Chapter 5: CPU Scheduling Exercises

1-

t Time

P1	P2	P4	P3

0 8 12 17 26

Com	8	12	26	17
TAT	8	11	24	14
WT	0	7	15	9

the processes using non-preemptive Shortest-Job-First
xecution, and calculate the completion time, turnaround time,
s. What is the **average waiting time**?

aining-Time-First): Now schedule the same set of
(also called Shortest-Remaining-Time-First, where a newly
current one). Draw the new Gantt chart and compute each
; time. Compare the average waiting times of the two
gives a smaller average (SJF is known to minimize average

AWT =

$$\frac{0+7+15+9}{4} = 7.75$$

ith arrival times and CPU bursts:

2-

ival times and CPU bursts:

P1	P2	P4	P1	P3

0 1 5 10 17 26

Com	17	5	26	10
TAT	17	4	24	7
WT	9	0	15	2

bin (RR) with a time quantum of **4 units**. Assume
Draw the Gantt chart showing time slices and indicate
culate the waiting time and turnaround time for each
me?

AWT $= \frac{9+0+15+2}{4}$

ntum of **10 units**. How does the average waiting time
hoice of time quantum affects RR performance (for
R behave like FCFS, while a very small quantum

$6.5 = \frac{26}{4}$

Preemptive SJF (SRTF) gives a **smaller average waiting time**
because it immediately executes the job with the shortest remaining time

3-

hich one gives a smaller average (SJF is known to minimize average
et).

TAT	30	7	10
WT	6	4	7

$AW = \frac{17}{3}$

$= 5.67$

2, P3) with arrival times and CPU bursts:

U Burst Time

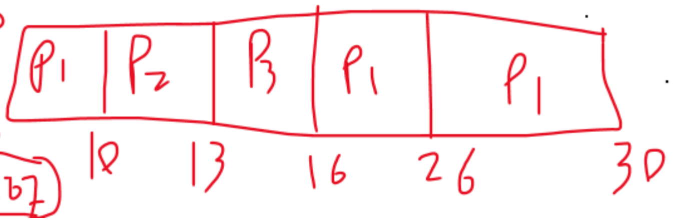
P1	P2	P3	P1	P1	P1	P1	P1

0 4 7 10 14 18 22 26 30

4-

change? Explain qualitatively how the choice of time quantum affects RR performance (for example, a very large quantum makes RR behave like FCFS, while a very small quantum increases context switches).

TAT	30	13	16
WT	6	10	13



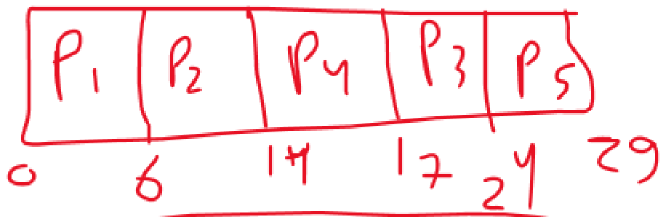
Dr. Abeer Alaql

$$\boxed{AWT = \frac{29}{3} = 9.67}$$

When we increased the time quantum to **10**, chart became very similar to **FCFS**.The short processes had to wait longer behind the long process, so the **average waiting time increased**.Therefore:
The larger the time quantum, the more Round Robin behaves like FCFS, and the average waiting time gets worse very large quantum makes RR behave like FCFS, while a very small quantum increases context switches

5-

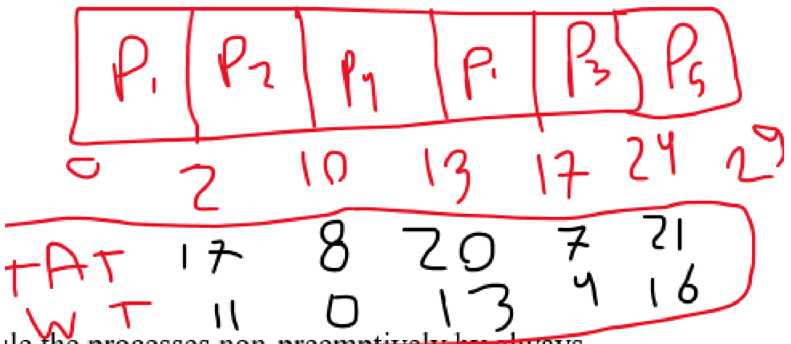
y



TAT	6	12	20	11	21
WT	0	4	13	8	16

No starvation every process eventually excecute

6-



ıle the processes non-preemptively by always

P2 waiting time decrease beacause starts immediately on arrival

P1 waiting time increases in case 2 because it was preempted and had to wait for higher priorities to complete.

P3 and p5 unchanged

P4 waiting time decrease

No starvation every process eventually excecute

7-

A low-priority process that waites a lot we can reduced priority value step by step; after waiting fixed amount of time in ready queue so priority becomes higher than processes that keep arriving That guarantees that eventually so aging converts priority starvation where waiting increases priority and eventually the waited process will execute without starvation and this would ensure every process eventually gets CPU time

Chapter 6: Synchronization Tools Exercises

1- the final value of `counter` after this interleaving counter = 4 beacause c overwrite on p

if counter++ and counter—are atomic then it will be uninterruptible and the result will be counter = 5 the result 4 is incorrect because race condition A race condition is an undesirable situation that occurs when p and c manipulate data concurrently and the outcome depends on the particular order the operations occur.

Semaphore Solution

p

```
wait(mutex);
register1 = counter;
register1 = register1 + 1;
counter = register1;
signal(mutex);
```

c

```
wait(mutex);
register2 = counter;
register2 = register2 - 1;
counter = register2;
signal(mutex);
```

that is correct because `wait(mutex)` ensures that **only one thread at a time** can enter critical section **Mutual exclusion** This prevents interleaving and Final value will always be consistent 5

Producer-Consumer with Semaphores

1-

The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to value 1.

• The empty semaphores count the number of empty buffers. The semaphore empty is initialized to the value n;

and full semaphores is initialized to the value 0 count the number of empty buffers

2-

The structure of the producer process

```
 while (true) {
...
/* produce an item in next_produced */
 ...
wait(empty);       //Decrease #Empty
 wait(mutex);         //Lock on
...
 /* add next produced to the buffer */
 ...
 signal(mutex);    //Lock off
 signal(full);      //Increase #Items
}
```

structure of the consumer process

```
while (true) {
wait(full);     //Decrease #Items
wait(mutex);     //Lock on
...
/* remove an item from buffer to next_consumed */
...
signal(mutex);      //Lock off
signal(empty);     //Increase #Empty
...
/* consume the item in next consumed */
...
}
```

3-

(i) Mutual exclusion on buffer access

- The **mutex** semaphore (initialized to 1) ensures that only one thread at a time
  Both the producer and consumer **wait(mutex)** If one process is inside the critical section, `mutex` becomes **0** Any other process trying to enter will block until **signal(mutex)** This prevents race conditions where both threads could manipulate the buffer simultaneously

(ii) Producer waits if the buffer is full

- The **empty** counting semaphore (initialized to N) is the number of available empty buffer.
- Before adding an item, the producer executes **wait(empty)**: If `empty > 0`, it decrements `empty`   If `empty = 0` (buffer full), the producer blocks
- This ensures the producer never adds to a full buffer.

(iii) Consumer waits if the buffer is empty

- The **full** counting semaphore (initialized to 0) is the number of items in the buffer.
- Before removing an item, the consumer executes **wait(full)**:
  - o If `full > 0`, it decrements `full`
  - o If `full = 0` (buffer empty), the consumer blocks
- This ensures the consumer never tries to remove from an empty buffer.

4-

a)

```
semaphore printers = 3;
```

This semaphore represents the number of available printers

b)

```
wait(printers);
print_ documents ();
signal(printers);
```

c)

- The counting semaphore `printers` starts at 3, meaning 3 printers are available.
- When a process wants to print, it calls `wait(printers)`:
    - o If `printers > 0`, it decrements the semaphore and start to print
    - o If `printers = 0`, all printers are busy, so the process blocks until one becomes available.
- Each `wait(printers)` decreases the semaphore count by 1.
- When a process finishes printing, it calls `signal(printers)`, which increments the semaphore count by 1, making one printer available again.

Chapter 7: Synchronization Examples

1-

- `wait(empty)` ensures the producer doesn't add to a full buffer (synchronization condition) producer must **wait** until the consumer consumes.
- `wait(mutex)` ensures mutual exclusion. Only one process at a time can modify the buffer structure

2-

`rw_mutex` protects **writers** from accessing the shared data when readers reads it

3-

signal(rw_mutex) never release it

- **No readers** can start reading
- **No other writers** can write
- **System deadlock for all**

4-

each philosopher uses two semaphores one for each chopsticks left and right

Deadlock occurs when:
All philosophers pick up their left fork simultaneously so all right forks are unavailable every philosopher waits forever for the right fork.

5-

A philosopher can avoid deadlock by **breaking the circular wait** condition

1- This solution put restriction that a philosopher may pick up his chopsticks only if both of them are available.

2- Number philosophers 0 to 4 Even-numbered philosophers: pick up **left fork first**, then right fork.Odd-numbered philosophers: pick up **right fork first**, then left fork.

3- We assign a number to each fork, and assume that each philosopher always picks up the fork with the lowest number first, then the next highest.

6-

- **Writer starvation it appears when** no reader should wait for other readers to finish so Since writers must wait until no readers are present, a writer may wait **forever**. if readers keep arriving writers may never get access.
- **Reader starvation**: Once a writer is ready, the writer performs its write as soon as possible so If a writer is waiting, **no new readers** are allowed to enter. if writers keep arriving, readers may be continuously blocked.

7-

None of them — this is the **deadlock**

If all 5 philosophers pick up their left chopstick at once Each philosopher holds one chopstick and No philosopher can get their right chopstick because held by their neighbor so All philosophers wait forever → **deadlock** No one gets to eat

8-

We use three semaphore

Semaphore mutex initialized to the value 1 ▪

Semaphore full initialized to the value 0

▪ Semaphore empty initialized to the value n

The mutex semaphore provides mutual exclusion for accesses to the buffer pool. • The empty and full semaphores count the number of empty and full buffers.

9-

No, it's not fair because:

- **Writer starvation**: Writers may wait indefinitely if readers keep arriving

So we have to be fair between readers and writers

10-

- Counting Semaphore has value ≥ 0 like
  - `empty` (initial = N)
  - `full` (initial = 0) Counting semaphores support **more than one access at the same time**

Binary Semaphore has value = 0 or 1 Controls **mutual exclusion** like a lock. **So** Only one process can enter the critical section. Example

- `mutex` , `rw_mutex`

11-

We start with

- Buffer: 3
- mutex = 1
- empty = 3
- full = 0

Step 1: P1 executes `wait(empty)`
empty = 2
P1 executes `wait(mutex)`
mutex = 0
P1 adds item to buffer

P1 executes `signal(mutex)`
mutex = 1
P1 executes `signal(full)`
full = 1

It ends with empty = 2 and full = 1

Step 2: P2 executes `wait(empty)`
empty = 1
P2 executes `wait(mutex)`
mutex = 0
P2 adds item to buffer
P2 executes `signal(mutex)`
mutex = 1
P2 executes `signal(full)`
full = 2

It ends with empty = 1 and full = 2

Step 3: C executes `wait(full)`
full = 1
C executes `wait(mutex)`
mutex = 0
C removes P1_item

C executes `signal(mutex)`
mutex = 1
C executes `signal(empty)`
empty = 2

It ends with empty = 2 and full = 1

Step 4: P1 executes `wait(empty)`
empty = 1
P1 executes `wait(mutex)`
mutex = 0
P1 adds item to Buffer P1 executes `signal(mutex)`
mutex = 1
P1 executes `signal(full)`
full = 2

It ends with empty = 1 and full = 2

Step 5: P2 executes `wait(empty)`
empty = 0
P2 executes `wait(mutex)`
mutex = 0
P2 adds item to Buffer
P2 executes `signal(mutex)`
mutex = 1

P2 executes `signal(full)`
full = 3

It ends with empty = 0 and full = 3

Step 6: P1 tries `wait(empty)`
empty = 0 → **P1 blocks** (buffer full

Step 7: C executes `wait(full)`
full = 2
C executes `wait(mutex)`
mutex = 0
C removes P2_item to Buffer
C executes `signal(mutex)`
mutex = 1
C executes `signal(empty)`
empty = 1

It ends with empty = 1 and full = 2

Step 8: P1 unblocks
P1 executes `wait(empty)`
empty = 0
P1 executes `wait(mutex)`
mutex = 0
P1 adds item to Buffer

P1 executes `signal(mutex)`
mutex = 1

P1 executes `signal(full)`
full = 3

It ends with empty = 0 and full = 3

how race conditions are **avoided** at each step

Every producer or consumer run `wait(mutex)` before executed and `signal(mutex)` after to prevent Mutual exclusion producers are blocked from inserting when `empty==0` `so` ensuring correct synchronization without races