* Gantt chart for SJF Scheduling.
In SJF Scheduling, the process of least size will be 1st



* Gantt chart for Non preemptive priority
Here, highest priority process scheduled first.



* Gantt chart for Round Robin (Quantum = 2).
In RR scheduling each process will get the equal time interval of CPU until it's completed.



b) Turn around time = Finish time − Arrival time.

* FCFS:
$P_1 = 2-0 = 2$, $P_2 = 3-0 = 3$, $P_3 = 11-0 = 11$
$P_4 = 15-0 = 15$, $P_5 = 20-0 = 20$.

* SJF:
$P_1 = 3-0 = 3$, $P_2 = 1-0 = 1$, $P_3 = 20-0 = 20$
$P_4 = 7-0 = 7$, $P_5 = 12-0 = 12$.

* Non-preemptive priority:
$P_1 = 15-0 = 15$, $P_2 = 20-0 = 20$, $P_3 = 8-0 = 8$
$P_4 = 19-0 = 19$, $P_5 = 13-0 = 13$

* RR:-
$P_1 = 2-0 = 2$, $P_2 = 3-0 = 3$, $P_3 = 20-0 = 20$
$P_4 = 13-0 = 13$ $P_5 = 18-0 = 18$

c* Waiting time = Finish time - burst time :

- FCFS

$P_1 = 2-2 = 0$, $P_2 = 3-1 = 2$ $P_3 = 11-8 = 3$

$P_4 = 15-3 = 12$ $P_5 = 20-5 = 15$

- SJF

$P_1 = 3-2 = 1$ $P_2 = 1-1 = 0$, $P_3 = 20-8 = 12$

$P_4 = 7-4 = 3$ $P_5 = 12-5 = 7$.

- Non preeptive priority:

$P_1 = 15-2 = 13$ $P_2 = 20-1 = 19$, $P_3 = 8-8 = 0$

$P_4 = 19-4 = 15$, $P_5 = 13-5 = 8$

- RR.

$P_1 = 2-2 = 0$ $P_2 = 3-1 = 2$ $P_3 = 20-8 = 18$

$P_4 = 13-4 = 9$ $P_5 = 18-5 = 13$

d* Average time :

FCFS = $0+2+3+12+15/5 = 6.4$

SJF = $1+0+12+3+7/5 = 4.6$

NPP = $13+19+0+15+8/5 = 11$

RR = $0+2+18+9+13/5 = 8.4$.

∴ the least average waiting time is 4.6

SO SJF is giving least waiting time

5.5

a)

P_1	P_1	Pidle	P_2	P_3	P_2	P_3	P_4	P_4	P_2	P_3	Pidle	P_5	P_6
0	10	20	25	35	45	55	65	75	80	85	90	100	110	120

* Turnaround time: completion time - starting time.

$P_1 = 20$

$P_2 = 85 - 25 = 60$

$P_3 = 90 - 35 = 55$

$P_4 = 80 - 65 = 15$

$P_5 = 110 - 100 = 10$

$P_6 = 120 - 110 = 10$

c) Waiting time

$P_1 = 0$

$P_2 = 10 + (80 - 55) = 35$

$P_3 = 5 + 10 + (85 - 65) = 35$

$P_4 = 5$

$P_5 = 0$

$P_6 = 5$

d) CPU utilization rate

Idle time = 15

CPU utilization rate: $(105/120) \times 100 = 87.5\%$

5.15

$T_{n+1} = a * t_n + (1-a) * \tau_n$

when $a = 0$ and $t_0 = 100$ milliseconds, the eqn is

$T_{n+1} = 0 * t_n + (1-0) * 100 = 100$ milliseconds

b) $T_{n+1} = a * t_n + (1-a) * \tau_n$

when $a = 0.99$ and $\tau_0 = 10$ milliseconds,

$= 0.99 * t_n + (1 - 0.99) * 10$

$= 0.99 t_n + 0.1$