Lista 02 - Sistemas Operacionais

September 15, 2020

Capítulo 05

Questão 5.2)

Como a ordem de chegada dos processos pode influenciar a ordem de escalonamento dos mesmos, temos que:

$$\#schedules = n!$$

Questão 5.3)

Conta: Tempo de CPU / Tempo para caminhar por todos os processos (com overhead)

a) $11/(10 * 1.1 + 1.1) = 11/12.1 \approx 0.91 = 91\%$.

b) $20/(10*1.1 + 10.1) = 20/21.1 \approx 0.95 = 95\%$.

Questão 5.9)

- First-come, first-served: Não sofre de inanição (starvation), pois todos os processos irão ser executados sempre na ordem de chegada (primeiro a chegar será o primeiro a ser executado).
- Shortest job first: Sofre de inanição (starvation), pois caso tenha um processo muito longo é capaz que ele nunca seja executado devido à sua baixa prioridade.
- Round robin: Não sofre de inanição (starvation), pois a troca de contexto ocorre a cada "quantum" garantindo que após algum tempo todos os processos já terão sido executados.
- Priority: Sofre de inanição (starvation), pois caso um processo tenha baixa prioridade ele pode nunca ser executado.

Questão 5.10)

Esse algoritmo irá favorecer processos I/O-bound pois eles necessitam de menos uso da CPU, gastando mais tempo esperando uma entrada e saída do que computando a mesma. Não haverá a inanição de processos CPU-bound pois após um curto período de tempo os processos I/O-bound irão deixar a CPU para realizar novamente o I/O.

Questão 5.12)

1. O escalonador RR (Round-Robin) iria escalonar o mesmo processo 2 vezes, dando assim maior prioridade para o mesmo.

2. Uma vantagem é que estariamos implementando um sistema de prioridades no nosso escalonador, o que pode ser benéfico; em contrapartida, processos que necessitam de pouco tempo de CPU estariam sendo prejudicados.

3. Podemos adaptar o algoritmo permitindo mais de um valor de "quantum" possível, atribuindo o maior valor ao processo que desejamos que tenha maior prioridade.

Questão 5.13)

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	3
P_4	1	4
P_5	5	2

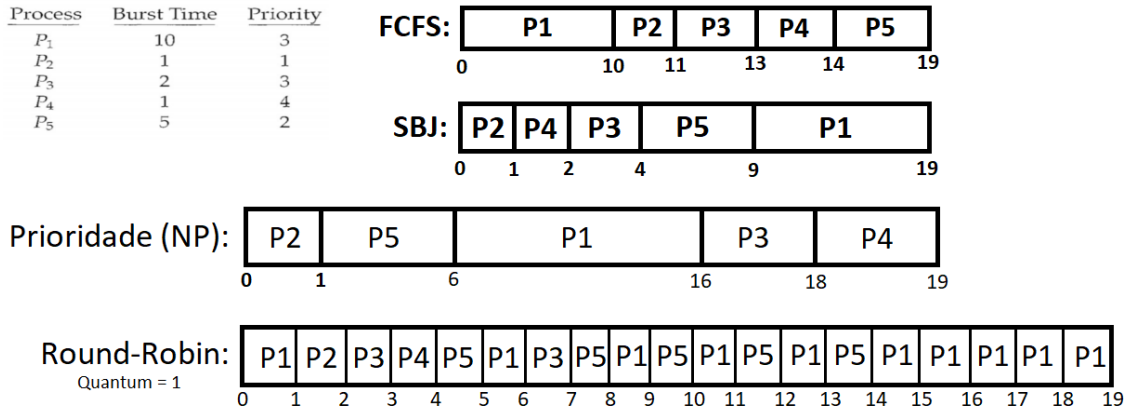

Figure 1: Grantt Charts

4. O escalonador que proporciona o menor tempo de espera médio é o SJF (Shortest Job First).

Questão 5.14)

- Prioridade P_1: $40/2 + 60 = 80$
- Prioridade P_2: $18/2 + 60 = 69$
- Prioridade P_3: $10/2 + 60 = 65$

O escalonador diminui a prioridade relativa dos processos CPU-bound.

	P1	P2	P3	P4	P5
FCFS	10	11	13	14	19
SBJ	19	1	4	2	9
Prioridade Não Preemptivo	16	1	18	19	6
Round-Robin Quantum = 1	19	2	7	4	14

Figure 2: Turnaround

	P1	P2	P3	P4	P5
FCFS	0	10	11	13	14
SBJ	9	0	2	1	4
Prioridade Não Preemptivo	6	0	16	18	1
Round-Robin Quantum = 1	9	1	5	3	9

Figure 3: Wanting Time

Questão 5.15)

1. Para maximizar o uso de CPU podemos diminuir o overhead associado à troca de contexto, tornando-a menos frequente. Porém, isso acarreta no aumento do tempo de resposta.

2. Para minimizarmos o tempo médio de turnaround devemos executar os processos menores primeiro. Contudo, podemos ter processos em inanição (starvation), aumentando o tempo de espera destes.

3. Para maximizarmos o uso de dispositivos I/O devemos escalonar primeiro os processos I/O-bound, visto que seu gasto de CPU é menor. Contudo, para maximizar o uso de CPU, devemos rodar longos processos CPU-bound a fim de evitar overheads gerados por troca de contexto, não garantindo assim a maximização do uso de dispositivos I/O.

Capítulo 06

Questão 6.9)

Podemos utilizar um semáforo para garantir essa limitação no servidor. Para tal, iremos inicializar o valor de "value" com N. Com isso, após N waits o semáforo não permitirá mais conexões até que um signal seja emitido, retratando a liberação de um socket no servidor.

Questão 6.10)

Esse tipo de mecanismo de sincronização não é usado em sistemas de uniprocessador, visto que não teremos outro processo para satisfazer a condição de liberar um lock; já em sistemas de multiprocessadores, temos processos/threads rodando concorrentemente, fazendo com que um libere o lock para outro.

Questão 6.11)

Se as operações wait e signal não forem atômicas, i.e executadas em 1 ciclo de instrução, dois processos poderão modificar o valor do semáforo, decrementando ou incrementando ele respectivamente. Sendo assim, um semáforo não irá garantir mais a exclusão mútua.

Questão 6.12)

```
int lock  = 0;
int value = 1; // mutexes
procList p = initList(); // garantindo busy waiting mínimo

wait() {
    while (TestAndSet(&lock)); // em espera
    if (value == 0) // não temos mais "espaço"
        p.add(processo); sleep(processo)
    else // decrementamos quantos processos podemos ter e liberamos o lock
        value--; lock  = 0;
}

signal() {
    while (TestAndSet(&lock)); // em espera
    if (value != 0 && !p.empty()) // temos processos dormindo
        wake(p.first())

    value++;
    lock = 0; // liberando o lock
}
```

Questão 6.17)

Se implementarmos primitivas de sincronização em sistemas de uniprocessador através da desabilitação de interrupções faremos com que o sistema execute apenas o processo do usuário, parando assim totalmente o S.O por não conseguir realizar trocas de contexto.