Operating Systems
Quiz Scheduling – Spring 2016

2. Consider the following set of processes with arrival times and CPU burst times.

Process	Arrival time	Burst time
P1	0	7
P2	3	6
P3	4	2
P4	8	8

Draw a Gantt chart for the CPU schedule for each of the following process scheduling algorithms, and calculate the wait time for each process for each algorithm. Show your work.

- Round-robin (quantum of 3)
- Shortest job first pre-emptive
- Shortest job first non-pre-emptive

RR (q=3)

P1 (0-3) – P2 (3-6) – P1(6-9) – P3 (9-11) – P2 (11-14) – P4 (14-17) – P1 (17-18) – P4 (18 -21)

Waiting times:
P1: 0+3+8 = 11
P2: 0+5 = 5
P3: 5 = 5
P4: 6+1 = 7

SJFP

P1 (0-4) – P3(4-6) – P1(6-9) – P2(9-15) – P4(15-23)

Waiting times:
P1: 0+2 = 2
P2: 6 = 6
P3: 0 = 0
P4: 7 = 7

SJFNP

P1 (0-7) – P3(7-9) – P2(9-15) – P4(15-23)

Waiting times:
P1: 0 = 2
P2: 6 = 6
P3: 3 = 3
P4: 7 = 7

2. Examine the following correct program. Assume that the OS is using a FCFS scheduling algorithm, with the following details:
   - Actions that can move a thread from running state to wait state are any forced waits (like on a semaphore or join) and a request for I/O.
   - As soon as a thread is created, it is put on the ready queue. When post is called on a semaphore, a single one of any threads waiting on the semaphore is chosen at random and put on the ready queue. When a thread exits, all threads waiting on that action with a join are moved onto the ready queue.
   - Explain what order the marked lines (lines R, S, T, W, X, Y, Z) will get executed in and why. Show the state of the ready queue at each line.

```
int i, x = 1;
sem_t A, B, C;

void *thread_A( void *arg ) {
      sem_wait( &A );
      x = 2;            // LINE X
      sem_post( &A );
      pthread_exit( NULL );
}

void *thread_B( void *arg ) {
      sem_wait( &B );
      x = 3;            // LINE Y
      sem_post( &B );
      pthread_exit( NULL );
}

void *thread_C( void *arg ) {
      sem_wait( &C );
      x = 4;            // LINE Z
      sem_post( &C );
      pthread_exit( NULL );
}

int main() {
      pthread_t tid[3];

      sem_init( &A, 0, 1 );
      sem_init( &B, 0, 1 );
      sem_init( &C, 0, 1 );
      sem_wait( &C );
      sem_wait( &B );
      sem_wait( &A );
      pthread_create( &tid[0], NULL, thread_C, NULL );  // LINE R
      pthread_create( &tid[1], NULL, thread_A, NULL );  // LINE S
      pthread_create( &tid[2], NULL, thread_B, NULL );  // LINE T
      sem_post( &B );
      sem_post( &C );
      sem_post( &A );
      x = 5;            // LINE W

      for (i = 0; i < 3; i++ ) pthread_join( &tid[i] );
}
```

Lines Order	
R	C
S	CA
T	CAB
W	CAB
Z	AB Main
Y	B Main
X	Main

3. Examine the following correct program. Below it are 4 different versions of the code for **`threadA`**. Compare and contrast how the program will behave in each of the cases, by explaining precisely how the program will behave in each case and why.

```
sem_t A, B;

int main()
{
        pthread_t    threads[THREADS];
        int          i;

        sem_init( &A, 0, THREADS );
        sem_init( &B, 0, 1 );

        for ( i = 0; i < THREADS; i++ )
                pthread_create( &threads[i], NULL, threadA, NULL );

        for ( i = 0; i < THREADS; i++ )
                pthread_join( threads[i], NULL );

        printf( "Finished\n" );
        pthread_exit( 0 );
}
```

VERSION 1

```
void *threadA( void *ign )
{
        sem_wait( &A );

        printf( "1\n" );
        sleep( 3 );

        sem_wait( &B );

        printf( "2\n" );
        sleep( 3 );

        sem_post( &B );
        sem_post( &A );

        pthread_exit( NULL );
}
```

V1: All threads pass &A and accumulate in front of &B where they wait for their turn to pass &B one thread at the time, exiting &B and &A one at a time.

VERSION 2

```
void *threadA( void *ign )
{
        sem_wait( &A );

        printf( "1\n" );
        sleep( 3 );

             sem_post( &A );
        sem_wait( &B );

        printf( "2\n" );
        sleep( 3 );

        sem_post( &B );

        pthread_exit( NULL );
}
```

V2: All threads pass &A and exit &A. All passed threads accumulate in front of &B where they wait for their turn to pass &B one thread at the time.

VERSION 3

```
void *threadA( void *ign )
{
        sem_wait( &B );

        printf( "1\n" );
        sleep( 3 );

        sem_post( &B );
        sem_wait( &A );

        printf( "2\n" );
        sleep( 3 );

        sem_post( &A );

        pthread_exit( NULL );
}
```

V3: All threads pass one at a time &B and exit &B. After that they pass &A and exits &A in an unpredictable order.

VERSION 4

```
void *threadA( void *ign )
{
        sem_wait( &B );

        printf( "1\n" );
        sleep( 3 );

        sem_wait( &A );

        printf( "2\n" );
        sleep( 3 );

        sem_post( &A );
        sem_post( &B );

        pthread_exit( NULL );
}
```

V4: All threads pass one at a time &B and pass &A and exits &B and &A in the exact order they entered – one at a time.

4. The shortest job first (SJF) scheduling algorithm is optimal with respect to process waiting time. Explain what makes it difficult to use in reality, and is there any way to partially overcome that issue?

    - the unpredicatble arrival time
    - the unknown real time of running

    - solution use tools to predict arrival time, jopb length, or CPU averages etc.

5. In question 1, you imagined a simplification of the SJF pre-emptive scheduling algorithm, with no execution quantum. In reality, pre-emptive scheduling algorithms have an execution quantum associated with them. Please explain why it is needed.

    - No quantum then we cannot check for rescheduling during one process being run. Starvation can occur.

6. Priority scheduling algorithms in general suffer from a problem. Please name and explain that problem, and name and explain the technique used to overcome it.

    Starvation – high priority processes are run always faster than a lower one that could end up waiting for ever.

7. Consider the following process arrival, CPU, and I/O burst times given. Assume that there is a single I/O device which operates in FCFS manner.

Process	Arrival time	CPU Burst 1	I/O Burst 1	CPU Burst 2	I/O Burst 2
P1	0	5	8	6	8
P2	5	5	10	3	9
P3	6	5	12	3	10

Draw the Gantt charts for the CPU schedule and the I/O device schedule for the FCFS CPU scheduling algorithm.

CPU

P1 (0-5) – P2 (5-10) – P3 (10-15) – P1 (15-21) - P2 (23-26) – P3 (35-38)

I/O

P1 (5-13) – P2 (13-23) – P3 (23-35) – P1 (35-43) – P2 (43-52) – P3 (52-62)