SO Teórico:

Exame modelo:

1 a) i) fflush → força o printf

. No início de cada processo é necessário que os semáforos estejam em zero poderem avançar. Inicialmente cada processo faz down de cada semáforo!

sem-down precisa que (no mínimo) o semáforo seja inicializado a 1

Para ser produzido um 'B' o sem2 tem que estar a 1

Como o processo 2 não faz up de nenhum outro semáforo, então 1 necessário ter cada semáforo inicializado a 1 → o sem^

Resposta: sem1 sem2 sem3
1 1 0

1c) A B C A B C A B C ou A C B A C B A C B

b) . Perda de informação do signal em relação ao up

Se o signal é emitido e não há nenhuma thread em wait, o signal perde-se

Numa sincronização com signal e wait, o wait tem que acabar primeiro, necessariamente

↳ com down e up, a ordem é irrelevante

No down, se o semáforo tiver sido levantado primeiramente, passa, quando que o wait bloqueia sempre

Nota: O número de ups tem que coincidir com o número de downs

c) . Onde existe o sem_down, substituir por:

→ lock mx
while cond:
wait (vcond, mx)
unlock *mx

* cond = True
↳ no meio da duas linhas
↳ fora do while

. É necessário uma vcond para cada processo

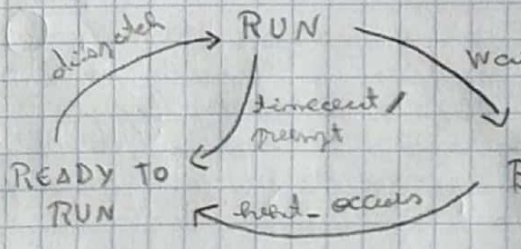
vcond 1 → processo 1
vcond 2 → processo 2
vcond 3 → processo 3

pode ser 1/2/3

. No sem_up colocar → signal vcond (x)

cond(x) = False (colocar antes do signal)

2a)



preemptive → o SO atribui-lhe um time quantum

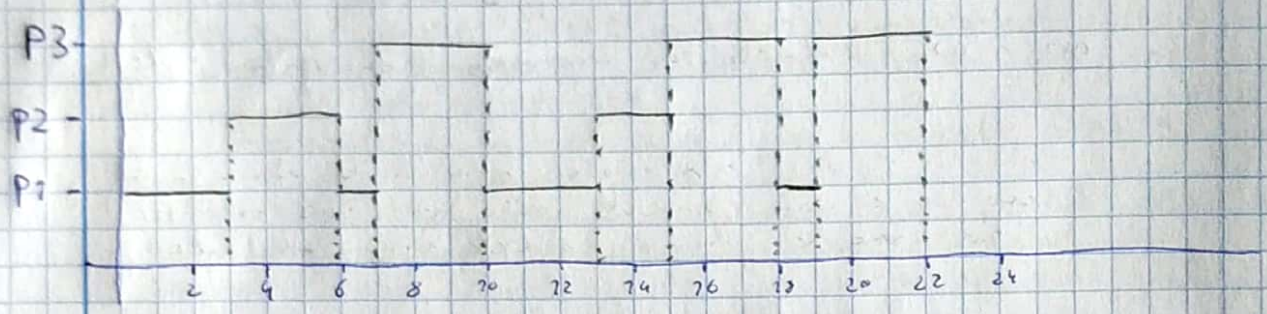
Num escalonador sem prioridades não existe o preempt!

dispatch → ocorre quando o processo que estava a usar o CPU sai → troca o contexto de execução do processo que foi selecionado de forma a que possa ser executado

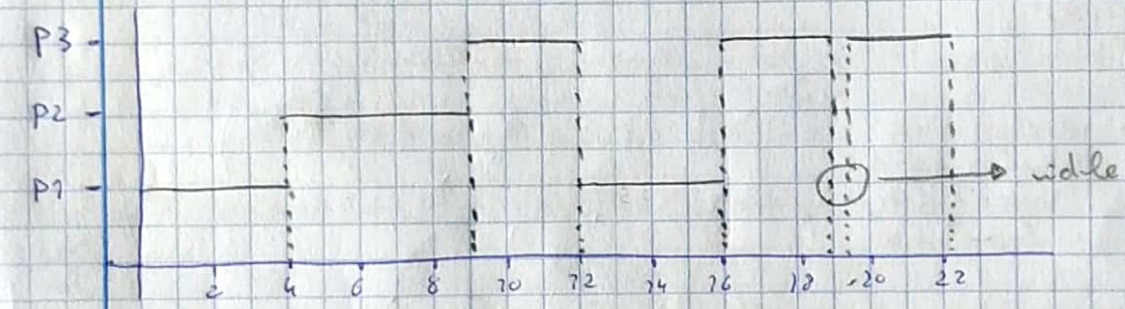
preempt → ocorre quando um processo de mais alta prioridade fica READY

timeout → ocorre quando o time quantum é ultrapassado

b) com prioridades e time quantum de 3



c) . Num sistema non-preemptive, FCFS



Tempo de turnaround → Tempo em que - Tempo em que
acaba é admitido

turnaround (P1) → 16 - 0 = 16

turnaround (P2) → 9 - 2 = 7

turnaround (P3) → 22,2 - 4 = 18,2

3a) Memória real → há uma correspondência biunívoca entre o endereçamento lógico e físico → o espaço de endereçamento lógico é visto como uma sequência de operações de memória que lêem ou escrevem totalmente residentes em memória ou totalmente fora dela

Nas partições fixas há uma divisão prévia da memória, ou seja, quando o SO arranca automaticamente define o conjunto de partições. Cada uma partição de memória

Nas partições variáveis considera-se que inicialmente toda a memória disponível é uma única partição e quando um processo tem que ser lá alojado, o que se faz é partir uma lacuna numa localização onde o processo cabe. O resto fica disponível

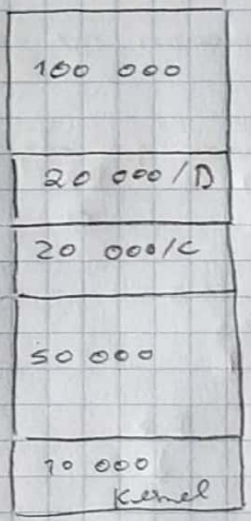
(ii) Registo limite → define o tamanho do espaço de endereçamento lógico → permite validar o endereço

Registo base → indica onde começa o espaço de endereçamento (offset)

(iii) dispatch → quando esta função seleciona um dado processo para ser no estado 'RUN', vai à tabela de controlo de processos e transfere para a MMU os valores do registo de limite e do registo base

(iv) A MMU recebe o endereço, compara-o com o registo limite. Se for maior, não é válido e gera um "segmentation fault". Se for menor, soma o valor do registo base e é gerado o endereço físico

c) . Partições variáveis
. 200 000 unidades de memória → 10 000 para o kernel



Listas ligadas:
↳ Ocupy [C, D]
↳ free [50 000, 100 000]
↓
pode-se representar (ordem do valor)
↓
[(10 000, 50 000); (100 000, 100 000)]
posição tamanho

Occupied: [(60 000, 20 000, C); (80 000, 20 000, D)]
posição tamanho processo

- deadlock prevention
- deadlock detection
- deadlock avoidance

4 a) . Para saber quantos recursos existiam no sistema é somar os recursos da tabela "recursos já adquiridos" com os da tabela "recursos disponíveis"

↳ R1: 8 / R2: 4 / R3: 3

R.: Deadlock avoidance

Deadlock prevention → é da responsabilidade dos processos intervenientes garantirem que nunca ocorre deadlock

Deadlock avoidance → é o gestor de recursos que garante que nunca ocorre deadlock

b) Safe → o sistema está numa situação onde há uma sequência em que todos os processos terminam → evita deadlock

unsafe → o estado embora não esteja em deadlock, dependendo da forma como os processos a seguir executem pode levar a uma situação de deadlock

Nota: Comparar os recursos por adquirir com os recursos que há disponíveis

↳ Quando o processo acaba vai-se à tabela "recursos adquiridos" os recursos que o processo em questão tem na tabela "recursos já adquiridos"

P4 → P1 → P2 → P3

2 2 1 — 5 3 3 — 7 3 3 — 8 4 3

. Consegue terminar todos os processos por isso é um estado safe

c)

	Recursos já adquiridos			Recursos por adquirir		
	R1	R2	R3	R1	R2	R3
P3	1	2	0	3	0	0

Recursos disponíveis

R1	R2	R3
1	0	1

P4 → P1 → P2 → P3

2 1 1 — 5 2 3 — 7 2 3 — 8 4 3

Sim, pode atribuir-lhe o recurso R2