perating System Exam 2024/12/12

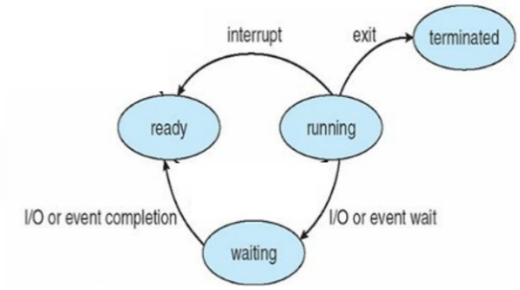
1. (8%) Please list the four situations in which CPU scheduling decisions can occur.

   1. Switches from running to waiting state

   2. Switches from running to ready state

   3. Switches from waiting to ready

   4. Terminates

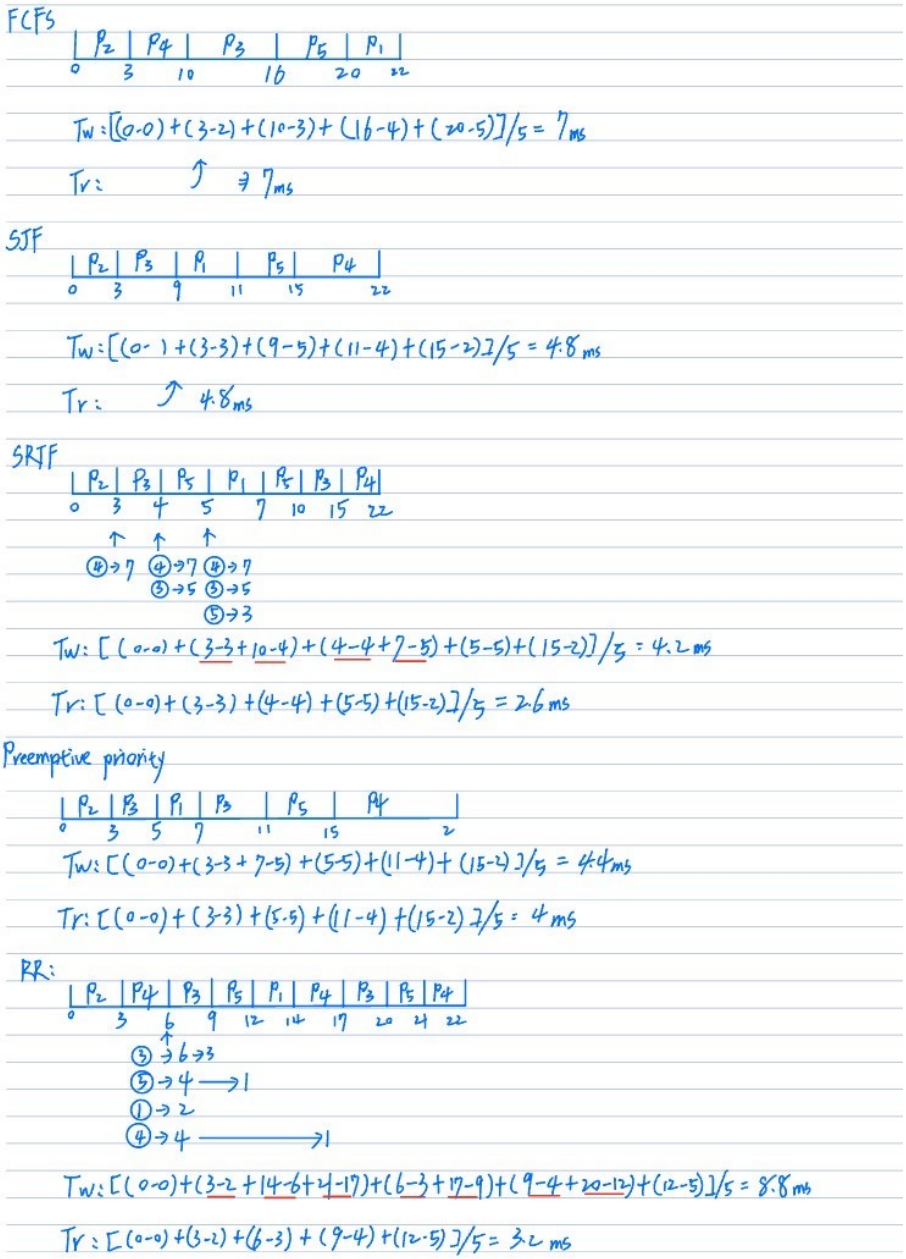


2. (35%) Consider the following set of processes, with the length of the CPU-burst time given in milliseconds:

Process	Arrival Time	Burst Time	Priority
P1	5	2	2
P2	0	3	1
P3	3	6	3
P4	2	7	5
P5	4	4	4

   a. (25%) Draw five Gantt charts illustrating the execution of these processes using FCFS, SJF(non-preemptive), SRTF, preemptive priority (a smaller priority number implies a higher priority), RR with time quantum=3. If two processes arrived in ready queue at the same time, the scheduler can schedule them in arbitrary order while follow the rules of scheduling policy.

   b. (10%) What are the average waiting time and average response time of processes for each of the scheduling algorithms in part (a)?



3. (6%) There are many methods to evaluate the performance of scheduling algorithms; please list two of them.

1.CPU utilization ->keep the CPU as busy as possible

2.Throughput ->of processes that complete their execution per time unit

3.Turnaround time->amount of time to execute a particular process

4.Waiting time->amount of time a process has been waiting in the ready queue

5.Response time->amount of time it takes from when a request was submitted until the first response is produced, not output

*** Drterministc model, queuing model, simulation, implementation**.

4. (5%) Starvation is a problem where low-priority processes may never execute due to the use of priority scheduling methods. Please give a solution to solve the problem

   Aging-> as time progresses increase the priority of the process

   Round Robin-> Timer interrupts every quantum to schedule next process

5. (6%) Please fill in the following blanks to complete the solution (Peterson's) of the critical-section (CS) problem and (10%) prove the solution satisfies all three requirements for the CS problem

   The two processes share two variables:

```
process pi:
int turn;
Boolean flag[2];
                do{
                    flag[i]=TRUE;
                    turn=j;
                    _____(a)_____;
                        critical section
                    _____(b)_____;
                        remainder section
                }while(TRUE);
```

process pi:

```
int turn;
Boolean flag[2];
                do {
                    flag[i] = TRUE;
                    turn = j;
                    while (flag[j] && turn == j);
                        critical section
                    flag[i] = FALSE;
                        remainder section
                } while (TRUE);
```

- Mutual Exclusion
  - Pi and Pj could not have successfully executed their while statements at the same time, since the value of turn can be either 0 or 1
- Progress and bounded-waiting
  - Pi can be prevented from entering the critical section (CS) only if it is stuck in the while loop
    - If Pj is not ready to enter CS, flag[j]=false, Pi can enter CS
    - If Pj has set flag[j] to true
      - turn =i , Pi can enter CS
      - turn =j , Pj can enter CS
        - Pj set flag[j] to false, Pi can enter CS
        - Pj reset flag[j] to true, Pj must also set turn to I, Pi can enter CS

1. 互斥性存在
2. 程序的要求能被滿足
3. 有限的等待要求

- 證明第一點:兩個程序無法同時執行 while 迴圈，因為 turn 必定在其中一個 process 手上。
- 證明第二、三點:當條件 flag[j]==true 與 turn==j 成立時，Pi 才會被 while 迴圈擋住，如果 Pj 不會進入 critical section，那麼 flag[j]==false，Pi 就能進入。並且，當 Pj 要離開 critical section 時，會設置 flag[j]為 false，以允許 Pi 進入，如果 Pj 重新設置 flag[j]為 true，那麼它也應設置 turn 為 i。因此由於行程 Pi 執行 while 語句時並不改變變量 turn 的值，所以 Pi 會進入 critical section，而且 Pi 在 Pj 進入臨界區後最多一次就能進入。

6. (6%) Consider three concurrently running processes: P1 with statement S1, P2 with a statement S2, P3 with a statement S3. Suppose we require that S2 be executed only after S1 has been completed and S1 be executed only after S3 has been completed, i.e. S3->S1->S2. Assume they share two common semaphores S and Q which are initialized to 0. Please write down the statements that use wait() and signal() operations to control the executing sequence of P1, P2, P3.

P1	P2	P3
Wait(Q) S1 Signal (S)	Wait (S) S2	S3 Signal(Q)

S、Q 可相反

7. (6%) There are two semaphores used in the reader program for solving Reader-

Writer problem as shown in the following figure, please explain why the two semaphores are needed.

```
do{
        wait(mutex);
        read_count++;
        if(read_count ==1)
                wait(rw_mutex);
        signal(mutex);
                //reading is performed
        wait(mutex);
        read_count--;
        if(read_count==0)
                signal(rw_mutex);
        signal(mutex);
  }while(true);
```

mutex：

用來確保同一時間只有一個讀者能夠更新或檢查 read_count 變數。

rw_mutex：

用來協調讀者和寫者的訪問權限。當第一個讀者進入時，發出 rw_mutex 信號，防止寫者進入。當最後一個讀者離開時，它再歸還 rw_mutex 信號，允許寫者進入。

8. (6%) What will happen if semaphores are misused in the critical section problem?

- Deadlock: Occurs when two or more processes are stuck waiting indefinitely for events that only the other waiting processes can trigger. As a result, none of the processes can proceed.
- Starvation: Happens when a process is indefinitely blocked because it remains in the semaphore queue without ever being scheduled to execute.
- Priority Inversion: A scheduling issue where a lower-priority process holds a lock needed by a higher-priority process, causing the higher-priority process to be delayed unnecessarily.

9. (12%) Please fill in the blanks in the following program to solve dining-philosophers problem (monitor solution)

```
monitor dp { enum {THINKING, EATING, HUNGRY}state[5];
             condition self[5];
    void pickup( int i ) {
          state[i]=___(a)____;
          test(i);
          if (state[i]!=_____(b)____)
               self[i].wait(); }
    void putdown (int i){
          state[i]=THINKING;
          ____test((i+1)%5);
          _____(c)_____; }
    void test( int i ) {
          if (_____________________
            _____(d)_____________) {
               state[i]=EATING;
               self[i].signal(); }   }
    intialization_code() {
          for (int i=0; i<5; i++)
               state[i]=THINKING; }   }
```

```
monitor DiningPhilosophers
  {
    enum { THINKING, HUNGRY, EATING} state [5] ;
    condition self [5];

    void pickup (int i) {
         state[i] = HUNGRY;
         test(i);
         if (state[i] != EATING) self [i].wait;
    }

     void putdown (int i) {
         state[i] = THINKING;
              // test left and right neighbors
         test((i + 4) % 5);
         test((i + 1) % 5);
     }
```

```
    void test (int i) {
         if ( (state[(i + 4) % 5] != EATING) &&
         (state[i] == HUNGRY) &&
         (state[(i + 1) % 5] != EATING) ) {
              state[i] = EATING ;
               self[i].signal () ;
          }
    }

     initialization_code() {
          for (int i = 0; i < 5; i++)
          state[i] = THINKING;
    }
}
```