Resolução Exame 2017-2018

1. Considere o programa apresentado a seguir, que representa excertos dos códigos de 3 processos colaborantes. A sincronização é feita através de 3 semáforos, associados às variáveis `sem1`, `sem2` e `sem3`. As operações de *down* e *up* são realizadas pelas funções `sem_down` e `sem_up`, respetivamente. Considere ainda que, após as devidas inicializações, cada processo/thread executa o ciclo `for` correspondente.

```
1  /* Processo/Thread 1: */
2  for (int i = 0; i < 3; i++)
3  {
4      sem_down(sem1);
5      printf("A"); fflush(stdout);
6      sem_up(sem3);
7  }
8
9  /* Processo/Thread 2: */
10 for (int i = 0; i < 3; i++)
11 {
12     sem_down(sem2);
13     printf("B"); fflush(stdout);
14 }
15
16 /* Processo/Thread 3: */
17 for (int i = 0; i < 3; i++)
18 {
19     sem_down(sem3);
20     printf("C"); fflush(stdout);
21     sem_up(sem1);
22     sem_up(sem2);
23 }
```

(a) i. Com que valores mínimos têm de ser inicializados os semáforos `sem1`, `sem2` e `sem3` de modo a que a saída "BCABCABCA" possa ocorrer.

R: 011. Isto apresenta *race conditions* naturalmente.

ii. Considerando a inicialização que indicou, apresente mais duas saídas possíveis.

R: Com 011, CBA|BCA|BCA, BCA|CBA|BCA

(b) Em ambientes *multithreading* é habitual usar-se variáveis de condição em vez de semáforos para a sincronização entre *threads*. Compare as operações de *wait* e *signal*, aplicáveis às variáveis de condição, com as operações de *down* e *up*, aplicáveis a semáforos.

signal é acordar, wait é bloquear.
signal envia sinal para uma thread que está em espera.

wait bloqueia a thread que a invoca até que outra qualquer lhe envie um sinal (utilizando signal)

R:

- Se o *signal* é emitido e nao ha nenhuma thread em *wait*, o signal perde-se. Logo, o *wait* tem que correr primeiro necessariamente. Se um *up* ocorrer, o semaforo fica levantado à espera que apareça o *down*.
- O *wait* fica sempre à espera de um futuro signal. Qualquer *signal* anterior nao acorda o *wait*. O *down* pode se acordar com um *up* que acordou previamente.
- O *wait* bloqueia logo no codigo. O down so bloqueia se o semaforo ja for == 0. Se o semaforo for diferente de 0, nem vai bloquear sequer, nao ocorrendo sincronizaçao. A sincronização é importante em ambientes multithreading pois todos os processos partilham o mesmo espaço de endereçamento.

(c) Considerando uma solução em threads, re-implemente o código dado usando variáveis de condição.

Nao tem a ver mas: mutexes, variaveis de condiçao e semaforos sao recursos do sistema operativo aos quais eu nao tenho o poder de aceder manualmente, mas sim com funçoes adequadas. para mutex, lock e unlock, para vcondiçao, wait e signal e para semaforos up e down.

Temos 3 pontos de sincronizaçao diferentes. Precisamos entao de 3 variaveis de condiçao.

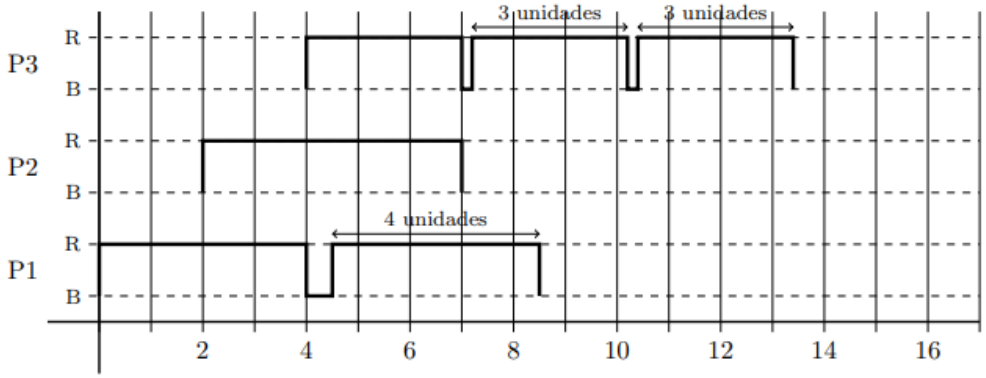
ficaria mais ou menos algo do genero

```
/* Processo/Thread 1: */
for (int i = 0; i < 3; i++) {
        lock (mx);

        while (!cond1) { // cond 1 é variavel booleana
                wait(vcond1, mx); // vcond1 é variavel de condiçao
        }
        cond1 = false; // esta linha so é necessaria porque somos nós que
declaramos as variaveis. Estamos habituados a cond1 ser uma funçao nao feita por
nós, tipo fifoIsEmpty, funçao essa que atualiza o valor booleano sozinha. Aqui
temos que atualizar nós manualmente.

        printf("A"); fflush(stdout);

        cond3 = true;
        signal vcond3;

        unlock (mx);
}

/* Processo Thread 2: */
for (int i = 0; i < 3; i++) {
        lock (mx);
```

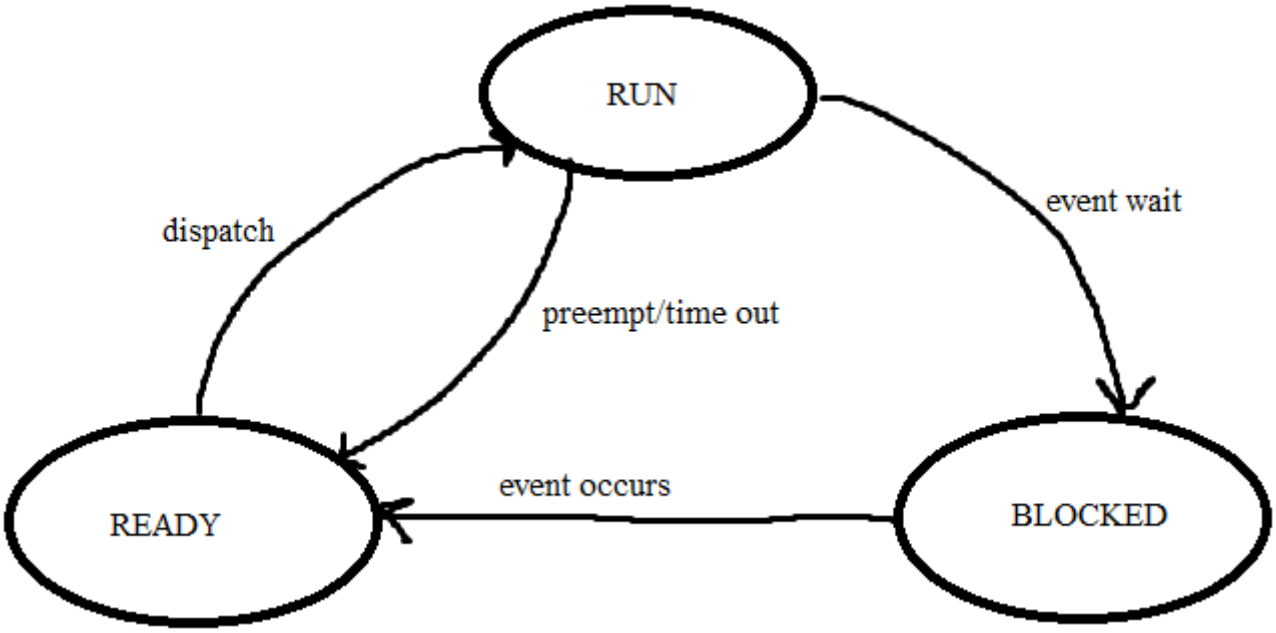
```
        while (!cond2) { // cond 2 é variavel booleana
                wait(vcond2, mx); // vcond2 é variavel de condiçao
        }

        cond2 = false;

        printf("B"); fflush(stdout);

        unlock (mx);
}

/* Processo/ Thread 3: */
for (int i = 0; i < 3; i++) {
        lock (mx);

        while (!cond3) { // cond 1 é variavel booleana
                wait(vcond3, mx); // vcond1 é variavel de condiçao
        }
        cond3 = false;

        printf("C"); fflush(stdout);

        cond1 = true;
        signal vcond1;
        cond2 = true;
        signal vcond2;

        unlock (mx);
}
```

Com semaforos era 011. Aqui será preciso cond2 = true e cond3 = true.

2. O gráfico seguinte representa o estado da execução de 3 processos independentes entre si (mesmo em termos de I/O), P1, P2 e P3, assumindo que correm em processadores (virtuais) distintos. R e B indicam, respetivamente, que o processo está no estado `RUN` (a usar o processador) ou no estado `BLOCKED` (bloqueado à espera de um evento).

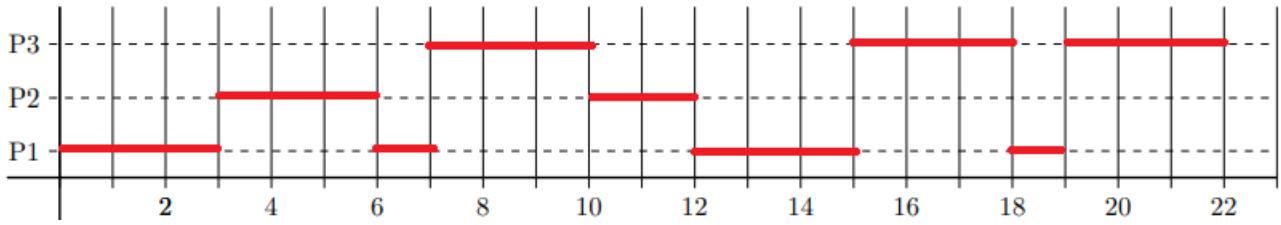


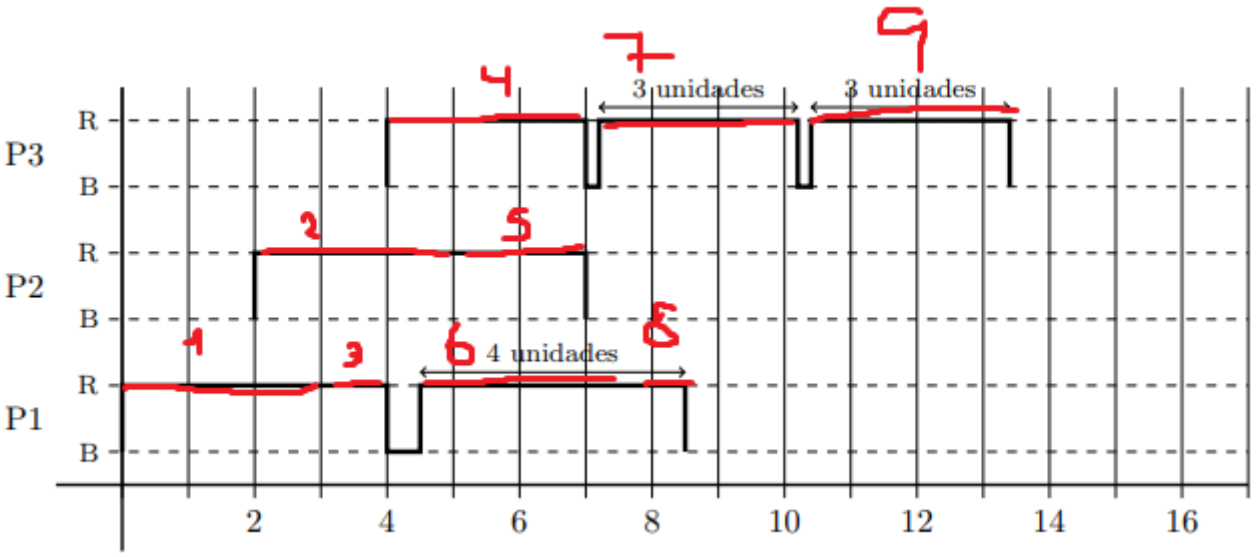
Um sistema preemptive tem transiçoes time-out. Um sistema preemptive com prioridades tem transiçoes time out e preempt. Um sistema non preemptive nao tem nenhuma destas (o que implica que processos usam o cpu indefinitivamente (ate acabarem ou ate o event wait)).



- Preempt significa que um processo de maior prioridade que o processo atualmente a correr entrou em estado READY e portanto, toma posse do processador, tendo o outro que abandonar.
- Time out significa que o processo a correr esgotou o seu time quantum.
- Dispatch mete um processo a correr.
- Event wait significa que um processo está à espera que aconteça um evento externo.
- Event occurs significa que ocorreu o evento externo.

(b) Considere que os 3 processos representados acima correm num ambiente multiprogramado monoprocessador. Usando o gráfico abaixo, trace o diagrama temporal de escalonamento do processador pelos processos P1, P2 e P3, considerando uma política de escalonamento Round Robin sem prioridades e com um *time quantum* (*time slot* atribuído a cada processo) de 3.



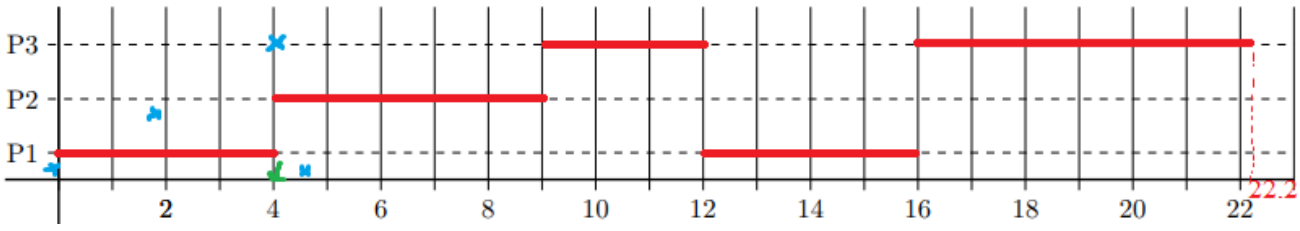


ajuda para resoluçao.

(c) Num sistema *batch*, o tempo de *turnaround* corresponde ao intervalo de tempo entre a submissão de uma tarefa (*job*) e a sua conclusão, incluindo os tempos de espera por recursos. Considerando que os 3 processos representados acima correm num sistema *batch* multiprogramado (não *preemptive*), usando uma disciplina de seleção FCFS (First Come First Served), calcule o tempo de *turnaround* dos processos P1, P2 e P3. Considere que o intervalo de tempo em que o processo P1 está bloqueado é de 0,5 unidades e que cada intervalo de tempo em que o processo P3 está bloqueado é de 0,2 unidades.

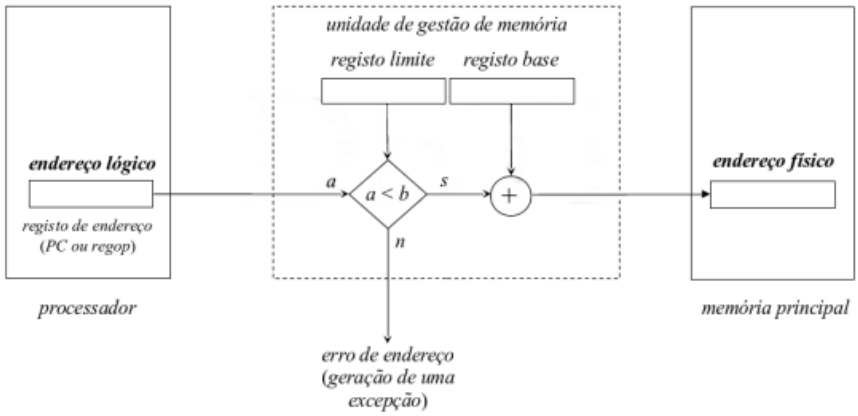
non-preemptive, ou seja, corre para sempre ate acabar ou event wait.
FCFS ou seja fifo.

P1 = 4 (R) + 5 (P2) + 3 (P3) + 4 (R) = 16
P2 = 2 (P1) + 5 (R) = 7
P3 = 5 (P2) + 3 (R) + 4 (P1) + 3 (R) + 0.2 (I) + 3 (R) = 18.2



a azul ta o instante de inserçao.

3. A figura seguinte representa a unidade de tradução do espaço de endereçamento lógico de um processo para endereços físicos, usado num sistema com organização de memória real.



Aqui o ambiente é real, ou seja, o processo ou está totalmente em memoria, ou está totalmente na area de swap e quando ta na memoria ocupa uma area continua.
Numa arquitetura virtual, o espaço de endereçamento do processo está dividido em coisas(blocos, paginas, segmentos...) e estes tao distribuidos pela memoria, portanto, nao necessariamente continuos.

(a) Caracterize uma arquitetura com organização de memória real e compare as duas abordagens para gestão dessa memória, partições fixas e partições variáveis.

A arquitetura real possui correspondencia unica entre o end. logico e fisico, ou seja, o espaço de endereçamento é continuo (implica estar ou totalmente em memoria, ou totalmente fora).

Partições fixas - As partições já têm um tamanho pré definido (podem ser diferentes entre partições), quando um processo é transferido pa memoria, é preciso arranjar uma partição cujo tamanho seja maior que o size do processo. E.g, um processo de size de 0x200 cabe numa partiçao livre de 0x400. Implica fragmentaçao interna.

Partiçoes variaveis - É tipo malloc. Inicialmente, considera-se que a memoria disponivel é uma unica partiçao e quando um processo tem que la ser alojado, "parte-se" esta partiçao numa de size adequada às necessidades do processo. E.g, um processo de size 0x200 cabe na partiçao principal que ocupa 0x10000. Dessa grande partição, fazemos uma partição com 0x200 e metemos o processo la dentro. Implica fragmentaçao externa.

Keep in mind allocation policies: best fit, worst, first, next.

(b) Atendendo à figura,

i. descreva os papeis dos registos base e limite;
ii. diga em que operação do escalonador do processador é que estes registos são alterados;
iii. descreva o procedimento de tradução de um endereço lógico num endereço físico.

1. O limit register deve conter o tamanho em bytes do logical address space; Permite validar o endereço . O base register deve conter a address do inicio da regiao de main memory onde a physical address space do processo é colocada.
2. Na funçao dispatch. Quando a função dispatch seleciona um processo para RUN, vai à PCT e transfere para a MMU os valores do registo limite e base.
3. Se a logical address fornecida pelo CPU for menor que o valor do limit register, é valida, ou seja, ocorre dentro do address space do processo. Se for maior é gerada uma exceção (addr error (seg fault)), porque o endereço nao corresponde a um endereço do espaço de endereçamento. A logical address é depois somada ao valor do base register para produzir o endereço físico.

Ou seja, o processador cria um addr logico 0x005. (Quer aceder ao 5º byte do inicio do processo idk) Esse processo digamos que tem um size de 0x200. Logo é valido. Pega-se no 0x005 e soma-se ao inicio da addr. fisica do processo (e.g 0x2000 + 0x0005, acedemos ao byte presente na addr fisica 0x2005 na memoria fisica).

(c) Considerando uma arquitetura de partições variáveis, considere que a memória real tem 200000 unidades de memória, das quais as primeiras 10000 são reservadas para o *kernel* do sistema de operação. Partindo da situação inicial (nenhum processo está alocado em memória), 4 processos (A, B, C e D) entram em jogo da seguinte forma:

- A, usando 10000 unidades de memória, é alocado;
- B, usando 40000 unidades de memória, é alocado;
- C, usando 20000 unidades de memória, é alocado;
- B, sai, libertando a memória que usava;
- D, usando 20000 unidades de memória, é alocado;
- A, sai, libertando a memória que usava;

Considerando que a política de alocação usada é a *worst fit*, represente o estado da memória real após a sequência de acções anterior.

Worst fit.

Temos 200k unidades, 10k primeiras alocadas permanentemente para o OS.

10.000 - OS	190.000

1. Entra A, usando 10k

10.000 - OS	10.000 - A	180.000

2. Entra B, usando 40k

10.000 - OS	10.000 - A	40.000 - B	140.000

3. Entra C, usando 20k

10.000 - OS	10.000 - A	40.000 - B	20.000 - C	120.000

4. B sai

10.000 - OS	10.000 - A	40.000	20.000 - C	120.000

5. Entra D, usando 20k

10.000 - OS	10.000 - A	40.000	20.000 - C	20.000 - D	100.000

6. A sai

10.000 - OS	10.000	40.000	20.000 - C	20.000 - D	100.000

Dois buracos continuos, fazemos merge.

10.000 - OS	50.000	20.000 - C	20.000 - D	100.000

Se pedirem occupied list é do tipo: `(start, size, process)` ou seja `[(60.000, 20.000, C), ...]`
Free list igual mas sem o processo, claro.

4. Considere que 4 processos (P1, P2, P3 e P4) partilham recursos de 3 categorias diferentes (R1, R2 e R3). Os recursos são geridos por uma entidade que exige aos processos a declaração inicial das quantidades máximas de cada tipo de recurso que podem eventualmente necessitar. A seguir, os processos podem pedir recursos e a entidade gestora apenas os atribui se o sistema se mantiver, após a atribuição, num estado seguro. A tabela **`Estado dos processos`** ilustra as necessidades máximas de recursos declaradas pelos vários processos, os já adquiridos e os ainda por adquirir. A tabela **`Recursos disponíveis`** indica os recursos que a entidade de gestão ainda tem disponíveis para atribuição.

Estados dos processos

	Recursos declarados			Recursos já adquiridos			Recursos por adquirir		
	R1	R2	R3	R1	R2	R3	R1	R2	R3
P1	5	1	2	3	1	2	2	0	0
P2	2	2	2	2	0	0	0	2	2
P3	4	2	0	1	1	0	3	1	0
P4	2	1	0	1	1	0	1	0	0

Recursos disponíveis

R1	R2	R3
1	1	1

(a) Estudou políticas de prevenção de *deadlock* em sentido estrito (*deadlock prevention*) e em sentido lato (*deadlock avoidance*). Em que categoria coloca o sistema apresentado acima? Justifique a sua resposta.

É deadlock avoidance e usa a approach de resource allocation denial.
No preventation, é da responsabilidade dos processos garantir que nunca ocorre deadlock.
Aqui, é o gestor de recursos que faz essa funçao.

(b) Um sistema deste tipo pode encontrar-se nos estado *safe*, *unsafe* ou em *deadlock*. Mostre que na situação representada o sistema se encontra num estado *safe*, apresentando uma sequência de execução (incluindo os correspondentes estados do sistema) que o probe.

- Deadlock - Estado deadlock. O sistema nao deve permiti-lo.
- Safe - Há sempre uma sequencia possivel de forma a evitar deadlock.
- Unsafe - Ainda nao ha deadlock, mas dependendo da forma de evoluçao de aquisiçao de recursos, pode levar a uma situaçao de deadlock que o gestor nao consegue evitar.

Neste momento está no estado safe. Se, e.g nao houvesse 1 R1, P4 podia nao conseguir terminar, e sendo assim, nenhum estado conseguiria terminar, estariamos em deadlock.

1. Metemos o gestor a apenas atender pedidos do P4 ate que ele termine e liberte recursos. Ficamos com [2 2 1] recursos.
2. Metemos o gestor a apenas atender pedidos do P1 ate que ele termine e liberte recursos. Ficamos com [5 3 3] recursos.
3. O P2 e o P3 podem executar como quiserem.

(c) Se o processo P3 pede um recurso do tipo R2, o sistema pode ou não atribuir-lho imediatamente? Justifique a sua resposta.

Simulamos este acontecimento (modificamos tabela) e voltamos a fazer a alinea b) Chegamos à conclusao que ha uma sequencia em que todos os processos terminam normalmente.
Pode.