LISTA DE EXERCÍCIOS PARA P2 – INCLUI PROVAS ATÉ 2005/2

GRAFO DE PRECEDÊNCIA

1) Desenhe o grafo de precedência do código abaixo:

```
main()
{
[1]     int f1, f2, f3;   /* Identifica processos filho*/

[2]     printf("Alo do pai\n");
[3]     f1 = fork( codigo_do_filho );          /* Cria filho 1 */
[4]     printf("Filho 1 criado\n");
[5]     f2 = fork( codigo_do_filho );          /* Cria filho 2 */
[6]     printf("Filho 2 criado\n");
[7]     wait( f1);
[8]     printf("Filho 1 morreu\n");
[9]     f3 = fork( codigo_do_filho );          /* Cria filho 3 */
[10]    printf("Filho 3 criado\n");
[11]    wait( f3);
[12]    printf("Filho 3 morreu\n");
[13]    wait( f2);
[14]    printf("Filho 2 morreu\n");
[15]    exit();
}

codigo_do_filho()
{
[16]    printf("Alo do filho\n");
[17]    exit();
}
```

SEMÁFOROS

2) O problema dos leitores/escritores consiste de um texto que pode ser lido ou escrito por vários processos. Considerando o código abaixo, responda justificando:

a) É possível vários leitores lerem ao mesmo tempo ?
b) É possível vários escritores escreverem ao mesmo tempo ?
c) É possível postergação indefinida de um leitor ?
d) É possível postergação indefinida de um escritor ?

```
int nl = 0;
semaphore tipo = 1;
semaphore exclusivo = 1;
```

void leitor(void)	void escritor(void)

{ ...
P(exclusivo);
if(nl > 0) ++ nl;
else{
P(tipo);
nl = 1;
}
V(exclusivo);
Acessa o texto
P(exclusivo);
-- nl;
if(nl == 0) V(tipo);
V(exclusivo);
...
}
	{ ...
	P(tipo);
	Acessa o texto
	V(tipo);
	...
	}

3) Considerando o código abaixo, uma variação da implementação do produtor/consumidor vista em aula, responda as seguintes perguntas (justificando):
a) Está garantido o acesso exclusivo à variável "buffer" no caso de 1 produtor e 1 consumidor ?
b) É possível a ocorrência de deadlock no caso de 2 produtores e 2 consumidores ?

```
struct  tipo_dado  buffer;
semaphore espera_vaga = 1;
semaphore espera_dado = 0;
```

void produtor(void)	void consumidor(void)

{ ...
P(espera_vaga);
buffer = dado_produzido;
V(espera_dado);
...
}
	{ ...
	P(espera_dado);
	dado_a_consumir = buffer;
	V(espera_vaga);
	...
	}

4) Considere o problema dos filósofos jantadores, onde 5 filósofos estão sentados em torno de uma mesa circular. A frente de cada um existe um prato e entre cada 2 filósofos existe 1 garfo. Cada filósofo vive um eterno ciclo de "pensa", "fica com fome", "pega os garfos", "come", "solta os garfos" e então "pensa" novamente. São necessários dois garfos para um filósofo comer.

Abaixo estão implementadas a rotina pega_garfos (chamada pelo filósofo f quando ele fica com fome) e a rotina solta_garfos (chamada pelo filósofo f quando ele termina de comer). Os filósofos são numerados de 0 a 4. A rotina esquerda(f) retorna o número do filófoso a esquerda do filósofo f. De forma semelhante, a rotina direita(f) retorna o número do filófoso a direita do filósofo f.

Análise o código abaixo e procure responder as questões abaixo, justificando através de seqüências de eventos:

Dois filósofos vizinhos podem comer simultaneamente ?
Todos os filósofos podem ficar bloqueados com fome ao mesmo tempo ?
Um filósofo pode ficar em postergação indefinida sem que todo o sistema pare ?

```
#define PENSA   0
#define FOME    1
#define COME    2
int estado[5] = { PENSA, PENSA, PENSA, PENSA, PENSA };
semaphore esperando[5] = { 0, 0, 0, 0, 0};
semaphore exclusivo = 1;
```

```
void pega_garfos(int f)
{
   P( exclusivo );
   if( estado[esquerda(f)] == PENSA
             &&
       estado[direita(f)] == PENSA )
        {
         estado[ f ] = COME;
         V( exclusivo );
         }
   else {
         estado[ f ] = FOME;
         V( exclusivo );
         P( esperando[f] );
         }
}
```

```
void solta_garfos( int f)
{
   P( exclusivo );
   estado[ f ] = PENSA;

   if( estado[esquerda(f)] == FOME
             &&
     estado[esquerda(esquerda(f) == PENSA )
        {
          estado[esquerda(f)] = COME;
          V( esperando(esquerda(f) );
          }

   if( estado[direita(f)] == FOME
             &&
     estado[direita(direita(f) == PENSA )
        {
          estado[direita(f)] = COME;
          V( esperando(direita(f) );
          }
   V( exclusivo );
}
```

5) O problema dos leitores/escritores consiste de um texto que pode ser lido ou escrito por vários processos. Considerando o código abaixo, responda justificando:

e) É possível vários leitores lerem ao mesmo tempo ?
f) É possível vários escritores escreverem ao mesmo tempo ?
g) É possível um leitor e um escritor acessarem ao mesmo tempo ?
h) É possível postergação indefinida de um escritor ?

```
int nl = 0;
semaphore tipo = 1;
semaphore exclusivo = 1;
```

```
void leitor( void)
{       ...
      P( exclusivo );
      if( nl > 0 )    ++ nl;
      else{
                  P( tipo );
                  nl = 1;
                  }
       V( exclusivo );
       Acessa o texto
       P( exclusivo );
        -- nl;
       if( nl == 0 )  V( tipo );
       V( exclusivo );
        ...
}
```

```
void escritor( void)
{       ...
      P( exclusivo );
      Acessa o texto
      V( exclusivo );
        ...
}
```

6) Usando **semáforos**, implemente rotinas cujo comportamento seja similar ao das funções wait e signal vistas em aula (pthreads) e descritas abaixo. Crie tantos semáforos e variáveis auxiliares do tipo inteiro quantos julgar necessário. Explique a lógica da solução.

WAIT(VC): Sempre bloqueia o processo até alguém executar um SIGNAL.

SIGNAL(VC): Se existe alguém bloqueado em VC, acorda apenas 1 deles.

Obs: O que é pedido são dois trechos de código em C, tal que um funcione como se fosse o "wait" e o outro funcione como se fosse o "signal". Neste código em C, para obter atomicidade e bloquear tarefas, devem ser usados semáforos.

7) Em um programa concorrente existem N processos trabalhadores e um único processo finalizador. O programa é tal que o processo finalizador precisa esperar todos os processos trabalhadores terminarem para então ele finalizar o programa. Este tipo de sincronização é chamada de "barreira" e é típica da programação em máquinas paralelas. O valor de N é conhecido.

Implemente uma solução para o problema da barreira usando semáforos. A solução consiste de duas rotinas:

/* processo trabalhador "meuPid" informa que acabou sua parte do serviço */
void acabei (int meuPid);

/* processo finalizador fica bloqueado até todos os trabalhadores acabarem o serviço */
void espera_todos(void);

Esqueleto do código dos processos:

Void trabalhador(int eu)
{
   ...
   acabei(eu);
   ...
}
	void finalizador(void)
	{
	   ...
	   espera_todos( );
	   ...
	}

PTHREADS

8) Em um programa concorrente existem N processos trabalhadores e um único processo finalizador. O programa é tal que o processo finalizador precisa esperar todos os processos trabalhadores terminarem para então ele finalizar o programa. Este tipo de sincronização é chamada de "barreira" e é típica da programação em máquinas paralelas.

Implemente uma solução para o problema da barreira usando mutexes e variáveis condição da biblioteca das pthreads. A solução consiste de 2 rotinas:

/* processo trabalhador "meuPid" informa que acabou sua parte do serviço */
void acabei (int meuPid);

/* processo finalizador fica bloqueado até todos os trabalhadores acabarem o serviço */
void espera_todos(void);

Dicas sobre o Posix:

```
pthread_mutex_t  m = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_lock( &m );
pthread_mutex_unlock( &m );

pthread_cond_t vc = PTHREAD_COND_INITIALIZER;
pthread_cond_wait( &vc , &m );
pthread_cond_signal( &vc );
```

9) Suponha a existência de um texto que é acessado por vários processos Leitores e um processo Escritor. Quando o Escritor está acessando, nenhum Leitor pode ler o texto, pois poderia acessar uma versão inconsistente. Por outro lado, é permitido que vários Leitores acessem o texto simultaneamente. Implemente a solução descrita abaixo, usando os recursos da biblioteca Pthreads. Não é necessário criar as threads, apenas mostrar o código de acesso ao texto.

Leitor:

```
Se número de leitores acessando for maior que zero
        Então  Incrementa número de leitores acessando
        Senão  Se escritor acessando
                        Então  Espera escritor sair
                        Senão  Faz Leitores acessando igual a 1
Acessa o texto
Decrementa número de leitores acessando
Se número de leitores acessando for igual a zero
        Então  Libera o texto
```

Escritor:

```
Se número de leitores acessando for maior que zero
        Então  Espera ficar livre
        Senão  Indica escritor acessando
Acessa o texto
Libera o texto
```

Dicas sobre o Posix:

```
pthread_mutex_t  m = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_lock( &m );
pthread_mutex_unlock( &m );

pthread_cond_t vc = PTHREAD_COND_INITIALIZER;
pthread_cond_wait( &vc , &m );
pthread_cond_signal( &vc );
```

10) Usando os mecanismos "mutex" e "variáveis condição" do Posix, implemente as operações P e V de um semáforo. Cada uma dessas operações deverá ser substituida por um código C com semântica similar. Lembre-se que estas operações devem ser atômicas. O valor inicial do semáforo é 1.

Dicas sobre o Posix:

```
pthread_mutex_t  m = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_lock( &m );
pthread_mutex_unlock( &m );
```

```
pthread_cond_t vc = PTHREAD_COND_INITIALIZER;
pthread_cond_wait( &vc , &m );
pthread_cond_signal( &vc );
```

11) Considere a situação de uma estrada que possui uma ponte cuja largura só comporta um carro. A estrada como um todo possui uma pista em cada sentido, mas a ponte possui uma única pista. Carros vindos do norte e carros vindos do sul chegam a ponte com uma só pista e precisam atravessa-la. Os que viajam no mesmo sentido podem atravessar a ponte ao mesmo tempo, mas os que viajam em sentido oposto precisam esperar que a ponte fique livre.

Modele os carros como threads e a ponte como um recurso, o qual pode ser utilizado simultaneamente por carros que travegam no mesmo sentido, mas nunca por carros em sentidos opostos. Implemente uma solução para este problema usando mutexes e variáveis condição da biblioteca das pthreads. A solução consiste de 2 rotinas, como mostrado abaixo:

```
/* função chamada pela thread carro para solicitar o acesso a ponte, informando o seu sentido
(NORTE ou SUL), ao retornar desta função a thread está autorizada a entrar na ponte */
        void  pede_para_entrar (int sentido);

/* função chamada pela thread carro para avisar que saiu da ponte, possivelmente liberando
processos do outro sentido */
        void avisa_que_saiu (int sentido);
```

<u>Dicas sobre o Posix:</u>

```
pthread_mutex_t  m = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_lock( &m );
pthread_mutex_unlock( &m );

pthread_cond_t vc = PTHREAD_COND_INITIALIZER;
pthread_cond_wait( &vc , &m );
pthread_cond_signal( &vc );
```

12) Considere a situação de uma parada de onibus que funciona no estilo de lotação. Os passageiros formam uma fila dentro da estação (FIFO). Quando chega o onibus com N lugares, os passageiros embarcam. Se o onibus lotou, ele parte em sua jornada. Senão, ele fica parado até que cheguem passageiros em número suficiente para lotar o onibus, quando então ele parte. Quando um passageiro chega na estação e o onibus está parado esperando passageiros, ele embarca imediatamente. Senão ele fica esperando até o ônibus chegar.

Modele as pessoas e o onibus como threads e a estação como um monitor. Implemente este sistema usando mutexes e variáveis condição da biblioteca das pthreads. A solução consiste de um monitor com 2 rotinas de entrada, como mostrado abaixo:

```
/* thread pessoa chega na estação para embarcar no onibus, fica bloqueada até seu embarque no
ônibus ser efetuado */
        void  embarca_no_onibus ( );

/* thread ônibus chega na estação para embarcar passageiros até sua lotação completa, fica
bloqueado até ficar com a lotação completa */
        void  espera_encher_onibus ( );
```

Esqueleto do código das threads:

```
void pessoa( void)
{
   ...
   Embarca_no_onibus( );
   ...
}
```

```
void onibus( void)
{
   ...
   Espera_encher_onibus( );
   ...
}
```

<u>Dicas sobre o Posix:</u>
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_lock(&m);
pthread_mutex_unlock(&m);

pthread_cond_t vc = PTHREAD_COND_INITIALIZER;
pthread_cond_wait(&vc , &m);
pthread_cond_signal(&vc);

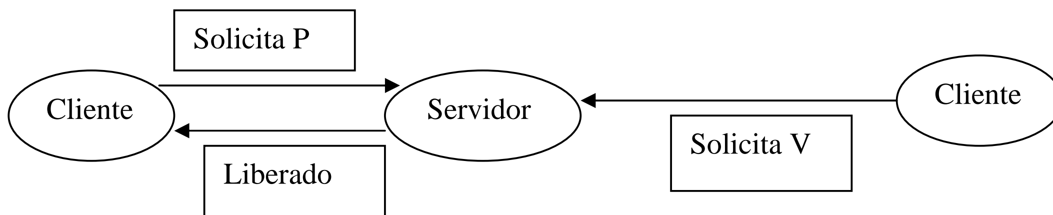
MENSAGENS

13) Em um sistema que suporta programação concorrente apenas através da troca de mensagens, será criado um Servidor para controlar o uso das portas seriais. Quando um processo Cliente deseja usar uma porta serial, ele envia uma mensagem "Aloca" para o Servidor. Existem N portas seriais, todas equivalentes, mas cada uma pode ser usada somente por um Cliente de cada vez. O Servidor informa ao Cliente a porta que ele vai usar através da mensagem "Porta p". Ao concluir o uso, o Cliente envia para o Servidor a mensagem "Libera p". Suponha que exista mais do que N processos Clientes. Mostre o algoritmo do Servidor, em português estruturado. Apresentar a solução como foi feito para o Servidor de Buffer visto em aula. Supor "receive" bloqueante.

14) Em um sistema que suporta programação concorrente apenas através da troca de mensagens, será criado um Servidor de Semáforo. Suponha que o processo S será o servidor, implementando a funcionalidade de um único semáforo inicializado com 1. O protocolo que permite a um processo cliente executar operações P e V é mostrado abaixo. Mostre o algoritmo do processo S, em português estruturado. Apresentar a solução como foi feito para o Servidor de Buffer visto em aula. Supor "receive" bloqueante.

P: --valor
Se valor < 0 então bloqueia quem executou o P

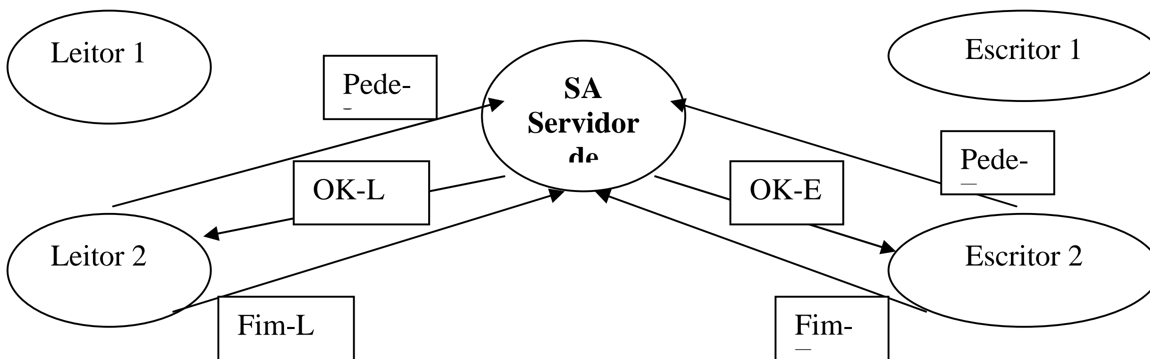
V: ++valor
Acorda o primeiro da fila, se houver alguém bloqueado



15) Em um sistema que suporta programação concorrente apenas através da troca de mensagens, será criado um **Servidor de Acesso** para controlar o acesso de vários processos a um dado arquivo. Os processos clientes podem ser do tipo Leitor (apenas lê o arquivo) ou do tipo Escritor (pode alterar o arquivo). O controle de acesso é tal que vários processos podem ler simultaneamente do arquivo, mas cada escritor deve acessar sozinho o arquivo.
- O protocolo entre Leitores/Escritores e o Servidor de Acesso é mostrado na figura.
- O esqueleto do código dos leitores e dos escritores é mostrado abaixo.

Mostre o algoritmo do Servidor de Acesso em português estruturado como feito em aula. Supor que o Servidor de Acesso é monothread e a chamada "receive()" é bloqueante.

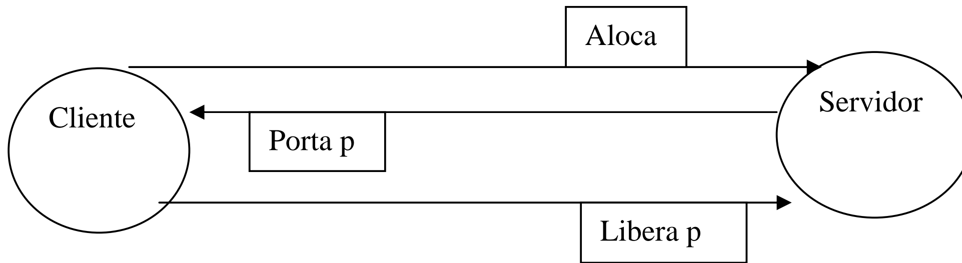
Em caso de disputa, o Servidor de Acesso poderá dar preferência para um ou outro tipo de cliente, mesmo que isto possa causar postergação indefinida do tipo de cliente preterido.



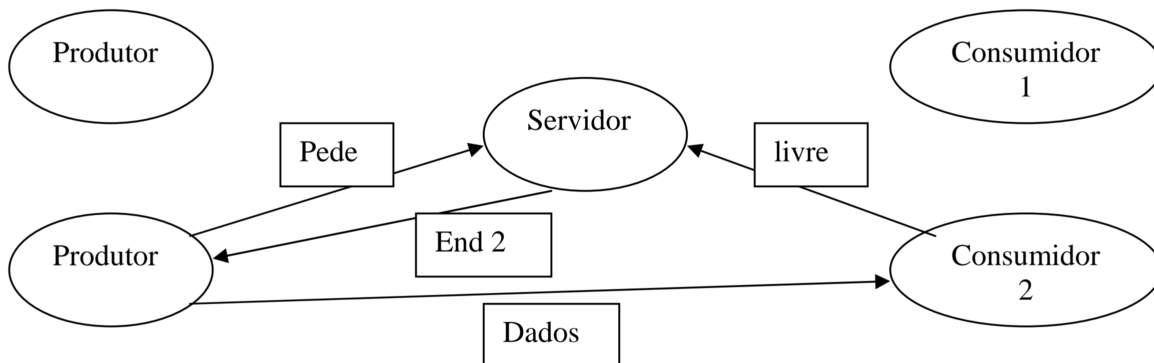
void leitor(int meu_id)	void escritor(int meu_id)

{
   While (true) {
      ...
      Send(SA, "pede-L");
      Receive(SA, "ok-L");
      ... lê ...
      Send(SA, "Fim-L");
      ...
   }
}
	{
	   While (true) {
	      ...
	      Send(SA, "pede-E");
	      Receive(SA, "ok-E");
	      ... escreve ...
	      Send(SA, "Fim-E");
	      ...
	   }
	}

16) Em um sistema que suporta programação concorrente apenas através da troca de mensagens, será criado um Servidor para controlar o uso das portas seriais. Quando um processo Cliente deseja usar uma porta serial, ele envia uma mensagem "Aloca" para o Servidor. Existem N portas seriais, todas equivalentes, mas cada uma pode ser usada somente por um Cliente de cada vez. O Servidor informa ao Cliente a porta que ele vai usar através da mensagem "Porta p". Ao concluir o uso, o Cliente envia para o Servidor a mensagem "Libera p". Suponha que exista mais do que N processos Clientes. Mostre o algoritmo do Servidor, em português estruturado. Apresentar a solução como foi feito para o Servidor de Buffer visto em aula. Supor "receive" bloqueante.



17) Em um sistema que suporta programação concorrente apenas através da troca de mensagens, será criado um Servidor para controlar o envio de dados entre produtores e consumidores. Os dados serão enviados diretamente de um Produtor para um Consumidor livre, mas o Produtor deve consultar o Servidor para obter o número do Consumidor que ele deverá usar. O Servidor deve manter uma lista com os números dos Consumidores livres no momento, e informar um desses números quando um Produtor solicita. Após receber e processar os dados, o Consumidor volta a informar que está livre para o Servidor. Mostre o algoritmo do Servidor, em português estruturado. Apresentar a solução como foi feito para o Servidor de Buffer visto em aula. Supor "receive" bloqueante.



TEMPO REAL

18) Determine se o conjunto de tarefas abaixo é escalonável com deadline monotônico (prioridades preemptivas), aplicando a análise para D ≤ P, vista em aula.

T1:	C1=2	P1=5	D1=5
T2:	C2=5	P2=25	D2=10
T3:	C3=10	P3=25	D3=25

19) Considerando o conjunto de tarefas abaixo, é possível criar uma escala de tempo preemptiva para atender os deadlines solicitados ? E uma não preemptiva ? Justifique.

T1:	C1=2	P1=5	D1=5
T2:	C2=5	P2=25	D2=10
T3:	C3=10	P3=25	D3=25

20) Determine se o conjunto de tarefas abaixo é escalonável com deadline monotônico, aplicando a análise para $D \le P$, vista em aula.

T1:	J1=0	C1=2	P1=5	D1=5	B1 = 0
T2:	J2=0	C2=5	P2=30	D2=10	B2 = 0
T3:	J3=0	C3=10	P3=25	D3=25	B3 = 0

$$W_i = C_i + B_i + \sum_{j \in HP(i)} \left\lceil \frac{J_j + W_i}{P_j} \right\rceil \times C_j \qquad R_i = J_i + W_i$$

21) Determine se o conjunto de tarefas abaixo é escalonável, aplicando a análise baseada em tempo de resposta vista em aula, quando:

(a) prioridades são atribuídas conforme o deadline monotônico;
(b) prioridades são atribuídas conforme o taxa monotônico (rate monotonic);

T1:	C1=3	P1=20	D1=6
T2:	C2=3	P2=15	D2=7
T3:	C3=4	P3=9	D3=8

22) Considerando o mesmo conjunto de tarefas, é possível criar uma escala de tempo não preemptiva para atender os deadlines solicitados ? Justifique tentando criar uma escala do tipo usada por executivo cíclico.

23) Ilustre através de exemplos como as características abaixo do sistema operacional podem aumentar o tempo de resposta de uma tarefa com alta prioridade. Para cada exemplo, desenhe um diagrama de tempo para mostrar o impacto sobre a tarefa de alta prioridade.

(a) Interrupções desabilitadas.
(b) Kernel não é preemptivo.

24) Indique dois aspectos do projeto de um sistema operacional que afetam o comportamento temporal de uma aplicação, atrapalhando o atendimento dos requisitos temporais.

25) Eventos são coletados em uma rede composta por 4 computadores, e associados com a hora local do computador que detecta cada evento. Para que os mesmos possam ser analisados em conjunto, é necessário um erro máximo entre relógios de 100 milisegundos. Os cristais usados nesses computadores possuem uma precisão de 10^{-5} em relação a sua freqüência nominal, para mais ou para menos.

Supondo que pode existir um erro inicial de 40 milissegundos entre quaisquer dois relógios da rede, calcule quanto tempo vai levar até que os dois relógios com maior erro entre eles apresentem um erro de 100 milisegundos, supondo sempre o pior caso.

26) Descreva a diferença entre deadlines hard, firm e soft, e indique um exemplo para cada um deles.

27) Indique dois aspectos do projeto de um sistema operacional que afeta o comportamento temporal de uma aplicação.

28) Qual a diferença entre tempo real crítico e tempo real brando ? Forneça também uma aplicação exemplo para cada tipo.

29) Determine se o conjunto de tarefas abaixo é escalonável se prioridades forem atribuidas segundo o deadline monotônico, aplicando a análise para D ≤ P, vista em aula. Considere que existe uma seção crítica entre T1 e T2, em função de uma tabela acessada pelas duas tarefas. A tarefa T1 permanece 1 unidade de tempo acessando a tabela, enquanto a tarefa T2 permanece 2 unidades de tempo acessando a mesma tabela.

T1:	J1=0	C1=2	P1=5	D1=5
T2:	J2=0	C2=5	P2=30	D2=10
T3:	J3=0	C3=10	P3=25	D3=25

$$W_i = C_i + B_i + \sum_{j \in HP(i)} \left\lceil \frac{J_j + W_i}{P_j} \right\rceil \times C_j \qquad R_i = J_i + W_i$$

30) Considerando o mesmo conjunto de tarefas, é possível criar uma escala de tempo não preemptiva para atender os deadlines solicitados ? Justifique tentando criar uma escala do tipo usada por executivo cíclico.

31) Considere um kernel preemptivo com as seguintes características:
- Interrupções do timer acontecem a cada 10 ms;
- O tratador de interrupções do timer demora 1 ms para executar;
- Dentro do kernel as interrupções ficam desabilitadas no máximo por 2 ms de cada vez;
- A carga do contexto de uma tarefa demora 0,1 ms e o salvamento do contexto de uma tarefa demora 0,2 ms.

Supondo que a tarefa da aplicação de mais alta prioridade execute a cada ativação um algoritmo que demora (tempo de computação) no pior caso 200 ms, qual será o seu tempo máximo de resposta ? Justifique cada termo da equação usada no cálculo.

32) Eventos são coletados em uma rede composta por 4 computadores, e associados com a hora local do computador que detecta cada evento. Para que os mesmos possam ser analisados em conjunto, é necessário um erro máximo entre relógios de 100 milisegundos. Os cristais usados nesses computadores possuem uma precisão de 10 elevado a -5 em relação a sua freqüência nominal, para mais ou para menos. Calcule quanto tempo vai levar até que os dois relógios com maior erro entre eles apresentem um erro de 100 milisegundos, supondo sempre o pior caso.

33) Considere uma aplicação composta por algumas tarefas de controle periódicas e algumas tarefas esporádicas associadas com alarmes. Compare a abordagem baseada em executivo cíclico com a abordagem baseada em prioridades com respeito ao escalonamento do sistema. Que fatores favorecem uma e outra ?

34) Determine se o conjunto de tarefas abaixo é escalonável com deadline monotônico (prioridades preemptivas), aplicando a análise para D ≤ P, vista em aula.

T1:	C1=2	P1=5	D1=5
T2:	C2=6	P2=30	D2=10
T3:	C3=10	P3=30	D3=25

$$R_i = J_i + W_i \qquad W_i = C_i + \sum_{j \in HP(i)} \left\lceil \frac{W_i + J_j}{P_j} \right\rceil \times C_j$$