PROVA 01 – 2010/01

Observação: as respostas abaixo salientam **apenas os principais conceitos** e não se espera que elas sejam a resposta padrão. Nas respostas da prova se espera que essas ideias sejam elaboradas de acordo com o solicitado e, respostas diferentes, bem fundamentadas, são consideradas de acordo com sua correção e argumentação.

1ª Questão

A barreira é um ponto de sincronização para n processos. Eles só irão adiante quando todos os processos participantes da barreira chegarem nesse ponto. A solução trivial é um implementar um contador para contabilizar quantos processos já chegaram no ponto de sincronização (a barreira). O contador é uma variável compartilhada entre os processos.

```
int count=0;

process worker[i=1 to n]
    while (true) {
        código da tarefa i;
        <count = count +1;>
        <await (count ==n);>
    }
}
```

O problema na solução acima é a necessidade zerar count no final da interação, antes que um processo comece a incrementar novamente o contador count. Inserir no final algo da forma abaixo adiantaria?

```
if ( i == 1)
     <count =0>;
else <await (count == 0);
```

A resposta é não, pois continua a existir a possibilidade de um processo que espera por count ==0 na etapa *k* ver o count incrementado por um outro processo que já terminou a etapa *k*-1. A solução é fazer os processos usarem um par de contadores compartilhados (count1 e count2) e a cada interação contar de zero a *n* e de *n* a zero.

```
step ++;    /*step é uma variável local a cada processo inicializada em zero */
if (step mod 2) { /* etapa impar */
   <count1 = count1 +1;>
   <await (count1 ==n);>
    <count1 = 0>;
}
else {/*etapa par*/
   <count2 = count2 -1;>
   <await (count2 ==0);>
    <count2 = n>;
}
```

No entanto, existem soluções mais elegantes, como por exemplo:

```
Monitor M{
Int count = 0;
ConditionVariable Barrier;
void barrier_wait( int processid) {
    if (++count != N)
        condition.wait(Barrier);
    else {
        count = 0;
        condition.broadcast(Barrier);
   }
}
```

OU

```
P(mutex);
if (++count != N)
   V(mutex);
   P(Barreira[id]);
}
else {
    count = 0;
    V(mutex);
    for(i=1, i<= N, i++)
        if ( i != id) V(Barreira[id])
}
```

Que podem ser reescritas usando apenas variáveis de condição ou semáforos de acordo com a equivalência entre esses mecanismos. Detalhes das soluções em Andrews (pg. 115).

Solução (bugada!!): empregar um semáforo contador barrier inicializado com zero. Cada processo que chegasse na barreira realizaria um P(barrier) é ficaria bloqueado até que o n-ésimo chegue na barreira é faça n-1 operações V(barrier) liberando os processos bloqueados anteriormente.

```
P(mutex);
If (++ count != n ) {
   V(mutex);
   P(barrier);
}
else {
    count = 0;
    V(mutex);
    for ( i = 1; i <  n; i++)
        V(barrier)
}
```

O problema: suponha N=3, os dois primeiros processos bloqueiam no P(barrier). O último processo chega na barreira, entra no *else*, zera o contador e inicia a execução do laço *for*, mas não completa ele. A barreira não foi concluída para uma etapa *k*. O primeiro processo que realizou P é então liberado para continuar (foi feito pelo menos um V), ele pode executar e concluir sua nova tarefa (etapa *k*+1) e voltar a esperar P(barrier). Nesse caso, se mistura em P(barrier) processos da etapa *k* com processos da etapa *k*+1. Solução: usar um semáforo por processo (vetor de semáforos).

A maior desvantagem de todas as soluções apresentadas é que o contador *count* se torna um gargalo, já que todos os n processos devem tratá-lo como uma seção crítica. A vantagem nessas soluções é a simplicidade. Para contornar o problema de gargalo (contenção) de acesso a n deve-se usar outras formas de sincronização (em árvore, butterfly, simétrica, etc) que são mais complexas.

2ª Questão

(a) **Mutex**: o emprego principal é para fazer exclusão mútua, fazendo a proteção de uma seção crítica. **Semáforos**: servem para realizar controle de recursos (binários e contadores) e para sincronização. **Monitores**: construção de mais alto nível para controle de recursos e sincronização. **Variáveis de condição**: sinalizar eventos que condicionam a execução de um processo/thread. Esses mecanismos não são totalmente equivalentes entre si, por exemplo, mutex não é capaz de sinalizar eventos, nem fazer sincronização entre processos.

(b) O principal é a questão da heterogeneidade de máquinas (representação interna de dados), de sistemas operacionais e de linguagens usadas no desenvolvimento das aplicações. A desvantagem é uma relação entre desempenho x portabilidade e facilidade de programação. A vantagem é justamente uma questão de portabilidade e facilidade de programação. Em relação a representação de dados tem-se RPC usando o XDR, serialização no RMI e XML no caso de web services.

3ª Questão

Ver detalhes na seção 7.3.1 do livro do Andrews em conjunto com a tabela 7.1 e figura 7.7, mas basicamente:

(a) as variáveis permanentes do monitor podem ser as variáveis globais do processo servidor e as variáveis locais são as variáveis locais de cada função dentro do processo servidor.

(b) A entrada no monitor é uma primitiva do tipo receive bloqueante que é executada pelo servidor. Se o tratamento das mensagens é sequencial, haverá apenas um processo dentro do monitor.

(c) A primitiva wait corresponde a não responder o processo cliente deixando pendente a resposta para ser enviada mais tarde. A primitiva signal recupera uma mensagem pendente e envia a resposta ao cliente

Adaptação da figura 7.5 para incluir dois métodos no monitor: produce e consume. O objetivo é exemplificar como o monitor seria implementado em um modelo cliente-servidor, não é ter o melhor código produtor-consumidor. **Na prova não se esperava o nível de detalhamento abaixo**. O importante era dizer como seriam simulados o *wait* e o *signal* e a presença de apenas um processo no monitor (**partes assinaladas em negrito**).

```
type      op_kind = enum (PRODUCE, CONSUME);
int       avail = 0, out = 0, int = 0,  item[MAXUNITS];
queue     pending_prod, pending_cons;

int consume_item( ){
    int unit;
    avail --; unit = item[out]; out = (out+1) mod MAXUNITS;
    return(unit);
}

void produce_item(int unit) {
      avail++; item[in] = unit; in = (in + 1) mod MAXUNITS;
 }

process PC {
   int    clientID, client,  unit ; op_kind kind ;
```

```
while ( true ) {
    ClientID = receive (ANY, kind, unit); /*se bloqueante garante apenas um no monitor*/

    if ( kind == CONSUME) {
       if ( avail > 0 ) {
          unit = consume_item( ); send (ClientID, unit);
          if client = empty (pending_prod)
             send(client, NULL); /*recuperar processo pendente na fila significa acordar*/
                                 alguém que esperava por essa condição = signal      */
          }
          else
                insert(pending_cons, clientID);  /* não responder é equivalente a por em */
                                                 /* wait o processo que fez a requisição   */
          } /*end of consume */



          if (kind == PRODUCE)
            if ( avail == MAXUNITS)
              insert(pending_prod, clientID); /*espera na fila até ter condições de produzir */
            else {/* pode produzir */
                if client = empty(pending_cons) { /* retorna primeiro da fila espera p/ consumo ou NULL */
                   send(client, unit);    /*libera processo que esperava por item = signal */
                   send(clientID, NULL);
                }
                else {/* se a fila estava vazia, produz normalmente */
                    produce_item(unit);
                    send(clientID, NULL);
                 }
             }  /* end if-if-else produce */
```

```
process client_producer{
    int item;
    while ( true ) {
        item = random( );
        send (PCserver, PRODUCE, item);
        receive(PCserver, NULL);
    }
}
```

```
process client_consumer{
    int item;
    while ( true ) {
        send(PCserver, CONSUME, NULL);
        receive(PCserver, &item);
    }
}
```

4ª Questão

(a) Não é idempotente. A cada chamada write o ponteiro mantido pelo sistema operacional (file pointer) é incrementado pela quantidade de bytes escritos. Se write é chamada duas vezes consecutivas, os dado serão escritos duas vezes, um conjunto após o outro. Para tornar idempotente basta fazer com o file pointer seja mantido pela aplicação (usuário). O file pointer passa a ser um quarto argumento que indica a posição onde os bytes devem ser escritos.

(b) **Deadlock** é a situação onde um conjunto de processos ficam **bloqueados** esperando por recursos mantidos por esses procesos **em uma dependência circular**. **Livelock** é a situação onde um ou mais processos ficam **executando** laços de espera ativa (busy wait) indefinidamente. **Starvation** é a situação onde um processo fica **bloqueado** esperando por recurso, mas nunca recebe esse recurso porque processos de maior prioridade **alocam o recurso antes dele**.

5ª Questão

(a) Isso é devido a semântica do *signal* e a forma pela qual a exclusão mútua. Se for usada uma semântica *signal and continue*, o processo que fez o *signal* pode continuar a executar e o processo sinalizado ser posto em um fila de espera para entrar no monitor associado a condição que acaba de ser satisfeita. Entretanto, um terceiro processo pode entrar no monitor e alterar a condição para falso.

(b) Inversão de prioridades é quando uma thread/processo com uma dada prioridade impede a execução de um processo/thread com prioridade maior. Isso ocorre, por exemplo, quando a thread (processo) detém um mutex e é preemptada por prioridade. Se as threads (processos) de maior prioridade tentarem usar o mesmo mutex eles ficarão bloqueados esperando pela execução da thread (processo) de menor prioridade. O período de espera depende ainda da existência ou não de threads (processos) de prioridades intermediárias. As soluções possíveis são:

- aumento da prioridade da thread (processo) que mantém o mutex para o nível da thread (processo) de maior prioridade que está sendo bloqueado
- aumento da prioridade da thread (processo) que mantém o mutex para um nível de prioridade pré-estabelecido (teto).

Em ambos os casos, após o processamento, a prioridade retorna ao nível anterior.