

1.a) Nota: Ter atenção que os fadões de saída gerados não são totalmente determinísticos. Ou seja, quando dois semáforos estão a '1' não há uma ordem direta sobre quem vai começar primeiro.

### I) Operações associadas a Cade Semáforo:

Processo 1	Processo 2	Processo 3
Down(1)	Down(2)	Down(3)
Print('A')	Print('B')	Print('C')
UP(3)		UP(1) UP(2)

Sem1      Sem2      Sem3

Configuração inicial: 0      1      1 → Como referido acima, este valor não garante o fadão de saída desejado, mas é o maior valor possível

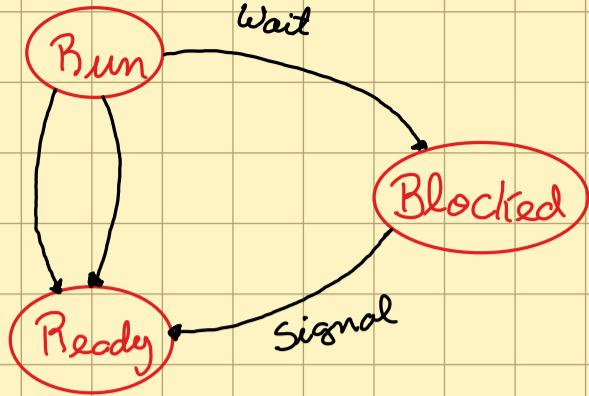
0      0      1 → B  
0      0      0 → C  
0      1      0 → A

...

II) Os outros fadões com a mesma configuração, poderiam ser "CABCABCAB" ou "CBACBACBA".

1.b) A operação DOWN do Semáforo é uma operação atómica e testa se pode decrementar este. Caso o seu valor seja 0 e não possa decrementar, o processo ficará bloqueado à espera que a acção possa ser executada, ou seja, o semáforo esteja com o valor 1. A operação equivalente com Threads é o WAIT, no entanto este não testa nenhuma condição e coloca logo o programa em Blocked até que outro Thread o sinalize, dai nos casos ser sempre usado um IF ou While com o wait, de forma a estes testarem a condição e garantirem também que o thread não vai de Blocked com as condições certas.

A operação UP dos semáforos também bloqueia o Processo se o semáforo já estiver a '1'. O SIGNAL também tem um sentido semelhante, no entanto, a maior diferença entre os dois é que o UP guarda o valor do semáforo incrementado, mesmo que nenhum outro Processo esteja à espera. No entanto, o valor do SIGNAL é perdido se ninguém estiver à espera com um WAIT nesse momento, também é possível que o Thread criado que esteja à espera seja doradoamente sinalizado e o que necessitava do SIGNAL não seja acordado.



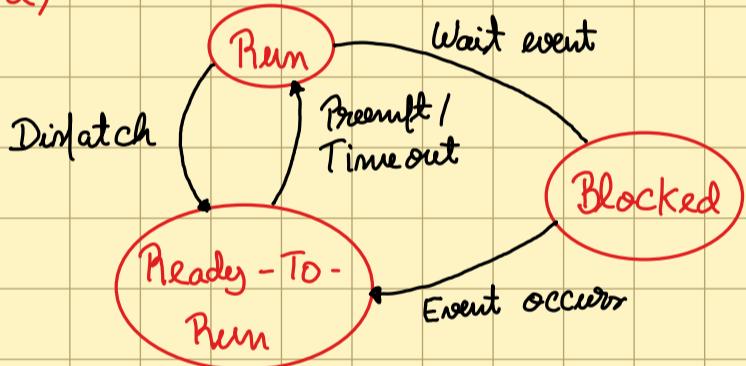
- Up e Down tanto podem estar entre Run → Blocked ou Blocked → Signal, visto que ambas operações podem Bloquear e libertar o Processo.

1. c)

Thread 1	Thread 2	Thread 3
lock (me);	lock (me);	lock (me);
while (!T1)	while (!T2)	while (!T3)
Wait;	Wait;	Wait;
...	...	...
Signal (T3);	unlock (me);	Signal (T1);
unlock (me);		Signal (T2);
		unlock (me);

→ Usar os mutau fará evitar Race Conditions

2. a)



Dispatch: O Processo com maior prioridade é selecionado para correr

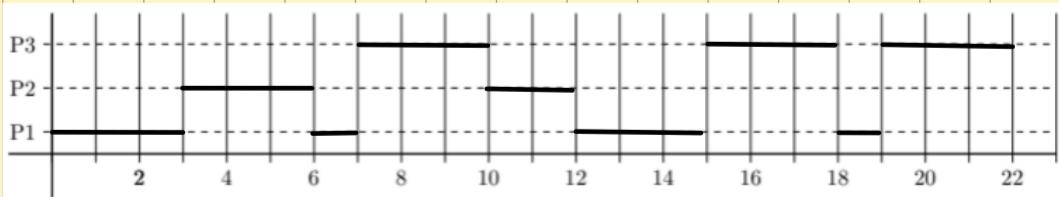
Wait: O Processo está à espera de um evento (Ex: I/O)

Event: O Processo já recebeu o que estava à espera e pode ser corrido

Preempt: Retirar o Processo de execução, pois existe um com maior prioridade

Timeout: O Processo foi retirado de execução porque excedeu o seu slot de tempo

2.5) Round robin (RR) → O Processo mais antigo em Ready é o que é selecionado para rodar. Usa o mecanismo de Preempt associado a um Time Quantum ou Slot temporal, para remover freguesias de Run → Ready quando estes gastam o seu slot temporal.



$t = 0$ :  $P_1 \rightarrow \text{Ready}$ ,  $P_1 \rightarrow \text{Run}$

$t = 2$ :  $P_2 \rightarrow \text{Ready}$

$t = 3$ :  $P_1 \rightarrow \text{Timeout} \rightarrow \text{Ready}$ ,  $P_2 \rightarrow \text{Run}$

$t = 4$ :  $P_3 \rightarrow \text{Ready}$

$t = 6$ :  $P_2 \rightarrow \text{Timeout} \rightarrow \text{Ready}$ ,  $P_1 \rightarrow \text{Run}$ ,  $P_3 \rightarrow \text{Ready}$  →  $P_1$  está à mais tempo em Ready

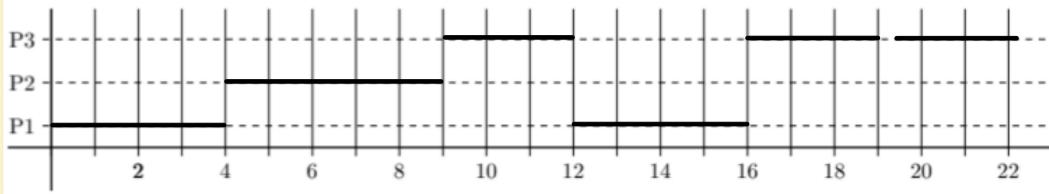
$t = 7$ :  $P_1 \rightarrow \text{Blocked}$ ,  $P_3 \rightarrow \text{Run}$ ,  $P_2 \rightarrow \text{Ready}$

$t = 7.5$ :  $P_1 \rightarrow \text{Ready}$

...

$t = 12$ :  $P_2 \rightarrow \text{Termina}$

2. c)



Tempos de Turnaround:

$$P_1: 16 - 0 = 16 \text{ ut}$$

$$P_2: 9 - 4 = 5 \text{ ut}$$

$$P_3: 22.2 - 9 = 13.2 \text{ ut}$$

3. a) Uma arquitetura com organização de memória real permite alocaçõe um segmento de memória a cada Processo, separando este da memória física. Para isso, introduz um dispositivo em hardware que permite, em tempo real, traduzir endereços lógicos, gerados pelo CPU, ou seja, endereços com base na memória alocada ao Processo, para endereços físicos, onde essa memória está localizada. Antes de cada acesso à memória, a MMU verifica se o endereço fornecido está dentro da gama de endereços fornecida ao Processo, se estiver, rompe o endereço base e interage com a memória, se não, gera uma exceção.

As Partições filhas não são blocos de memória do Processo Pré-dividido em blocos imutáveis. Quando um Processo entra em execução, tem de arranjar um dos slots de memória e caso este seja menor que o slot, o espaço extra é desperdiçado.

No caso da Partições Variáveis, a memória toda é um slot, depois, com base no espaço necessário, fraciona-se a parte livre, de forma a alocaçõe o processo em execução. Quando o Processo acaba, a memória alocada deixa de ser usada e todo o espaço é defragmentado. Este método permite utilizar a memória de uma maneira mais eficiente.

3. b)

I) O registo base contém o endereço inicial do Processo na memória física e o registo limite contém o tamanho de memória alocada. Quando um Processo tenta aceder a um endereço < registo base ou > registo base + limite, a exceção Segmentation Fault é gerada.

II) Estes registos não alterados na ação Dispatch, quando o Processo passa Ready → Run. Nessa etapa, ele consulta a tabela de Processos para saber o que atribuir aos registos.

III) O CPU envia para a MMU um offset de acesso à memória, a MMU soma o offset ao registo base e verifica se o endereço resultante se encontra dentro do segmento de memória alocada ao Processo, por este não menor que o registo base + limite. Se for, então o acesso procede e as operações sobre os dados são realizadas, se não, a exceção Segmentation Fault é gerada.

3.c) Inicia:	A entra:	B entra:	C entra:	B Sai:	D entra:
10000 - OS	10000 - OS	10000 - OS	10000 - OS	10000 - OS	10000 - OS
190000 - livre	10000 - A  180000 - livre	10000 - A  40000 - B  140000 - livre	10000 - A  40000 - B  20000 - C  120000 - livre	10000 - A  40000 - B  20000 - C  120000 - livre	10000 - A  40000 - B  20000 - C  120000 - livre

A Sai:

10000 - OS
50000 - livre
20000 - C
20000 - D
100000 - livre

4.a) Deadlock Prevention defende que compete a quem faz a aplicação garantir que não haja situações de Deadlock ou estados imseguros.

No Caso de Deadlock Avoidance, presente no sistema apresentado, existe uma aplicação / gestor externo que faz a atribuição dos recursos automaticamente, garantindo que não haja situações de Deadlock ou estados imseguros.

4.b)

Recursos Disponíveis	Atribuições
(1, 1, 1)	$P_4 \rightarrow (1, 0, 0) \rightarrow$ Termine
(2, 2, 1)	$P_1 \rightarrow (2, 0, 0) \rightarrow$ Termine
(5, 3, 3)	$P_2, P_3 \rightarrow$ Podem correr livremente (estado Safe)

4.b) Vamos avaliar o estado do Sistema por Simulação

Recursos Disponíveis	Atribuições
(1, 1, 1)	$P_3 \rightarrow (0, 1, 0) \rightarrow$ Fica com (3, 0, 0) e irá adquirir
(1, 0, 1)	$P_4 \rightarrow (1, 0, 0) \rightarrow$ Termine
(2, 1, 1)	$P_1 \rightarrow (2, 0, 0) \rightarrow$ Termine
(5, 2, 3)	$P_2, P_3 \rightarrow$ Podem correr livremente (estado Safe)