Esta ficha foi elaborada para acompanhar o estudo da disciplina de Sistemas de Operação (a4s1) ou para complementar a preparação para os momentos de avaliação finais da mesma. Num segundo ficheiro poderás encontrar um conjunto de propostas de solução aos exercícios que estão nesta ficha. É conveniente relembrar que algum conteúdo destes documentos pode conter erros, aos quais se pede que sejam notificados pelas vias indicadas na página web, e que serão prontamente corrigidos, com indicações de novas versões.

1. A possibilidade de ocorrência de deadlock pressupõe a satisfação de quatro condições.

a) Quais são elas?

As quatro condições de ocorrência de deadlock são a de exclusão mútua (em que a posse de um recurso não pode ser atribuída a mais que um processo), a de espera com retenção (em que cada processo, ao requerer um novo recurso, mantém na sua posse todos os recursos previamente solicitados), a de não-libertação (em que nenhum segundo ou outro processo pode tirar recursos a um primeiro) e a de espera circular (em que estando num ciclo vicioso, há uma cadeia de processos e recursos, em que cada processo requer um recurso que está na posse do processo seguinte na cadeia).

b) Dadas as quatro condições, o que é necessário fazer para corrigir uma situação de deadlock, depois de analisar o problema?

Para corrigir uma situação de deadlock basta negar uma das condições, dado que a implicação de uma conjunção no consequente é equivalente à implicação da disjunção no antecedente. Assim, podemos procurar uma primeira solução tentando corrigir a exclusão mútua ou a espera com retenção ou a não-libertação ou a espera circular.

2. Na área de redes, mais em particular no protocolo Ethernet, existe um modo de acesso ao meio denominado de CSMA/CD (Carrier Sensing Multiple Access with Collision Detection), onde se tenta aproximar a solução de exclusão mútua (numa ligação partilhada entre máquinas), de forma a que duas máquinas não usem um mesmo canal em simultâneo, perturbando o envio/receção de pacotes. Este protocolo de acesso ao meio usa um algoritmo modelado para a linguagem C, conforme o código abaixo, onde temos a entrada e a saída de uma região crítica, para uma comunicação entre duas máquinas (considere-se que a função delay(s) provoca um atraso de s segundos).

```
unsigned bool wantEnter[2] = {false, false};

void enterCriticalSection(unsigned int pid) {
  unsigned int otherPID = 1 - pid;
  wantEnter[pid] = true;
  while (wantEnter[otherPID]) {
    wantEnter[pid] = false;
    delay(random);
    wantEnter[pid] = true;
  }
}

void exitCriticalSection(unsigned int pid) {
  wantEnter[pid] = false;
}
```

a) A solução acima tem alguma possibilidade de garantir exclusão mútua, na comunicação entre as duas máquinas?

Sim, porque um dos processos recua temporariamente, de forma a resolver qualquer impasse que possa existir na atribuição da região a um processo. Este recuo é feito de forma arbitrária, através do atraso aleatório da atribuição.

b) A solução acima pode ser considerada válida para garantir exclusão mútua?

Não. Como a solução se baseia num processo aleatório poderá sempre acontecer que o atraso atribuído aos dois processos seja o mesmo, logo fazendo com que a atribuição fique presa num impasse. Dado que o método não é determinístico, então a solução acima não é válida para garantir a exclusão mútua.

c) O algoritmo Dekker usa o mecanismo de alternância da solução acima, mas de forma determinística, atribuindo prioridades aos processos. No entanto esta solução apenas é válida, não se tendo ainda achado forma, para 2 processos. Tenta modificar o código para atingir a solução criada por Dekker.

```
unsigned bool wantEnter[2] = {false, false};
unsigned int priority = 0;

void enterCriticalSection(unsigned int pid) {
  unsigned int otherPID = 1 - pid;
  wantEnter[pid] = true;
```

```
  while (wantEnter[otherPID]) {
    if (pid != priority) {
      wantEnter[pid] = false;
      while (pid != priority);
      wantEnter[pid] = true;
    }
  }
}

void exitCriticalSection(unsigned int pid) {
  unsigned int otherPID = 1 - pid;
  priority = otherPID;
  wantEnter[pid] = false;
}
```

d) Como dito na alínea anterior, o algoritmo de Dekker apenas é válido para o caso de 2 processos. Para N processos, Gary Peterson criou uma solução que torna a de Dekker obsoleta, à qual hoje denominamos de algoritmo de Peterson. Em que é que difere da anterior?

O algoritmo de Peterson, ao invés de identificar prioridades na atribuição dos processos, usa uma seriação por ordem de chegada para resolver o conflito resultante da contenção destes. Basicamente, quando um processo entra numa região crítica é obrigado a escrever a sua própria identificação numa variável global, sendo que a entrada só se torna real depois da confirmação que ninguém mais está na região.

e) O que é o adiamento indefinido? O algoritmo de Peterson resolve a questão do adiamento indefinido?

O adiamento indefinido ocorre quando um ou mais processos competem pelo acesso a uma região crítica e, devido a um conjunto de circunstâncias em que surgem continuamento processos novos, o acesso é sucessivamente adiado, pelo que se está, por isso, perante um impedimento real à continuação dele(s). O algoritmo de Peterson resolve este problema, identificando sempre de quem é que é a vez de entrar na região crítica, atribuindo sempre o recurso a um processo.

3. Em 1965, Edsger Dijkstra sugeriu usar uma variável inteira para contar o número de vezes que os processos são acordados.

a) De que estrutura de dados é que estamos a falar e no que consiste?

Estamos a falar dos semáforos e estes são variáveis que podem ter o valor 0, indicando que nenhum wakeup foi guardado ou um número positivo se um ou mais wakeups estiverem pendentes.

b) Quais são as operações que podemos executar com semáforos?

Com os semáforos podemos executar as operações de up e down, de forma atómica. A operação de down serve para verificar se o valor de um semáforo é maior que 0, decrementando-o, caso seja. Caso não seja maior que 0, então o processo é colocado em estado sleep. A operação de up incrementa o valor do semáforo endereçado. Se um ou mais processos estiverem a dormir num mesmo semáforo, à espera de poderem terminar um down que tenha ficado em espera, então um deles (escolhidos pelo sistema de operação) é permitido de fazer o down.

c) Dados os seguintes processos executados em paralelo e as suas respetivas sequências de código, indica se existiria ou não uma situação de deadlock e explica porquê. De qualquer forma, indica também qual seria o output e o valor dos semáforos, considerando que são todos inicializados a 0.

```
// processo 1
printf("3");
sem_post(&s3);
printf("4");
sem_post(&s2);
sem_post(&s1);

// processo 2
sem_wait(&s1);
printf("1");
sem_wait(&s3);
sem_post(&s4);
sem_wait(&s3);

// processo 3
sem_wait(&s2);
sem_wait(&s4);
printf("2");
printf("5");
sem_post(&s3);
```

Inicialmente são executadas as linhas primeiras de todos os processos, imprimindo "3" e bloqueando os processos 2 e 3 pelos semáforos s1 e s2, respetivamente, dado que ambos s1 e s2 estavam a 0. Tendo sido bloqueados, então passamos para a execução do sem_post(&s3) (o que incrementa o semáforo, passando para 1) e para a impressão da string "4". Depois fazemos um incremento do semáforo s2, que liberta o processo 3, que estava bloqueado e que agora decrementa o semáforo s4, bloqueando o processo 3 novamente. Enquanto isto acontece é feita a libertação do processo 2 por vias do incremento de s1 no processo 1. Este processo agora imprimirá "1", decrementa s3 (passando para 0) e faz post do s4, incrementando-o para 1 e libertando a execução do processo 3, que imprimirá "2" e "5". Entretanto o processo 2 bloqueia com o decremento de s3 (que já estava a 0) e é libertado com o post de s3 no processo 3, terminando a execução dos processos. O output final é então "34125" e não haverá nenhuma situação de deadlock.

4. No programa abaixo criaram-se 3 threads capazes de escrever o carater A, B ou C no terminal. Introduz os semáforos de forma oportuna para que o output seja ABCABCABCABCABCABC.

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>

#define MAX 6

void* writeA(void* arg) {
  int num;
  for (num = 0; num != MAX; num++) {
    printf("A");
    fflush(stdout);
    sleep(random() % 3);
  }
  pthread_exit(NULL);
}

void* writeB(void* arg) {
  int num;
  for (num = 0; num != MAX; num++) {
    printf("B");
    fflush(stdout);
    sleep(random() % 2);
  }
  pthread_exit(NULL);
}

void* writeC(void* arg) {
  int num;
  for (num = 0; num != MAX; num++) {
    printf("C");
    fflush(stdout);
    sleep(random() % 2);
  }
  pthread_exit(NULL);
}

int main(int argc, char* argc[]) {
  pthread_t thread1, thread2, thread3;
  srandom(time(NULL));
  pthread_create(&thread1, NULL, writeA, NULL);
  pthread_create(&thread2, NULL, writeB, NULL);
  pthread_create(&thread3, NULL, writeC, NULL);
  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);
  pthread_join(thread3, NULL);
  return 0;
}
```

Considerem-se, ainda, as seguintes assinaturas de funções:

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_destroy(sem_t *sem);
int sem_post(sem_t *sem);
int sem_wait(sem_t *sem);
```

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
```

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

#define MAX 6

sem_t semA, semB, semC;

void* writeA(void* arg) {
  int num;
  for (num = 0; num != MAX; num++) {
    sem_wait(&semA);
    printf("A");
    fflush(stdout);
    sem_post(&semA);
    sleep(random() % 3);
  }
  pthread_exit(NULL);
}

void* writeB(void* arg) {
  int num;
  for (num = 0; num != MAX; num++) {
    sem_wait(&semB);
    printf("B");
    fflush(stdout);
    sem_post(&semB);
    sleep(random() % 2);
  }
  pthread_exit(NULL);
}

void* writeC(void* arg) {
  int num;
  for (num = 0; num != MAX; num++) {
    sem_wait(&semC);
    printf("C");
    fflush(stdout);
    sem_post(&semC);
    sleep(random() % 2);
  }
  pthread_exit(NULL);
}

int main(int argc, char* argc[]) {
  pthread_t thread1, thread2, thread3;
  srandom(time(NULL));
  sem_init(&semA, 0, 1);
  sem_init(&semB, 0, 0);
  sem_init(&semC, 0, 0);
  pthread_create(&thread1, NULL, writeA, NULL);
  pthread_create(&thread2, NULL, writeB, NULL);
  pthread_create(&thread3, NULL, writeC, NULL);
  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);
  pthread_join(thread3, NULL);
  sem_destroy(&semA);
  sem_destroy(&semB);
  sem_destroy(&semC);
  return 0;
}
```

5. Relaciona semáforos com mutexes.

Os semáforos que são inicializados a 1 e são usados por dois ou mais processos para garantir que somente um deles é que pode entrar numa região crítica em simultâneo são denominados de semáforos binários. Estes semáforos podem ser considerados uma forma de mutex. Um mutex (palavra proveniente de MUTual EXclusion) são estruturas de dados fáceis de manipular com um de dois estados: bloqueado ou liberto (tal como um semáforo binário).

6. Identifica a diferença entre monitores de Hoare e de Brinch Hansen.

No monitor de Hoare a thread que invoca a operação de envio de sinal é colocado fora do monitor, para que a thread acordada possa prosseguir. Esta implementação exige que haja uma stack onde são colocadas as threads

postas fora do monitor por invocação de um sinal. Com a solução apresentada por Brinch Hansen, a thread que invoca a operação de envio de sinal liberta imediatamente o monitor.