Sistemas de Operação

Ano lectivo de 2010-2011

Comunicação entre Processos

1. Como caracteriza os processsos em termos de interacção? Mostre como em cada categoria se coloca o problema da exclusão mútua.
2. O que é a competição por um recurso? Dê exemplos concretos de competição em, pelos menos, duas situações distintas.
3. Quando se fala em região crítica, há por vezes alguma confusão em estabelecer-se se se trata de uma *região crítica* de código, ou de dados. Esclareça o conceito. Que tipo de propriedades devem apresentar as primitivas de acesso com exclusão mútua a uma região crítica?
4. Não havendo exclusão mútua no acesso a uma região crítica, corre-se o risco de inconsistência de informação devido à existência de *condições de corrida* na manipulação dos dados internos a um recurso, ou a uma região partilhada. O que são condições de corrida? Exemplifique a sua ocorrência numa situação simples em que coexistem dois processos que cooperam entre si.
5. Distinga *deadlock* de adiamento indefinido. Tomando como exemplo o *problema dos produtores / consumidores*, descreva uma situação de cada tipo.
6. A solução do problema de acesso com exclusão mútua a uma região crítica pode ser enquadrada em duas categorias: soluções ditas *software* e soluções ditas *hardware*. Quais são os pressupostos em que cada uma se baseia?
7. Dijkstra propôs um algoritmo para estender a solução de Dekker a N processos. Em que consiste este algoritmo? Será que ele cumpre todas as propriedades desejáveis? Porquê?
8. O que é que distingue a solução de Dekker da solução de Peterson no acesso de dois processos com exclusão mútua a uma região crítica? Porque é que uma é passível de extensão a N processos e a outra não (não é conhecido, pelo menos até hoje, qualquer algoritmo que o faça).
9. O tipo de fila de espera que resulta da extensão do algoritmo de Peterson a N processos, é semelhante àquela materializada por nós quando aguardamos o acesso a um balcão para pedido de uma informação, aquisição de um produto, ou obtenção de um serviço. Há, contudo, uma diferença subtil. Qual é ela?
10. O que é o *busy waiting*? Porque é que a sua ocorrência se torna tão indesejável? Haverá algum caso, porém, em que não é assim? Explique detalhadamente.
11. O que são *flags de locking*? Em que é que elas se baseiam? Mostre como é que elas podem ser usadas para resolver o problema de acesso a uma região crítica com exclusão mútua.
12. O que são semáforos? Mostre como é que eles podem ser usados para resolver o problema de acesso a uma região crítica com exclusão mútua e para sincronizar processos entre si.
13. O que são monitores? Mostre como é que eles podem ser usados para resolver o problema de acesso a uma região crítica com exclusão mútua e para sincronizar processos entre si.
14. Que tipos de monitores existem? Distinga-os.
15. Que vantagens e inconvenientes as soluções baseadas em monitores trazem sobre soluções baseadas em semáforos?
16. Em que é que se baseia o paradigma da passagem de mensagens? Mostre como se coloca aí o problema do acesso com exclusão mútua a uma região crítica e como ele está implicitamente resolvido?

17. O paradigma da passagem de mensagens adequa-se de imediato à comunicação entre processos. Pode, no entanto, ser também usado no acesso a uma região em que se partilha informação. Conceba uma arquitectura de interacção que torne isto possível.

18. Que tipo de mecanismos de sincronização podem ser introduzidos nas primitivas de comunicação por passagem de mensagens? Caracterize os protótipos das funções em cada caso (suponha que são escritas em Linguagem C).

19. O que são sinais? O *standard* Posix estabelece que o significado de dois deles é mantido em aberto para que o programador de aplicações lhes possa atribuir um papel específico. Mostre como poderia garantir o acesso a uma região crítica com exclusão mútua por parte de dois processos usando um deles.
    Sugestão – Veja no manual *on-line* como se pode usar neste contexto a chamada ao sistema *wait*.

20. Mostre como poderia criar um canal de comunicação bidireccional (*full-duplex*) entre dois processos parentes usando a chamada ao sistema *pipe*.

21. Como classifica os recursos em termos do tipo de apropriação que os processos fazem deles? Neste sentido, como classificaria o canal de comunicações, uma impressora e a memória de massa?

22. Distinga as diferentes políticas de *prevenção de deadlock no sentido estrito*. Dê um exemplo ilustrativo de cada uma delas numa situação em que um grupo de processos usa um conjunto de blocos de disco para armazenamento temporário de informação.

23. Em que consiste o algoritmo dos banqueiros de Dijkstra? Dê um exemplo da sua aplicação na situação descrita na questão anterior.

24. As políticas de *prevenção de deadlock no sentido lato* baseiam-se na transição do sistema entre estados ditos *seguros*. O que é um estado seguro? Qual é o princípio que está subjacente a esta definição?

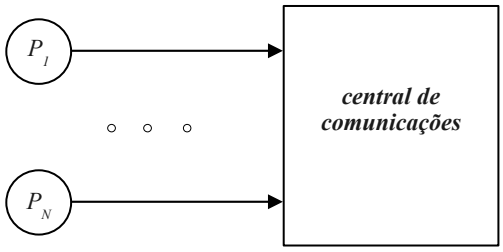
25. Que tipos de custos estão envolvidos na implementação das políticas de *prevenção de deadlock nos sentidos estrito e lato*? Descreva-os com detalhe.

Sistemas de Operação

Ano lectivo de 2010-2011

Problema técnico

A figura abaixo representa esquematicamente um sistema de troca de mensagens entre processos de um mesmo sistema computacional que comunicam entre si.



Supõe-se que em cada instante podem estar a ocorrer trocas de mensagens entre um número variável de pares de processos. Antes que tal suceda, porém, cada processo tem que se registar na central através de um identificador único, usando para isso a primitiva

```
int register_in (unsigned int id);
```

que devolve os valores

0 – *operação realizada com sucesso*
1 – *já existe o registo de um processo com o id fornecido*
2 – *tabela de id's cheia* (*não se aceitam mais registos*) .

Do mesmo modo, no fim da aplicação, todos os processos envolvidos, antes de terminarem, devem notificar a central de que os seus serviços não são mais necessários, executando a primitiva

```
int register_out (unsigned int id);
```

que devolve os valores

0 – *operação realizada com sucesso*
1 – *id desconhecido* (*nada é feito*) .

Note que, se houver mensagens pendentes quando um processo se desliga da central, elas serão todas descartadas e os processos, eventualmente bloqueados a aguardar comunicação com o processo em causa, serão acordados com a indicação de que não existe qualquer processo com o *id* fornecido.

Estão implementados os tipos de sincronização na passagem de mensagens seguintes

Sincronização não bloqueante

```
int non_block_send (unsigned int source_id, unsigned int dest_id,
                    MESSAGE m);
int non_block_rec (unsigned int source_id, unsigned int dest_id,
                   MESSAGE *pm, int *p_is_mess);
```

Sincronização bloqueante remota

```
int async_block_send (unsigned int source_id, unsigned int dest_id,
                      MESSAGE m);
int async_block_rec (unsigned int source_id, unsigned int dest_id,
                     MESSAGE *pm);  .
```

Nota o processo destinatário deve reservar sempre espaço no espaço de endereçamento local para arma-zenamento completo da mensagem que vai ser recebida antes da invocação de uma primitiva de recepção.

Os valores devolvidos pela invocação das primitivas acima são

0 – *operação realizada com sucesso*
1 – *source_id igual a dest_id*
2 – *source_id desconhecido*
3 – *dest_id desconhecido*
4 – *ponteiro nulo*
5 – *falta de memória para processamento da mensagem* .

A mensagem comunicada tem o formato genérico indicado abaixo

```
typedef struct
        { unsigned int size;                        /* tamanho em bytes */
          void *strt;                    /* ponteiro para o seu início */
        } MESSAGE;  .
```

A central de comunicações é, por outro lado, definida pela estrutura de dados

```
#define  N   100
typedef struct
        { unsigned int access;           /* id do semáforo de acesso à
                                          manipulação da estrutura de dados */
          unsigned int n_reg_pt;     /* n.º de pontos de registo livres */
          REG_PT reg[N];               /* array dos pontos de registo */
        } COMM_CENTR;
```

em que o ponto de registo se caracteriza por

```
typedef struct
        { int busy;                  /* indicação do estado de ocupação
                                                (0 – livre / 1 - ocupado) */
          unsigned int id;              /* n.º de registo do processo */
          INFO *list;                 /* lista biligada das mensagens
                                                destinadas ao processo */
        } REG_PT;  .
```

Esta lista biligada contém toda a informação necessária à recolha de mensagens dos diferentes processos remetentes e está organizada como se indica a seguir

```
struct info
{ unsigned int id;                 /* identificação do processo remetente */
  int arrival;          /* sinalização da sua chegada ao ponto de encontro
                                   (0 – não chegou ainda / 1 – já chegou) */
  unsigned int wait;    /* identificação do semáforo de espera pelo outro
                           processo, no caso de sincronização bloqueante
                            o valor 0 significa que não está a ser usado */
  MESSAGE m;                                                /* mensagem */
  struct info *ant, *next;       /* ponteiros de ligação (lista biligada) */
};
typedef struct info INFO;
```

Como é fácil concluir do exposto, sempre que ocorre uma troca de mensagens entre um processo remetente e um processo destinatário, será criado e introduzido na lista biligada do ponto de registo do processo destinatário um novo nó, pelo processo remetente (caso da sincronização não-bloqueante), ou pelo processo que chegue em primeiro lugar (caso da sincronização bloqueante remota).

Assuma ainda que tem à sua disposição as primitivas básicas de reserva e libertação de espaço em memória dinâmica (nível do *kernel*)

```
void *malloc (unsigned int size);
void free (void *pnt);
```

de manipulação de semáforos

```
int sem_create (void); ,
```

é devolvido o identificador do semáforo (que fica com o campo *val* a zero após a sua criação)

```
void sem_destroy (unsigned int sem_id);
void sem_down (unsigned int sem_id);
void sem_up (unsigned int sem_id); ,
```

de manipulação da lista biligada

pesquisa de existência de um nó para um par remetente-destinatário específico

```
INFO *msg_search (unsigned int source_id, unsigned int reg_index);
```

é devolvido o ponteiro nulo se não existir qualquer nó

introdução de um novo nó na lista biligada (numa perspectiva *FIFO*)

```
void msg_in (INFO *new, unsigned int reg_index);
```

retirada de um nó específico da lista biligada (sempre que possível numa perspectiva *FIFO*)

```
void msg_out (INFO *old, unsigned int reg_index);
```

e que foi definida a variável seguinte

```
static COMM_CENTR comm;            /* central de comunicações */  .
```

1. O campo *wait* da estrutura de dados *INFO* tem como função conter o identificador do semáforo de espera, usado em situações de sincronização bloqueante. Indique para todos os casos relevantes qual é processo que cria o semáforo e qual é o que o destroi.
2. Será que as primitivas de sincronização podem ser usadas por cada par de processos de um modo completamente livre; isto é, o remetente pode invocar um *send* bloqueante (não bloqueante) e o destinatário invocar um *rec* não bloqueante (bloqueante) para a mesma transacção? Justifique a sua resposta.
3. Assumindo que existem N processos que comunicam entre si usando a central de comunicações, procure determinar para cada um dos casos, *sincronização não bloqueante* e *sincronização bloqueante*, qual é o número máximo de nós necessários à correcta operação do sistema. Justifique cuidadosamente a sua resposta.
4. Construa as primitivas de manipulação da lista biligada `msg_search`, `msg_in` e `msg_out`.
5. Construa as primitivas de envio e de recepção de mensagens, com sincronização não bloqueante e bloqueante remota, respectivamente, `non_block_send`, `non_block_rec`, `async_block_send` e `async_block_rec`.
6. Construa as primitivas de registo e de anulação de registo na central de comunicações `register_in` e `register_out`.