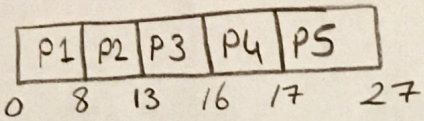
Ayça Avcı
53961

12/04/2020

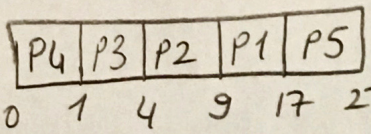
COMP 304

Assignment #2

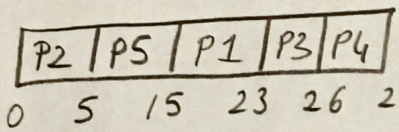
1) a)



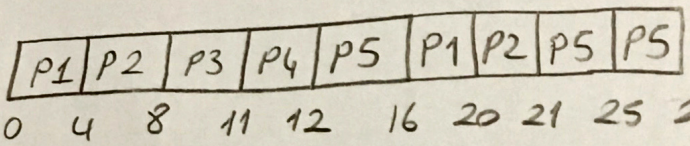
→ FCFS



→ SJF (non-preemptive)



→ Non-preemptive priority



→ RR

b) FSFS ⇒ P1: 0 ms.
P2: 8 ms.
P3: 13 ms.
P4: 16 ms.
P5: 17 m.s

$$\text{average waiting time} = \frac{0+8+13+16+17}{5} = 10,8 \text{ ms}$$

SJF ⇒ P_1: 9 ms.
P_2: 4 ms.
P_3: 1 ms.
P_4: 0 ms.
P_5: 17 ms.

$$\text{average waiting time} = \frac{9+4+1+0+17}{5} = 6,2 \text{ ms.}$$

Non-preemptive priority ⇒ P_1: 15 ms.
P_2: 0 ms.
P_3: 23 ms.
P_4: 26 ms.
P_5: 5 ms.

$$\Rightarrow \underbrace{\frac{15+0+23+26+5}{5} = 13,8 \text{ ms.}}_{\text{average waiting time}}$$

RR ⇒ P1 : 12 ms.
P2 : 16 ms.
P3 : 8 ms.
P4 : 11 ms.
P5 : 17 ms.

average waiting time $= \frac{12+16+8+11+17}{5} = 12{,}8$ ms.

Hence, SJF scheduling has minimum average waiting time with 6,2 ms.

c) FCFS ⇒ $\frac{\overbrace{0+8+13+16+17}^{\text{waiting times}} + \overbrace{27}^{\text{total burst time}}}{5} = 16{,}2$ ms.

SJF ⇒ $\frac{9+4+1+0+17+27}{5} = 11{,}6$ ms.

Non-preemptive priority ⇒ $\frac{15+0+23+26+5+27}{5} = 19{,}2$ ms.

RR ⇒ $\frac{12+16+8+11+17+27}{5} = 18{,}2$ ms.

2) CPU utilization $= 100 \times \frac{\text{Total burst time}}{(\text{Total burst time} + \text{context switch time})}$

FCFS ⇒ $100 \times \frac{27}{27+(0.5\times4)} = 93{,}10$

SJF ⇒ $100 \times \frac{27}{27+(0.5\times4)} = 93{,}10$

Non-preemptive priority ⇒ $100 \times \frac{27}{27+(0.5\times4)} = 93{,}10$

RR ⇒ $100 \times \frac{27}{27+(0.5\times7)} = 88{,}52$

3) a) There is a race condition in the given code. Output will be depend on order of the execution of the processes. available_connections is a shared variable and it can be written by threads at the same time whenever disconnect() and connect() methods are called. Race condition might occur when more than one process reach the available_connections variable in the memory and try to change it instantenously.

b)
```
#define MAX_CONNECTIONS 5000
#include <pthread.h>
int available_connections = MAX_CONNECTIONS;
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_init (&mut, NULL);

int connect() {
    pthread_mutex_lock(&mut);
    if (available_connections < 1) {
        pthread_mutex_unlock(&mut);
        return -1;
    } else {
        available_connections--;
    }
    pthread_mutex_unlock(&mut);
    return 0;
}
int disconnect() {
    pthread_mutex_lock(&mut);
    available_connections++;
    pthread_mutex_unlock(&mut);
    return 0;
}
```

c) Using atomic integer will prevent the race condition, but it won't be enough. Still, while one of the thread inside one of the function, another thread can change the available_connection variable before other thread returning any result. This situation might affect the result of the thread. To solve this situation, increment and decrement methods from the atomic library should be used. (atomic_inc(), atomic_dec())

4) In the below code, I used binary semaphore with initial value 1. If there is no available for a party, party will wait until necessary number of rooms are become available.

```
int available_room = M;
semaphore S = 1;

void book_room(int num_person) {
    wait(S);
    if (available_room < ceil(num_person/4.0)) {   // double casting

    } else {
        available -= ceil(num_person/4.0);
    }
    signal(S);
    return;
}
void release_room(int num_person) {
    wait(S);
    available_room += ceil(num_person/4.0);
    signal(S);
    return;
}
```

5)

```
monitor hospital {
    int available_doctor = 4;
    int patiens[M];
    int num_patients = 0;
    condition c;

    void request_doctor(int priority) {
        if (available_doctor > 0){
            available_doctor--;
            return;
        }
        patiens[num_patiens++] = priority;
        sort(patients); //in descending order
        while (available_doctor == 0 || patiens[0] != priority){
            c.wait();
            patients[0] = patiens[num_patients - 1];
            num_patients--;
            sort(patients);
            available_doctor--;
        }
    }

    void release_doctor() {
        available_doctor++;
        c.broadcast();
    }
}
```

6) a) Need = Max. − Allocation

	Need			
	A	B	C	D
P1	0	6	4	2
P2	0	0	2	0
P3	1	0	0	2
P4	0	0	0	0
P5	0	7	5	0

b) Yes, system is in a safe state. After applying Banker's Safety Algorithm, one of the safe state sequence is P2, P3, P4, P5, P1.

For P1 ⇒ (0 6 4 2) ≰ (1 5 20) ✗ → work.

For P2 ⇒ (0 0 2 0) ≤ (1 5 20), so, work = (1 5 20) + (0 6 3 2) ✓ (update the work; allocation of P2)
= (1 11 5 2)

For P3 ⇒ (1 0 0 2) ≤ (1 11 5 2), so, work = (2 14 10 6) ✓

For P4 ⇒ (0 0 0 0) ≤ (2 14 10 6), so, work = (2 14 11 8) ✓

For P5 ⇒ (0 7 5 0) ≤ (2 14 11 8), so, work = (3 14 11 8) ✓

For P1 ⇒ (0 6 4 2) ≤ (3 14 11 8) ✓

c) $Request_{P_5}$ = (0 3 3 0). Below conditions must hold.

① $Request_{P_5} \leq Need_{P_5}$ ⇒ (0 3 3 0) ≤ (0 7 5 0) ✓
True.

② $Request_{P_5} \leq Available$ ⇒ (0 3 3 0) ≰ (1 5 0 2) ✗
False.

Since $Request_{P_5} \leq Available$ does not hold, P5 should not be granted immediately.