Sistemas Operativos – P4 – Hugo Paixão a64514 – Amine Berqia e José Bastos

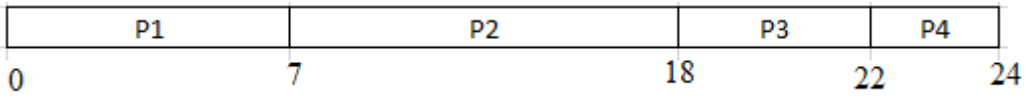
Exercício 1.

1) O parâmetro quantum permite quantificar o tempo que cada processo pode utilizar na sua execução. O quantum funciona como um quantificador de janelas temporárias e os processos que estão a ser executados num dado instante têm de obedecer ao tempo estabelecido pelo parâmetro. Pode-se constatar que o aumento do tempo, permite que um processo tenha mais tempo para a sua execução e possivelmente poderá apenas ser executado uma única vez, o que no caso de uma janela mais pequena precisaria de mais tempo.

2) O algoritmo SJF consiste na execução do programa que demora menos tempo a ser executado primeiro. Podemos dizer que o algoritmo SJF é um caso particular do algoritmo de escalonamento com prioridades, pois no SJF é como se a prioridade fosse atribuída conforme o tempo de execução de um programa, o que tem menor tempo de execução é o programa com a prioridade mais elevada e o que tem o maior tempo de execução é o que tem a prioridade mais baixa.

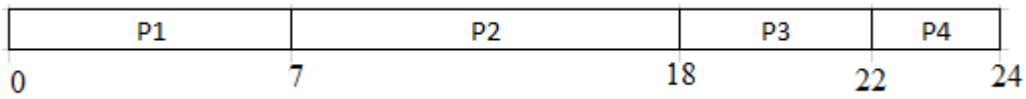
3)

FCFS – sem preempção



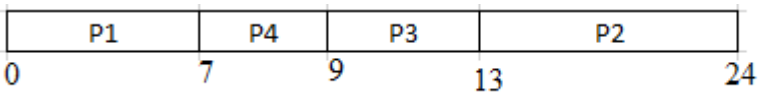
	P1	P2	P3	P4	Avg	Eficiencia
WT	**0** = 7-7	**6** = 17-11	**14** =18-4	**16**=18-2	36/4	4/24= 0.166…
TAT	**7**= 7-0	**17**=18-1	**18**=22-4	**18**=24-6	60/4	

SJF – sem preempção



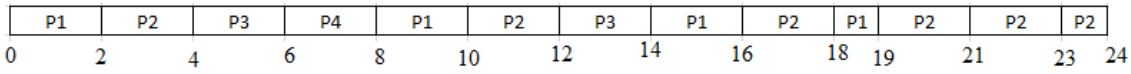
	P1	P2	P3	P4	Avg	Eficiencia
WT	**13**	**24**	**6**	**2**	45/4	4/24= 0.166…
TAT	**7**= 7-0	**17**=18-1	**18**=22-4	**18**=24-6	60/4	

SJF – com preempção



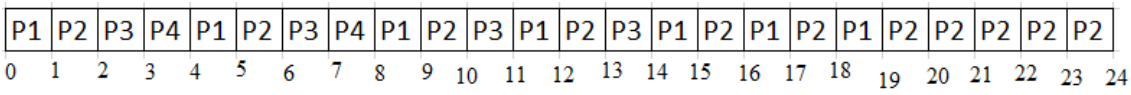
	P1	P2	P3	P4	Avg	Eficiencia
WT	**13**	**24**	**6**	**2**	45/4	4/24= 0.166…
TAT	**7**= 7-0	**17**=18-1	**18**=22-4	**18**=24-6	60/4	

RR (quantum = 2ms)



	P1	P2	P3	P4	Avg	Eficiencia
WT	**12** =19-7	**12** = 23-11	**8** = 12-4	**4** = 6- 2	36/4	4/24= 0.166…
TAT	**19**= 19-0	**23**=24-1	**12**=16-4	**6**=12-6	60/4	

RR (quantum = 1ms)



	P1	P2	P3	P4	Avg	Eficiencia
WT	**12** =19-7	**12** = 23-11	**9** = 13-4	**5** = 7-2	38/4	4/24= 0.166…
TAT	**19**= 19-0	**23**=24-1	**13**=17-4	**7**=13-6	62/4	

Exercício 2.

1) O jantar dos filósofos é um problema de sincronização de processos usado para avaliar situações em que há necessidade de alocar vários recursos para vários processos, que se baseia em 5 filósofos silenciosos sentados numa mesa-redonda com tigelas de

espaguete. Garfos são colocados entre cada par de filósofos adjacentes. Cada filósofo deve pensar e comer alternadamente. No entanto, um filósofo só pode comer espaguete quando tem garfos direito e esquerdo. Cada garfo pode ser segurado por apenas um filósofo e, portanto, um filósofo só pode usar o garfo se não estiver sendo usado por outro filósofo. Depois que um filósofo termina de comer, ele precisa abaixar os dois garfos para que fiquem disponíveis para outras pessoas. Um filósofo só pode pegar o garfo à direita ou à esquerda quando ficam disponíveis e não pode começar a comer antes de pegar os dois garfos. Comer não é limitado pelas quantidades restantes de espaguete ou espaço do estômago; uma oferta infinita e uma demanda infinita são assumidas. O problema é como projetar uma disciplina de comportamento (um algoritmo concorrente) de forma que nenhum filósofo morra de fome; ou seja, cada um pode continuar para sempre alternando entre comer e pensar, assumindo que nenhum filósofo pode saber quando os outros podem querer comer ou pensar.

2)

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

#define N 5
#define THINKING 2
#define HUNGRY 1
#define EATING 0
#define LEFT (phnum + 4) % N
#define RIGHT (phnum + 1) % N

int state[N];
int phil[N] = { 0, 1, 2, 3, 4 };

sem_t sem1;
sem_t sem2[N];

void test(int phnum) {

    if (state[phnum] == HUNGRY && state[LEFT] != EATING && state[RIGHT] !
= EATING) {

        state[phnum] = EATING;

        sleep(2);

        printf("Philosopher %d takes fork %d and %d\n", phnum + 1, LEFT +
 1, phnum + 1);

        printf("Philosopher %d is Eating\n", phnum + 1);
```

```
        sem_post(&sem2[phnum]);
    }
}


void take_fork(int phnum) {

    sem_wait(&sem1);

    state[phnum] = HUNGRY;

    printf("Philosopher %d is Hungry\n", phnum + 1);

    test(phnum);

    sem_post(&sem1);

    sem_wait(&sem2[phnum]);

    sleep(1);
}


void put_fork(int phnum) {

    sem_wait(&sem1);

    state[phnum] = THINKING;

    printf("Philosopher %d putting fork %d and %d down\n", phnum + 1, LEF
T + 1, phnum + 1);
    printf("Philosopher %d is thinking\n", phnum + 1);

    test(LEFT);
    test(RIGHT);

    sem_post(&sem1);
}

void* philospher(void* num) {

    while (1) {

        int* i = num;

        sleep(1);

        take_fork(*i);
```

```
        sleep(0);

        put_fork(*i);
    }
}

int main() {

    pthread_t thread[N];

    for(int i = 0; i < N; i++)
        printf("Philosopher %d joined the table.\n", i+1);

    sem_init(&sem1, 0, 1);

    for (int i = 0; i < N; i++)
        sem_init(&sem2[i], 0, 0);

    for (int i = 0; i < N; i++) {
        pthread_create(&thread[i], NULL, philospher, &phil[i]);

        printf("Philosopher %d is thinking\n", i + 1);
    }

    for (int i = 0; i < N; i++)
        pthread_join(thread[i], NULL);
}
```